Intel® Integrated Performance Primitives for Intel® Architecture

Reference Manual

Volume 1: Signal Processing

September 2007

Document Number: A24968-021US

World Wide Web: http://www.intel.com

Version	Version Information	Date
-1001	Original Issue	09/2000
-1002	Documents Intel IPP 1.0 final release. Functions NormDiff, AutoCorr, and ZeroMean have been added. Derivatives Functions section have been revised.	02/2001
-1101	Documents Intel IPP 1.1 beta release.	04/2001
-2001	Documents Intel IPP 2.0 beta release. General audio coding, MP3, and transcendental vector functions have been added. Speech recognition API have been revised.	08/2001
-2002	Documents Intel IPP 2.0 release. New Intel IPP common functions have been added. The set of arithmetic, vector initialization, statistical, and filtering functions have been expanded.	11/2001
-3001	Documents Intel IPP 3.0 pre-beta. New speech codec functions have been added. The set of conversion and windowing functions have been expanded. Speech recognition API have been updated.	04/2002
-3002	Documents Intel IPP 3.0 beta.	06/2002
-3003	Documents Intel IPP 3.0 beta update.	09/2002
-3004	Documents Intel IPP 3.0 release.	11/2002
-4001	Documents Intel IPP 4.0 beta. New functions for cross-architecture development have been added.	05/2003
-4002	Documents Intel IPP 4.0 release.	10/2003
-012	Documents Intel IPP 4.1 beta release.	04/2004
-013	Documents Intel IPP 4.1 release. Added new statistical functions and flavors for arithmetic and filtering functions.	07/2004
-014	Documents Intel IPP 5.0 beta release. New data compression functions and functions for regular expressions have been added. The set of arithmetic, conversion, transform, and filtering functions have been expanded.	03/2005
-015	Documents Intel 5.0 release. Added new data compression functions and flavors for vector initialization and Fourier transform functions.	08/2005
-016	Documents Intel IPP 5.1 beta release.	10/2005
-017	Documents Intel IPP 5.1 release. Added new flavors for arithmetic functions.	02/2006

Version	Version Information	Date
-018	Documents Intel IPP 5.2 beta release. Added new data compression fucntions for bzip2 compatibilty, new functions for the Extended AMR Wideband (AMRWB+) Speech Codec, vector conversion and initialization. Added new flavors for arithmetic functions.	09/2006
-019	Documents Intel IPP 5.2 release. Added descriptions of new conversion and statistical fucntions, SBR Encoder functions for audio coding. Added more new code examples for different functions.	01/2007
-020	Documents Intel IPP 5.3 beta release. Added descriptions of new speech coding functions (EC subband DT controller), new flavors of Goertz functions. Added new code examples.	06/2007
-021	Documents Intel IPP 5.3 release. Added descriptions of new speech coding functions (RT Audio), new arithmetic functions, new section on DTS Audio Coding.	09/2007

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

MPEG-1, MPEG-2, MPEG-4, H.261, H.263, H.264, MP3, DV, MJPEG, AC3, and AAC are international standards promoted by ISO, IEC, ITU, ETSI and other organizations. Implementations of these standards, or the standard enabled platforms may require licenses from various entities, including Intel Corporation.

This document contains information on products in the design phase of development.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2000 - 2007, Intel Corporation. All rights reserved

Contents

Chapter 1 Overview

About This Software 1-1
Hardware and Software Requirements 1-2
Platforms Supported 1-2
Cross-Architecture Alignment 1-2
Cross Architecture Overview 1-2
API Changes in Version 5.0 1-3
Technical Support 1-4
Intel IPP Code Samples 1-5
About This Manual 1-5
Manual Organization 1-5
Function Descriptions 1-6
Audience for This Manual 1-7
Online Version 1-7
Related Publications 1-7
Notational Conventions 1-7
Font Conventions 1-7
Signal Name Conventions 1-8
Naming Conventions 1-8

Chapter 2 Intel® Integrated Performance Primitives Concepts

Basic Features 2-1
Function Naming 2-2
Data-Domain 2-2
Name 2-2
Data Types 2-3
Descriptor 2-5
Parameters 2-6
Structures and Enumerators 2-7
Library Version Structure 2-7
Complex Data Structures 2-7
Function Context Structures 2-7
Enumerators 2-8
Data Ranges 2-11
Data Alignment 2-12
Integer Scaling 2-12
Error Reporting 2-13
Code Examples 2-18

Chapter 3 Support Functions

Version Information Functions 3-2
ippsGetLibVersion 3-2
Memory Allocation Functions 3-3
ippsMalloc 3-4
ippsFree 3-5
Common Functions 3-6
ippGetStatusString 3-6
ippGetCpuType 3-7
ippGetCpuClocks 3-9
ippGetCpuFreqMhz 3-9
ippGetMaxCacheSizeB 3-10
ippSetFlushToZero 3-10
ippSetDenormAreZeros 3-11
ippAlignPtr 3-12

ippSetNumThreads 3-13
ippGetNumThreads 3-14
ippMalloc 3-14
ippFree 3-15
Dispatcher Control Functions 3-16
ippStaticInit 3-16
ippStaticInitCpu 3-17

Chapter 4 Vector Initialization Functions

Vector Initialization Functions..... 4-3
Copy 4-3
PackBits 4-5
Move 4-6
Set..... 4-8
Zero 4-9
Sample-Generating Functions 4-10
Tone-Generating Functions 4-10
ToneInitAllocQ15 4-11
ToneFree 4-12
ToneGetStateSizeQ15 4-12
ToneInitQ15 4-13
ToneQ15 4-14
Tone_Direct 4-16
ToneQ15_Direct 4-17
Triangle-Generating Functions 4-19
TriangleInitAllocQ15 4-20
TriangleFree 4-22
TriangleGetStateSizeQ15 4-22
TriangleInitQ15 4-23
TriangleQ15 4-24
Triangle_Direct..... 4-25
TriangleQ15_Direct 4-27
Uniform Distribution Functions 4-29
RandUniformInitAlloc 4-29

RandUniformFree 4-30
RandUniformInit 4-31
RandUniformGetSize 4-32
RandUnifrom 4-32
RandUniform_Direct 4-34
Gaussian Distribution Functions 4-35
RandGaussInitAlloc 4-35
RandGaussFree 4-36
RandGaussGetSize 4-37
RandGaussInit 4-38
RandGauss 4-39
RandGauss_Direct 4-40
Special Vector Functions 4-41
VectorJaehne 4-41
VectorSlope 4-43
VectorRamp 4-44

Chapter 5 Essential Functions

Logical and Shift Functions 5-6
AndC 5-6
And 5-7
OrC 5-8
Or 5-9
XorC 5-10
Xor 5-11
Not 5-12
LShiftC 5-13
RShiftC 5-14
Arithmetic Functions 5-15
AddC 5-16
Add 5-18
AddProduct 5-20
AddProductC 5-21
MulC 5-22

Mul 5-24
SubC 5-27
SubCRev 5-29
Sub 5-31
DivC 5-33
DivCRev 5-35
Div 5-36
Div_Round 5-39
Abs 5-41
Sqr 5-42
Sqrt 5-44
Cubrt 5-46
Exp 5-47
Ln 5-49
10Log10 5-50
SumLn 5-52
Arctan 5-53
Normalize 5-54
Conversion Functions 5-55
SortAscend, SortDescend 5-56
SortIndexAscend, SortIndexDescend 5-57
SwapBytes 5-58
Convert 5-60
Join 5-63
JoinScaled 5-65
SplitScaled 5-66
Conj 5-67
ConjFlip 5-68
Magnitude 5-69
MagSquared 5-71
Phase 5-72
PowerSpectr 5-73
Real 5-75
Imag 5-76

RealToCplx 5-77
CplxToReal 5-78
Threshold 5-79
Threshold_LT, Threshold_GT 5-82
Threshold_LTAbs, Threshold_GTAbs 5-85
Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal 5-88
Threshold_LTInv 5-92
CartToPolar 5-94
PolarToCart 5-96
MaxOrder 5-97
Preemphasize 5-98
Flip 5-99
FindNearestOne 5-100
FindNearest 5-101
Viterbi Decoder Functions 5-102
GetVarPointDV 5-102
CalcStatesDV 5-103
BuildSymblTableDV4D 5-104
UpdatePathMetricsDV 5-105
Windowing Functions 5-106
Understanding Window Functions 5-106
WinBartlett 5-107
WinBlackman 5-109
WinHamming 5-112
WinHann 5-114
WinKaiser 5-115
Statistical Functions 5-117
Sum 5-117
Max 5-119
MaxIndx 5-120
MaxAbs 5-121
MaxAbsIndx 5-122
Min 5-123
MinIndx 5-124

MinAbs 5-125
MinAbsIndx 5-125
MinMax 5-126
MinMaxIndx 5-127
Mean 5-128
StdDev 5-130
MeanStdDev 5-132
Norm 5-133
NormDiff 5-135
DotProd 5-138
MaxEvery, MinEvery 5-141
CountInRange 5-142
Sampling Functions 5-143
SampleUp 5-143
SampleDown 5-145

Chapter 6 Filtering Functions

Convolution and Correlation Functions 6-1
Conv 6-2
ConvBiased 6-4
ConvCyclic 6-5
AutoCorr 6-6
CrossCorr 6-8
UpdateLinear 6-10
UpdatePower 6-12
Filtering Functions 6-14
SumWindow 6-17
FIR Filter Functions 6-18
FIRInitAlloc 6-19
FIRMRInitAlloc 6-21
FIRFree 6-24
FIRInit 6-25
FIRMRInit 6-27
FIRGetStateSize, FIRMRGetStateSize 6-30

FIRGetTaps 6-33
FIRSetTaps 6-34
FIRGetDlyLine 6-36
FIRSetDlyLine 6-38
FIROne 6-39
FIR 6-41
FIROne_Direct 6-47
FIR_Direct 6-50
FIRMR_Direct 6-54
FIRSparseInit 6-58
FIRSparseGetStateSize 6-59
FIRSparse 6-60
FIR Filter Coefficient Generating Functions 6-63
FIRGenLowpass 6-63
FIRGenHighpass 6-65
FIRGenBandpass 6-67
FIRGenBandstop 6-69
Single-Rate FIR LMS Filter Functions 6-70
FIRLMSInitAlloc 6-71
FIRLMSFree 6-72
FIRLMSGetTaps 6-73
FIRLMSGetDlyLine 6-74
FIRLMSSetDlyLine 6-75
FIRLMS 6-76
FIRLMSOne_Direct 6-78
Multi-Rate FIR LMS Filter Functions 6-80
FIRLMSMRInitAlloc 6-81
FIRLMSMRFree 6-82
FIRLMSMRSetMu 6-83
FIRLMSMRUpdateTaps 6-83
FIRLMSMRGetTaps 6-84
FIRLMSMRSetTaps 6-85
FIRLMSMRGetTapsPointer 6-86
FIRLMSMRGetDlyLine 6-87

FIRLMSMRSetDlyLine 6-88
FIRLMSMRGetDlyVal 6-89
FIRLMSMRPutVal 6-90
FIRLMSMROne 6-90
FIRLMSMROneVal 6-92
IIR Filter Functions 6-93
IIRInitAlloc 6-95
IIRInitAlloc_BiQuad 6-97
IIRFree 6-99
IIRInit 6-100
IIRInit_BiQuad 6-102
IIRGetStateSize 6-105
IIRGetStateSize_BiQuad 6-106
IIRSetTaps 6-107
IIRGetDlyLine 6-109
IIRSetDlyLine 6-110
IIROne 6-112
IIR 6-113
IIROne_Direct 6-120
IIROne_BiQuadDirect 6-121
IIR_Direct 6-122
IIR_BiQuadDirect 6-123
IIRSparseInit 6-124
IIRSparseGetStateSize 6-126
IIRSparse 6-127
IIRGenLowpass
IIRGenHighpass 6-130
Median Filter Functions 6-131
FilterMedian 6-131

Chapter 7 Transform Functions

Fourier Transform Functions 7-4
Special Arguments 7-4
Flag and Hint Arguments 7-4

Packed Formats 7-5
Pack Format 7-5
Perm Format 7-6
CCS Format 7-6
Format Conversion Functions 7-7
ConjPack 7-7
ConjPerm 7-9
ConjCcs 7-11
Functions for Packed Data Multiplication 7-13
MulPack 7-14
MulPerm 7-16
MulPackConj 7-18
Fast Fourier Transform Functions 7-19
FFTInitAlloc_R, FFTInitAlloc_C 7-20
FFTFree_R, FFTFree_C 7-22
FFTInit_R, FFTInit_C 7-23
FFTGetSize_R, FFTGetSize_C 7-25
FFTGetBufSize_R, FFTGetBufSize_C 7-27
FFTFwd_CToC 7-29
FFTInv_CToC 7-31
FFTFwd_RToPack, FFTFwd_RToPerm, FFTFwd_RToCCS 7-34
FFTInv_PackToR, FFTInv_PermToR, FFTInv_CCSToR 7-38
Discrete Fourier Transform Functions 7-41
DFTInitAlloc_R, DFTInitAlloc_C 7-42
DFTFree_R, DFTFree_C 7-44
DFTGetBufSize_R, DFTGetBufSize_C 7-45
DFTFwd_CToC 7-46
DFTInv_CToC 7-48
DFTFwd_RToPack, DFTFwd_RToPerm, DFTFwd_RToCCS 7-50
DFTInv_PackToR, DFTInv_PermToR, DFTInv_CCSToR 7-53
DFTOutOrdInitAlloc_C 7-56
DFTOutOrdFree_C 7-57
DFTOutOrdGetBufSize_C 7-57
DFTOutOrdFwd_CToC 7-58

DFTOutOrdInv_CToC 7-59
DFT for a Given Frequency (Goertzel) Functions 7-61
Goertz 7-61
GoertzTwo 7-63
Discrete Cosine Transform Functions 7-65
DCTFwdInitAlloc 7-65
DCTInvInitAlloc 7-66
DCTFwdFree 7-67
DCTInvFree 7-67
DCTFwdGetBufSize 7-68
DCTInvGetBufSize 7-69
DCTFwdInit 7-70
DCTInvInit 7-71
DCTFwdGetSize 7-72
DCTInvGetSize 7-73
DCTFwd 7-74
DCTInv 7-76
Hilbert Transform Functions 7-79
HilbertInitAlloc 7-79
HilbertFree 7-80
Hilbert 7-81
Wavelet Transform Functions 7-83
Transforms for Fixed Filter Banks 7-85
WTHaarFwd, WTHaarInv 7-85
Transforms for User Filter Banks 7-91
WTFwdInitAlloc, WTInvInitAlloc 7-91
WTFwdFree, WTInvFree 7-94
WTFwd 7-95
WTFwdSetDlyLine, WTFwdGetDlyLine 7-98
WTInv 7-101
WTInvSetDlyLine, WTInvGetDlyLine 7-104
Wavelet Transforms Example 7-107

Chapter 8 Speech Recognition Functions

Basic Arithmetics 8-8
AddAllRowSum 8-8
SumColumn 8-10
SumRow 8-11
SubRow 8-13
CopyColumn_Indirect 8-14
BlockDMatrixInitAlloc 8-16
BlockDMatrixFree 8-17
NthMaxElement 8-17
VecMatMul 8-18
MatVecMul 8-20
Feature Processing 8-21
ZeroMean 8-22
CompensateOffset 8-23
SignChangeRate 8-24
LinearPrediction 8-25
Durbin 8-27
Schur 8-29
LPToSpectrum 8-30
LPToCepstrum 8-31
CepstrumToLP 8-32
LPToReflection 8-33
ReflectionToLP 8-34
ReflectionToAR 8-35
ReflectionToTilt 8-36
PitchmarkToF0 8-38
UnitCurve 8-39
LPToLSP 8-40
LSPToLP 8-42
MelToLinear 8-43
LinearToMel 8-44
CopyWithPadding 8-45
MelFBankGetSize 8-46

MelFBankInit 8-47
MelFBankInitAlloc 8-48
MelLinFBankInitAlloc 8-50
EmptyFBankInitAlloc 8-53
FBankFree 8-54
FBankGetCenters 8-54
FBankSetCenters 8-55
FBankGetCoeffs 8-56
FBankSetCoeffs 8-57
EvalFBank 8-58
DCTLifterGetSize_MulC0 8-59
DCTLifterInit_MulC0 8-60
DCTLifterInitAlloc 8-61
DCTLifterFree 8-63
DCTLifter 8-63
NormEnergy 8-66
SumMeanVar 8-68
NewVar 8-70
RecSqrt 8-71
AccCovarianceMatrix 8-72
Derivative Functions 8-74
CopyColumn 8-74
EvalDelta 8-75
Delta 8-77
DeltaDelta 8-82
Pitch Super Resolution 8-87
CrossCorrCoeffDecim 8-87
CrossCorrCoeff 8-88
CrossCorrCoeffInterpolation 8-90
Model Evaluation 8-92
AddNRows 8-92
ScaleLM 8-93
LogAdd 8-94
LogSub 8-95

LogSum 8-96
MahDistSingle 8-98
MahDist 8-99
MahDistMultiMix 8-100
LogGaussSingle 8-102
LogGauss 8-105
LogGaussMultiMix 8-109
LogGaussMax 8-111
LogGaussMaxMultiMix 8-115
LogGaussAdd 8-118
LogGaussAddMultiMix 8-121
LogGaussMixture 8-123
LogGaussMixtureSelect 8-127
BuildSignTable 8-130
FillShortlist_Row 8-132
FillShortlist_Column 8-134
DTW 8-137
Model Estimation 8-139
MeanColumn 8-140
VarColumn 8-141
MeanVarColumn 8-142
WeightedMeanColumn 8-144
WeightedVarColumn 8-145
WeightedMeanVarColumn 8-147
NormalizeColumn 8-150
NormalizeInRange 8-151
MeanVarAcc 8-153
GaussianDist 8-154
GaussianSplit 8-156
GaussianMerge 8-157
Entropy 8-158
SinC 8-159
ExpNegSqr 8-160
BhatDist 8-161

UpdateMean 8-163
UpdateVar 8-164
UpdateWeight 8-165
UpdateGConst 8-166
OutProbPreCalc 8-167
DcsClustLAccumulate 8-169
DcsClustLCompute 8-170
Model Adaptation 8-172
AddMulColumn 8-172
AddMulRow 8-173
QRTransColumn 8-174
DotProdColumn 8-175
MulColumn 8-176
SumColumnAbs 8-177
SumColumnSqr 8-178
SumRowAbs 8-179
SumRowSqr 8-180
SVD, SVDSort 8-181
WeightedSum 8-182
Vector Quantization 8-184
FormVector 8-184
CdbkGetSize 8-187
CdbkInit 8-188
CdbkInitAlloc 8-190
CdbkFree 8-191
GetCdbkSize 8-192
GetCodebook 8-193
VQ 8-194
VQSingle_Sort, VQSingle_Thresh 8-195
SplitVQ 8-196
FormVectorVQ 8-198
Polyphase Resampling 8-200
ResamplePolyphaseInit 8-200
ResamplePolyphaseGetSize 8-201

ResamplePolyphaseSetFilter 8-202
ResamplePolyphaseGetFilter 8-203
ResamplePolyphaseInitAlloc 8-206
ResamplePolyphaseFree 8-208
ResamplePolyphase 8-209
Advanced Aurora Functions 8-212
SmoothedPowerSpectrumAurora 8-212
NoiseSpectrumUpdate_Aurora 8-213
WienerFilterDesign_Aurora 8-214
MelFBankInitAlloc_Aurora 8-216
TabsCalculation_Aurora 8-217
ResidualFilter_Aurora 8-217
WaveProcessing_Aurora 8-218
LowHighFilter_Aurora 8-220
HighBandCoding_Aurora 8-221
BlindEqualization_Aurora 8-223
DeltaDelta_Aurora 8-224
VADGetBufSize_Aurora 8-229
VADInit_Aurora 8-229
VADDecision_Aurora 8-230
VADFlush_Aurora 8-231
Ephraim-Malah Noise Suppressor 8-232
Noise Suppressor Architecture 8-232
Ephraim-Malah Noise Suppressor Details 8-233
Algorithm Steps 8-233
Data Structures 8-236
Filter Update Primitives 8-237
FilterUpdateEMNS 8-237
FilterUpdateWiener 8-239
Noise Floor Estimation Primitives 8-241
GetSizeMCRA 8-241
InitMCRA 8-242
AltInitMCRA 8-243
UpdateNoisePSDMCRA 8-244
Voice Activity Detector 8-244

Voice Activity Detector Architecture 8-245
Voice Activity Detection Primitives 8-246
FindPeaks 8-246
PeriodicityLSPE 8-247
Periodicity 8-248

Chapter 9 Speech Coding Functions

Rounding mode 9-2
Notational Conventions 9-3
Definitions 9-3
Data Structures 9-4
Common Functions 9-5
ConvPartial 9-6
InterpolateC_NR 9-7
Mul_NR 9-8
MulC_NR 9-10
MulPowerC_NR 9-11
AutoScale 9-12
DotProdAutoScale 9-13
InvSqrt 9-14
AutoCorr 9-14
AutoCorrLagMax 9-15
AutoCorr_NormE 9-17
CrossCorr 9-18
CrossCorrLagMax 9-19
SynthesisFilter 9-20
G.729 Related Functions 9-22
Basic Functions 9-24
DotProd_G729 9-24
Interpolate_G729 9-25
Linear Prediction Analysis Functions 9-27
AutoCorr_G729 9-27
LevinsonDurbin_G729 9-28
LPCToLSP_G729 9-30

LSFToLSP_G729 9-32
LSFQuant_G729 9-32
LSFDecode_G729 9-34
LSFDecodeErased_G729 9-35
LSPToLPC_G729 9-36
LSPQuant_G729 9-37
LSPToLSF_G729 9-40
LagWindow_G729 9-41
Codebook Search Functions 9-43
OpenLoopPitchSearch_G729 9-43
AdaptiveCodebookSearch_G729 9-46
DecodeAdaptiveVector_G729 9-49
FixedCodebookSearch_G729 9-50
GainCodebookSearch 9-54
ToeplizMatrix_G729 9-55
Codebook Gain Functions 9-57
DecodeGain_G729 9-57
GainControl_G729 9-58
GainQuant_G729 9-60
AdaptiveCodebookContribution_G729 9-62
AdaptiveCodebookGain_G729 9-63
Filter Functions 9-65
ResidualFilter_G729 9-65
SynthesisFilter_G729 9-67
LongTermPostFilter_G729 9-68
ShortTermPostFilter_G729 9-72
TiltCompensation_G729 9-73
HarmonicFilter 9-75
HighPassFilterSize_G729 9-77
HighPassFilterInit_G729 9-77
HighPassFilter_G729 9-78
IIR16s_G729 9-79
PhaseDispersionGetStateSize_G729D 9-80
PhaseDispersionInit_G729D 9-81

PhaseDispersionUpdate_G729D 9-81
PhaseDispersion_G729D 9-82
Preemphasize_G729A 9-83
WinHybridGetStateSize_G729E 9-84
WinHybridInit_G729E 9-84
WinHybrid_G729E 9-85
RandomNoiseExcitation_G729B 9-86
FilteredExcitation_G729 9-87
G.723.1 Related Functions 9-88
Linear Prediction Analysis Functions 9-89
AutoCorr_G723 9-90
AutoCorr_NormE_G723 9-91
LevinsonDurbin_G723 9-92
LPCToLSF_G723 9-93
LSFToLPC_G723 9-95
LSFDecode_G723 9-96
LSFQuant_G723 9-97
Codebook Search Functions 9-98
OpenLoopPitchSearch_G723 9-98
ACELPFixedCodebookSearch_G723 9-99
AdaptiveCodebookSearch_G723 9-101
MPMLQFixedCodebookSearch_G723 9-102
ToeplizMatrix_G723 9-104
Gain Quantization 9-105
GainQuant_G723 9-105
GainControl_G723 9-107
Filter Functions 9-108
HighPassFilter_G723 9-108
IIR16s_G723 9-109
SynthesisFilter_G723 9-110
TiltCompensation_G723 9-111
HarmonicSearch_G723 9-112
HarmonicNoiseSubtract_G723 9-113
DecodeAdaptiveVector_G723 9-114

PitchPostFilter_G723 9-116
GSM-AMR Related Functions 9-118
Basic Functions 9-120
Interpolate_GSMAMR 9-120
FFTFwd_RToPerm_GSMAMR 9-121
LP Analysis and Quantization Functions 9-122
AutoCorr_GSMAMR 9-122
LevinsonDurbin_GSMAMR 9-124
LPCToLSP_GSMAMR 9-126
LSPToLPC_GSMAMR 9-127
LSFToLSP_GSMAMR 9-129
LSPQuant_GSMAMR 9-129
QuantLSPDecode_GSMAMR 9-132
Adaptive Codebook Functions..... 9-134
Open-Loop Pitch Search (OLP) 9-134
OpenLoopPitchSearchNonDTX_GSMAMR 9-136
OpenLoopPitchSearchDTXVAD1_GSMAMR 9-138
OpenLoopPitchSearchDTXVAD2_GSMAMR 9-141
ImpulseResponseTarget_GSMAMR 9-144
AdaptiveCodebookSearch_GSMAMR 9-145
AdaptiveCodebookDecode_GSMAMR 9-148
AdaptiveCodebookGain_GSMAMR 9-150
Fixed Codebook Search 9-152
AlgebraicCodebookSearch_GSMAMR 9-152
FixedCodebookDecode_GSMAMR 9-155
Discontinuous Transmission (DTX) 9-156
Preemphasize_GSMAMR 9-156
VAD1_GSMAMR 9-157
VAD2_GSMAMR 9-160
EncDTXSID_GSMAMR 9-162
EncDTXHandler_GSMAMR 9-164
EncDTXBuffer_GSMAMR, DecDTXBuffer_GSMAMR 9-165
Post Processing 9-167
PostFilter_GSMAMR 9-167

AMR Wideband Related Functions 9-170
ResidualFilter_AMRWB 9-172
LPC Analysis Functions 9-173
LPCToISP_AMRWB 9-173
ISPToLPC_AMRWB 9-174
ISPToISF_Norm_AMRWB 9-176
ISFToISP_AMRWB 9-177
Open-loop Pitch Analysis Functions 9-178
OpenLoopPitchSearch_AMRWB 9-178
Filtering Functions 9-181
HighPassFilterGetSize_AMRWB 9-181
HighPassFilterInit_AMRWB 9-181
HighPassFilter_AMRWB 9-182
HighPassFilterGetDlyLine_AMRWB 9-184
HighPassFilterSetDlyLine_AMRWB 9-185
Preemphasize_AMRWB 9-185
Deemphasize_AMRWB 9-186
SynthesisFilter_AMRWB 9-187
Discontinuous Transmission (DTX) Functions 9-188
VADGetSize_AMRWB 9-188
VADInit_AMRWB 9-189
VAD_AMRWB 9-189
VADGetEnergyLevel_AMRWB 9-190
Codebook Search Functions 9-191
AlgebraicCodebookSearch_AMRWB 9-191
AlgebraicCodebookDecode_AMRWB 9-196
AdaptiveCodebookGainCoeff_AMRWB 9-197
AdaptiveCodebookSearch_AMRWB 9-199
AdaptiveCodebookDecodeGetSize_AMRWB 9-202
AdaptiveCodebookDecodeInit_AMRWB 9-202
AdaptiveCodebookDecodeUpdate_AMRWB 9-203
AdaptiveCodebookDecode_AMRWB 9-204
Quantization Functions 9-205
ISFQuant_AMRWB 9-205

ISFQuantDecode_AMRWB 9-207
ISFQuantDTX_AMRWB 9-208
ISFQuantDecodeDTX_AMRWB 9-209
GainQuant_AMRWB 9-209
DecodeGain_AMRWB 9-212
EncDTXBuffer_AMRWB 9-213
DecDTXBuffer_AMRWB 9-215
AMR Wideband Plus Related Functions 9-217
SNR_AMRWBE 9-219
OpenLoopPitchSearch_AMRWBE 9-220
LPCToISP_AMRWBE 9-221
Filtering Functions 9-222
SynthesisFilter_AMRWBE 9-222
Deemphasize_AMRWBE 9-223
FIRGenMidBand_AMRWBE 9-224
PostFilterLowBand_AMRWBE 9-225
Fast Fourier Transform Functions 9-226
FFTFwd_RToPerm_AMRWBE 9-226
FFTInv_PermToR_AMRWBE 9-227
Codebook Search Functions 9-228
AdaptiveCodebookSearch_AMRWBE 9-228
AdaptiveCodebookDecode_AMRWBE 9-229
Resample Functions 9-230
Downsample_AMRWBE 9-230
Upsample_AMRWBE 9-231
BandSplit_AMRWBE 9-232
BandJoin_AMRWBE 9-233
BandSplitDownsample_AMRWBE 9-234
BandJoinUpsample_AMRWBE 9-236
ResamplePolyphase_AMRWBE 9-237
Quantization Functions 9-239
ISFQuantDecode_AMRWBE 9-239
ISFQuantDecodeHighBand_AMRWBE 9-240
ISFQuantHighBand_AMRWBE 9-241

GainQuant_AMRWBE 9-242
QuantTCX_AMRWBE 9-243
GainQuantTCX_AMRWBE 9-244
GainDecodeTCX_AMRWBE 9-246
EncodeMux_AMRWBE 9-247
DecodeDemux_AMRWBE 9-248
GSM Full Rate Related Functions..... 9-249
RPEQuantDecode_GSMFR 9-249
Deemphasize_GSMFR 9-250
ShortTermAnalysisFilter_GSMFR 9-251
ShortTermSynthesisFilter_GSMFR 9-252
HighPassFilter_GSMFR 9-253
Schur_GSMFR 9-254
WeightingFilter_GSMFR 9-255
Preemphasize_GSMFR 9-256
G.722.1 Related Functions..... 9-257
DCTFwd_G722, DCTInv_G722 9-257
DecomposeMLTToDCT 9-258
DecomposeDCTToMLT 9-260
HuffmanEncode_G722 9-261
G.726 Related Functions..... 9-262
EncodeGetStateSize_G726 9-262
EncodeInit_G726 9-263
Encode_G726 9-264
DecodeGetStateSize_G726 9-265
DecodeInit_G726 9-265
Decode_G726 9-266
G.728 Related Functions..... 9-268
IIRGetStateSize_G728 9-269
IIR_Init_G728 9-269
IIR_G728 9-270
SynthesisFilterGetStateSize_G728 9-271
SynthesisFilterInit_G728 9-272
SyntesisFilter_G728 9-272

CombinedFilterGetStateSize_G728 9-273
CombinedFilterInit_G728 9-274
CombinedFilter_G728 9-274
PostFilterGetStateSize_G728 9-276
PostFilterInit_G728 9-276
PostFilter_G728 9-277
PostFilterAdapterGetStateSize_G728 9-278
PostFilterAdapterStateInit_G728 9-279
LPCInverseFilter_G728 9-279
PitchPeriodExtraction_G728 9-280
WinHybridGetStateSize_G728 9-282
WinHybridInit_G728 9-283
WinHybrid_G728 9-284
LevinsonDurbin_G728 9-286
CodebookSearch_G728 9-287
ImpulseResponseEnergy_G728 9-288
Echo Canceller Related Functions 9-289
SubbandProcessGetSize 9-291
SubbandProcessInit 9-292
SubbandAnalysis 9-293
SubbandSynthesis 9-294
SubbandControllerGetSize_EC 9-295
SubbandControllerInit_EC 9-296
SubbandControllerUpdate_EC 9-297
SubbandController_EC 9-298
SubbandControllerReset_EC 9-300
SubbandControllerDTGetSize_EC 9-300
SubbandControllerDTInit_EC 9-301
SubbandControllerDTReset_EC 9-302
SubbandControllerDT_EC 9-303
SubbandControllerDTUpdate_EC 9-305
ToneDetectGetStateSize_EC 9-306
ToneDetectInit_EC 9-307
ToneDetect_EC 9-308

FullbandControllerGetSize_EC 9-309
FullbandControllerInit_EC 9-310
FullbandControllerUpdate_EC 9-311
FullbandController_EC 9-312
FullbandControllerReset_EC 9-313
FIR_EC 9-314
FIRSubband_EC, FIRSubbandLow_EC 9-315
FIRSubbandCoeffUpdate_EC, FIRSubbandLowCoeffUpdate_EC .. 9-316
NLMS_EC 9-318
G722 Sub-Band ADPCM Speech Codec Related Functions 9-319
SBADPCMEncodeStateSize_G722 9-320
SBADPCMEncodeInit_G722 9-320
SBADPCMEncode_G722 9-321
QMFEncode_G722 9-322
SBADPCMDecodeStateSize_G722 9-323
SBADPCMDecodeInit_G722 9-323
SBADPCMDecode_G722 9-324
QMFDecode_G722 9-325
Companding Functions 9-327
MuLawToLin 9-327
LinToMuLaw 9-328
ALawToLin 9-329
LinToALaw 9-330
MuLawToALaw 9-331
ALawToMuLaw 9-332
RT Audio Related Functions 9-334
LPCToLSP_RTA 9-335
LSPToLPC_RTA 9-335
LevinsonDurbin_RTA 9-336
QMFGetStateSize_RTA 9-337
QMFInit_RTA 9-338
QMFDecode_RTA 9-338
QMFEncode_RTA 9-339
PostFilterGetStateSize_RTA 9-340

PostFilterInit_RTA 9-341
PostFilter_RTA 9-341
AdaptiveCodebookSearch_RTA 9-342
FixedCodebookSearch_RTA, FixedCodebookSearchRandom_RTA.. 9-343
LSPQuant_RTA 9-345
HighPassFilter_RTA 9-346
BandPassFilter_RTA 9-347

Chapter 10 Audio Coding Functions

Interleaved to Multi-Row Format Conversion Functions 10-9
Interleave 10-9
Deinterleave 10-11
Spectral Data Prequantization Functions 10-13
Pow34 10-13
Pow43 10-16
Pow43Scale 10-17
Scale Factors Calculation Functions 10-20
CalcSF 10-20
Mantissa Conversion and Scaling Functions 10-22
Scale 10-22
MakeFloat 10-24
Modified Discrete Cosine Transform Functions..... 10-25
MDCTFwdInitAlloc, MDCTInvInitAlloc 10-25
MDCTFwdInit, MDCTInvInit 10-27
MDCTFwdFree, MDCTInvFree 10-28
MDCTFwdGetSize, MDCTInvGetSize 10-29
MDCTFwdGetBufSize, MDCTInvGetBufSize 10-30
MDCTFwd, MDCTInv 10-31
Block Filtering Functions..... 10-34
FIRBlockInitAlloc 10-34
FIRBlockFree 10-35
FIRBlockOne 10-36
Frequency Domain Prediction Functions 10-38
FDPInitAlloc 10-38

FDPInit 10-39
FDPFree 10-40
FDPGetSize 10-40
FDPReset 10-41
FDPResetSfb 10-42
FDPResetGroup 10-43
FDPFwd 10-44
FDPInv 10-45
VLC Functions 10-47
VLCDecodeEscBlock_MP3 10-47
VLCDecodeEscBlock_AAC 10-50
VLCDecodeUTupleEscBlock_MP3 10-52
VLCDecodeUTupleEscBlock_AAC 10-53
VLCCountEscBits_MP3 10-55
VLCCountEscBits_AAC 10-56
VLCEncodeEscBlock_MP3 10-57
VLCEncodeEscBlock_AAC 10-58
Psychoacoustic Functions 10-60
Spread 10-60
Vector Quantization Functions 10-61
VQCodeBookInitAlloc 10-61
VQCodeBookInit 10-62
VQCodeBookFree 10-63
VQCodeBookGetSize 10-64
VQPreliminarySelect 10-64
VQMainSelect 10-66
VQIndexSelect 10-68
VQReconstruction 10-71
MP3 Audio Coding Functions 10-72
Macros and Constants 10-72
Data Structures 10-73
Frame Header 10-73
Side Information 10-74
MP3 Huffman Table Structure 10-75

MP3 Psychoacoustic Model Two Analysis 10-75
Psychoacoustic Model Two State 10-76
MP3 Bit Reservoir 10-77
MP3 Codec Enumerated Types 10-78
MP3 Audio Encoder 10-79
AnalysisPQMF_MP3 10-80
AnalysisFilterInit_PQMF_MP3 10-81
AnalysisFilterInitAlloc_PQMF_MP3 10-82
AnalysisFilterGetSize_PQMF_MP3 10-82
AnalysisFilterFree_PQMF_MP3 10-83
AnalysisFilter_PQMF_MP3 10-84
MDCTFwd_MP3 10-85
PsychoacousticModelTwo_MP3 10-86
JointStereoEncode_MP3 10-91
Quantize_MP3 10-94
PackScaleFactors_MP3 10-99
HuffmanEncode_MP3 10-102
PackFrameHeader_MP3 10-104
PackSideInfo_MP3 10-105
BitReservoirInit_MP3 10-107
MP3 Audio Decoder 10-108
UnpackFrameHeader_MP3 10-109
UnpackSideInfo_MP3 10-110
UnpackScaleFactors_MP3 10-111
HuffmanDecode_MP3, HuffmanDecodeSfb_MP3, HuffmanDecodeSfbMbp_MP3 10-113
ReQuantize_MP3, ReQuantizeSfb_MP3 10-115
MDCTInv_MP3 10-117
SynthPQMF_MP3 10-119
SynthesisFilterInit_PQMF_MP3 10-120
SynthesisFilterInitAlloc_PQMF_MP3 10-121
SynthesisFilterGetSize_PQMF_MP3 10-122
SynthesisFilterFree_PQMF_MP3 10-122
SynthesisFilter_PQMF_MP3 10-123

Advanced Audio Coding Functions 10-125
Global Macros 10-125
Data Types and Structures 10-126
ADIF Header 10-126
ADTS Frame Header 10-127
Individual Channel Side Information 10-128
AAC Scalable Main Element Header 10-129
AAC Scalable Extension Element Header 10-129
AAC Program Config Element 10-129
Temporal Noise Shaping (TNS) Structure for One Layer 10-131
Long Term Prediction (LTP) Structure 10-132
Channel Pair Element 10-132
Channel Information 10-133
AAC Decoding Primitives 10-134
MPEG-2 AAC Primitives 10-136
UnpackADIFHeader_AAC 10-136
UnpackADTSFrameHeader_AAC 10-137
DecodePrgCfgElt_AAC 10-138
DecodeChanPairElt_AAC 10-139
NoiselessDecoder_LC_AAC 10-140
DecodeDatStrElt_AAC 10-144
DecodeFillElt_AAC 10-145
QuantInv_AAC 10-146
DecodeMsStereo_AAC 10-148
DecodeIsStereo_AAC 10-150
DeinterleaveSpectrum_AAC 10-151
DecodeTNS_AAC 10-153
MDCTInv_AAC_32s16s 10-157
MDCTInv_AAC_32s_I 10-159
MPEG-4 AAC Primitives 10-160
DecodeMainHeader_AAC 10-160
DecodeExtensionHeader_AAC 10-161
DecodePNS_AAC 10-162
DecodeMsPNS_AAC 10-164

DecodeChanPairElt_MP4_AAC 10-165
LongTermReconstruct_AAC 10-166
MDCTFwd_AAC_32s 10-167
MDCTFwd_AAC_32s_I 10-168
EncodeTNS_AAC 10-170
LongTermPredict_AAC 10-172
NoiselessDecode_AAC 10-173
LtpUpdate_AAC 10-174
Spectral Band Replication Functions 10-176
SBR Audio Encoder Functions 10-176
DetectTransient_SBR 10-177
EstimateTNR_SBR 10-179
AnalysisFilterEncGetSize_SBR 10-181
AnalysisFilterEncInit_SBR 10-181
AnalysisFilterEnc_SBR 10-182
AnalysisFilterEncInitAlloc_SBR 10-183
AnalysisFilterEncFree_SBR 10-184
SBR Audio Decoder Functions 10-185
QMF Functions 10-185
AnalysisFilterInitAlloc_SBR 10-187
SynthesisFilterInitAlloc_SBR 10-188
SynthesisDownFilterInitAlloc_SBR 10-189
AnalysisFilterFree_SBR 10-190
SynthesisFilterFree_SBR 10-191
SynthesisDownFilterFree_SBR 10-192
AnalysisFilterGetSize_SBR 10-193
SynthesisFilterGetSize_SBR 10-194
SynthesisDownFilterGetSize_SBR 10-196
AnalysisFilterInit_SBR 10-198
SynthesisFilterInit_SBR 10-199
SynthesisDownFilterInit_SBR 10-201
AnalysisFilter_SBR 10-203
SynthesisFilter_SBR 10-207
SynthesisDownFilter_SBR 10-211

Prediction Functions 10-217
PredictCoef_SBR 10-218
PredictOneCoef_SBR 10-220
DTS Audio Coding Functions 10-222
DTS Audio Decoder 10-222
SynthesisFilterInit_DTS 10-222
SynthesisFilterInitAlloc_DTS 10-223
SynthesisFilterGetSize_DTS 10-224
SynthesisFilterFree_DTS 10-224
SynthesisFilter_DTS 10-225

Chapter 11 String Functions

String Manipulation 11-2
Find,FindRev 11-3
FindC, FindRevC 11-5
FindCAny, FindRevCAny 11-6
Insert 11-8
Remove 11-9
Compare 11-10
CompareIgnoreCase, CompareIgnoreCaseLatin 11-13
Equal 11-14
TrimC 11-15
TrimCAny, TrimStartCAny, TrimEndCAny 11-16
ReplaceC 11-17
Uppercase, UppercaseLatin 11-18
Lowercase, LowercaseLatin 11-19
Hash 11-21
Concat 11-22
ConcatC 11-23
SplitC 11-24
Regular Expressions 11-25
RegExpInitAlloc 11-26
RegExpFree 11-27
RegExpInit 11-27

RegExpGetSize 11-29
RegExpSetMatchLimit 11-29
RegExpFind 11-30

Chapter 12 Fixed-Accuracy Arithmetic Functions

Power and Root Functions 12-5
Inv 12-5
Div 12-7
Sqrt 12-9
InvSqrt 12-11
Cbrt 12-12
InvCbrt 12-14
Pow 12-16
Powx 12-19
Hypot 12-21
Exponential and Logarithmic Functions 12-24
Exp 12-24
Ln 12-26
Log10 12-28
Trigonometric Functions 12-30
Cos 12-30
Sin 12-32
SinCos 12-34
Tan 12-36
Acos 12-38
Asin 12-40
Atan 12-42
Atan2 12-44
Hyperbolic Functions 12-46
Cosh 12-46
Sinh 12-48
Tanh 12-50
Acosh 12-52
Asinh 12-54

Atanh 12-56
Special Functions 12-58
Erf 12-58
Erfc 12-60
ErfInv 12-61
Rounding Functions 12-64
Floor 12-64
Ceil 12-65
Trunc 12-66
Round 12-67
NearbyInt 12-68
Rint 12-70
Modf 12-71

Chapter 13 Data Compression Functions

Application Notes 13-1
VLC and Huffman Coding Functions..... 13-3
Data Compression VLC functions 13-4
VLCEncodeInitAlloc 13-5
VLCEncodeFree 13-6
VLCEncodeInit 13-6
VLCEncodeGetSize 13-7
VLCEncodeBlock 13-8
VLCEncodeOne 13-9
VLCCountBits 13-10
VLCDecodeInitAlloc 13-11
VLCDecodeFree 13-12
VLCDecodeInit 13-12
VLCDecodeGetSize 13-14
VLCDecodeBlock 13-15
VLCDecodeOne 13-16
Example of Using VLC Coding Functions..... 13-17
VLCDecodeUTupleInitAlloc 13-18
VLCDecodeUTupleFree 13-19

VLCDecodeUTupleInit 13-20
VLCDecodeUTupleGetSize 13-21
VLCDecodeUTupleBlock 13-22
VLCDecodeUTupleOne 13-23
Huffman Coding 13-25
EncodeHuffInitAlloc 13-26
HuffFree 13-27
EncodeHuffInit 13-28
HuffGetSize 13-29
EncodeHuffOne 13-29
EncodeHuff 13-30
EncodeHuffFinal 13-31
HuffGetLenCodeTable 13-32
DecodeHuffInitAlloc 13-32
DecodeHuffInit 13-33
DecodeHuffOne 13-34
DecodeHuff 13-35
HuffGetDstBuffSize 13-36
Application Notes..... 13-36
HuffLenCodeTablePack 13-37
HuffLenCodeTableUnpack 13-38
Dictionary-Based Compression Functions 13-39
LZSS Compression Functions 13-40
EncodeLZSSInitAlloc 13-41
LZSSFree 13-41
EncodeLZSSInit 13-42
LZSSGetSize 13-42
EncodeLZSS 13-43
EncodeLZSSFlush 13-44
DecodeLZSSInitAlloc 13-47
DecodeLZSSInit 13-47
DecodeLZSS 13-48
ZLIB Coding Functions 13-50
Special Parameters 13-51

EncodeLZ77Init 13-56
EncodeLZ77GetSize 13-57
EncodeLZ77InitAlloc 13-57
LZ77Free 13-58
EncodeLZ77 13-59
EncodeLZ77SelectHuffMode 13-60
EncodeLZ77FixedHuff 13-61
EncodeLZ77DynamicHuff 13-62
EncodeLZ77StoredBlock 13-63
EncodeLZ77Flush 13-64
EncodeLZ77GetPairs 13-65
EncodeLZ77SetPairs 13-66
EncodeLZ77GetStatus 13-67
EncodeLZ77SetStatus 13-67
EncodeLZ77Reset 13-68
DecodeLZ77Init 13-69
DecodeLZ77GetSize 13-69
DecodeLZ77InitAlloc 13-70
DecodeLZ77 13-71
DecodeLZ77GetBlockType 13-72
DecodeLZ77FixedHuff 13-73
DecodeLZ77DynamicHuff 13-74
DecodeLZ77StoredBlock 13-75
DecodeLZ77GetPairs 13-76
DecodeLZ77SetPairs 13-77
DecodeLZ77GetStatus 13-78
DecodeLZ77SetStatus 13-78
DecodeLZ77Reset 13-79
Adler32 13-80
CRC32 13-80
Example of Using Intel IPP ZLIB Coding Functions..... 13-81
IBWT-Based Compression Functions..... 13-83
Burrows-Wheeler Transform 13-85

BWTFwdGetSize 13-87
BWTFwd 13-88
BWTInvGetSize 13-89
BWTInv 13-89
BWTGetSize_SmalBlock 13-91
BWTFwd_SmallBlock 13-91
BWTInv_SmallBlock 13-92
Generalized Interval Transformation Coding 13-93
Special Arguments..... 13-93
EncodeGITInitAlloc 13-94
GITFree 13-95
EncodeGITInit 13-95
EncodeGITGetSize 13-96
EncodeGIT 13-97
DecodeGITInitAlloc 13-100
DecodeGITInit 13-101
DecodeGITGetSize 13-102
DecodeGIT 13-102
Move To Front Functions..... 13-105
MTFInitAlloc 13-106
MTFFree 13-107
MTFInit 13-107
MTFGetSize 13-108
MTFFwd 13-108
MTFInv 13-109
Run Length Encoding Functions 13-110
EncodeRLE 13-110
DecodeRLE 13-111
bzip2 Coding Functions 13-114
EncodeRLEInitAlloc_BZ2 13-114
RLEFree_BZ2 13-114
EncodeRLEInit_BZ2 13-115
RLEGetSize_BZ2 13-115
EncodeRLE_BZ2 13-116

EncodeRLEFlush_BZ2 13-117
RLEGetInUseTable 13-118
DecodeRLE_BZ2 13-118
EncodeZ1Z2_BZ2 13-119
DecodeZ1Z2_BZ2 13-120
ReduceDictionary 13-121
ExpandDictionary 13-122
CRC32_BZ2 13-122
EncodeHuffGetSize_BZ2 13-123
EncodeHuffInit_BZ2 13-124
EncodeHuffInitAlloc_BZ2 13-125
EncodeHuffFree_BZ2 13-126
PackHuffContext_BZ2 13-126
EncodeHuff_BZ2 13-127
DecodeHuffGetSize_BZ2 13-128
DecodeHuffInit_BZ2 13-129
DecodeHuffInitAlloc_BZ2 13-130
DecodeHuffFree_BZ2 13-130
UnpackHuffContext_BZ2 13-131
DecodeHuff_BZ2 13-132

Appendix A Handling of Special Cases

Appendix B Removed Functions

Bibliography

Glossary

Index

Overview

1

This document describes the structure, operation and functions of the Intel® Integrated Performance Primitives (Intel® IPP) for Intel® architecture that operate on one-dimensional signals. This is the first volume of the Intel IPP Reference Manual, which also comprises descriptions of the Intel IPP for image and video processing (volume 2), operations on small matrices, 3D data processing and rendering (volume 3), and cryptography functions (volume 4).
The Intel IPP software package supports many functions whose performance can be significantly enhanced on Intel architecture, particularly using the MMX™ technology, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), as well as Intel® Itanium® architecture. For information on Intel IPP implementation for Intel® IXP4XX product line, see Cross-Architecture Alignment section later in this chapter.

The Intel IPP for signal processing software is a collection of low-overhead, high-performance operations performed on one-dimensional (1D) data arrays.

This manual explains the Intel IPP concepts as well as specific data type definitions and operation models used in the signal processing domain and provides detailed descriptions of the Intel IPP signal processing functions.

This chapter introduces the Intel IPP software and explains the organization of this manual.

About This Software

The Intel IPP for Intel architecture software enables taking advantage of the parallelism of the single-instruction, multiple-data (SIMD) instructions that comprise the core of the MMX technology and Streaming SIMD Extensions. These technologies improve the performance of computation-intensive signal, image, and video processing applications. Use of Intel IPP

primitive functions can help to drastically reduce development costs and accelerate time-to-market by eliminating the need of writing processor-specific code for computation intensive routines.

Hardware and Software Requirements

Intel IPP for Intel architecture software runs on personal computers that are based on processors using IA-32, Intel® 64 or IA-64 architecture and running Microsoft Windows*, Linux*, or Mac OS* X. Intel IPP integrates into the customer's application or library written in C or C++.

Platforms Supported

Intel IPP for Intel architecture software runs on Windows*, Linux*, and Mac OS* X platforms. The code and syntax used in this manual for function and variable declarations are written in the ANSI C style. However, versions of Intel IPP for different processors or operating systems may, of necessity, vary slightly.

Cross-Architecture Alignment

Cross Architecture Overview

Intel IPP has been designed to support application development on various Intel® architectures.

By providing a single cross-architecture API, Intel IPP allows software application repurposing and enables developers to port to unique features across Intel® processor-based desktop, server, mobile and handheld platforms. Developers can write their code once, to realize application performance over many processor generations.

The Table 1-1 summarizes the functionality covered in each Intel IPP implementation. Only integer flavors of the most functions described in this manual are supported for Intel® IXP4XX product line. Please refer to the corresponding header files.

For additional information on API additions and changes from previous versions, please refer to the product release notes and the document *Migration Guide for Intel® IPP Cross Architecture API Alignment* available at http://www.intel.com/software/products/ipp.

The following table summarizes the functionality covered in each Intel IPP implementation.

Table 1-1 Function Coverage in Intel IPP

Function Group	Intel® Pentium® 4 processors	64-bit Intel® Xeon® processors	Intel® Itanium® 2 processors	Intel® IXP4XX Product Line
Signal Processing	available	available	available	available[1]
Image Processing	available	available	available	available[1]
JPEG	available	available	available	available[1]
Speech Recognition	available	available	available	available[1]
Speech Coding	available	available	available	available[1]
Audio Codecs	available	available	available	available[1]
Video Codecs	available	available	available	available[1]
Matrix	available	available	available	n/a
Vector Math	available	available	available	n/a
Computer Vision	available	available	available	available[1]
Cryptography	available	available	available	available[1]
Data Compression	available	available	available	available[1]
Color Conversion	available	available	available	available[1]
String Processing	available	available	available	available[1]
Realistic Rendering	available	available	available	n/a

1. Only a subset of these functions is supported, please refer to the corresponding header file.

API Changes in Version 5.0

Starting from the version 5.0 Intel IPP has a limited compatibility with the versions 4.x or lower. To improve the library in several aspects the following changes have been done:

- some functions have been replaced by new functions with extended functionality;
- some functions have been changed to make API more consistent and handy;
- odd and unusable functions have been discarded;
- several complicated functions have been discarded from the Intel IPP, but their functionalities have been moved to the codec/sample level.

All these changes affect existent applications. Table 1 in the Appendix B lists such functions for the signal processing functional domains and specifies the corresponding Intel IPP 5.0 functions to substitute for them. If an application calls functions listed in this table, then the source code should be modified.

Several functions have been moved to the newly created domains. These changes do not affect existent applications.

Additionally, certain modifications breaking binary compatibility have been made in the Intel IPP 5.0, for example, directory tree structure has been modified, the `ipp20` folder has been discarded. These changes also affect existent applications that should be at least rebuilt.

The version 5.1 as well as the next following versions has full backward compatibility with the version 5.0.

Technical Support

Intel IPP provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: http://developer.intel.com/software/products/.

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, and more (visit http://support.intel.com/support/).

Registering your product entitles you to one-year technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing the following services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, or contact Intel, or seek product support, please visit: http://www.intel.com/software/products/support

Intel IPP Code Samples

An extensive library of code samples and codecs have been implemented using the Intel IPP functions to help demonstrate the use of Intel IPP and to help accelerate the development of your application, components and codecs. The samples can be downloaded form http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

About This Manual

This manual provides a background for the signal processing concepts used in the Intel IPP software as well as detailed descriptions of the Intel IPP for Intel architecture signal processing functions.

NOTE. In the following discussion throughout the manual, Intel IPP always refers to the Intel Integrated Performance Primitives for Intel architecture, unless specifically stated otherwise.
For description of Intel PCA version of Intel IPP, refer to the Intel IPP for Intel XScale® microarchitecture Reference Manual.

The Intel IPP functions are combined in groups by their functionality. Each group of functions is described in a separate chapter (chapters 3 through 12).

Manual Organization

This manual contains the following chapters:

Chapter 1 "Overview." Introduces the Intel IPP software, provides information on manual organization, and explains notational conventions.

Chapter 2 "Intel Integrated Performance Primitives Concepts." Explains the basic concepts underlying signal processing in Intel IPP and describes the supported data formats and operation modes.

Chapter 3 "Support Functions." Describes functions used to manipulate the signals, such as `Copy` and `Set`. Also describes data conversion and memory allocation functions.

Chapter 4 "Vector Initialization Functions." Describes initialization and sample-generating functions.

Chapter 5 "Essential Functions." Describes vector manipulation functions.

Chapter 6 "Filtering Functions." Detais filtering operations that use linear and non-linear filters.

Chapter 7 "Transform Functions." Describes domain transform functions (Fourier, Wavelet, Cosine).

Chapter 8 "Speech Recognition Functions." Describes functions used in the speech recognition applications.

Chapter 9 "Speech Coding Functions." Describes functions used as buiding blocks for implementing speech codecs.

Chapter 10 "Audio Coding Functions." Includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio codec, MP3 encoder and decoder, for implementation of a portable optimized MPEG-4 AAC Main profile decoder and a portable optimized MPEG-1, 2 Layer III encoder. Also includes Spectral Band Replication (SBR) functions as an audio coding bandwidth extension technology.

Chapter 11 "String Functions." Describes functions that operate on strings of symbols.

Chapter 12 "Fixed-Accuracy Arithmetic Functions." Describes Intel IPP fixed-accuracy transcendental mathematical real and complex functions of vector arguments. The functions are suitable for multimedia and signal processing in real time applications.

Chapter 13 Data Compression Functions. Describes Intel IPP data compression functions that support a number of different conversion methods.

The manual also includes an Appendix, a Glossary of terms, a Bibliography, and an Index.

Function Descriptions

In Chapters 3 through 12, each function is introduced by its short name (without the `ipps` prefix and modifiers) and a brief description of its purpose. This is followed by the function call sequence, definition of its parameters (arguments), and more detailed explanation of the function's purpose. The following sections are included in function description:

Syntax	Lists function prototypes
Parameters	Describes all the function parameters (arguments).
Description	Defines the function and describes the operation performed by the function. This section may also include the code examples and descriptive equations.
Application Notes	If present, describes any special information which application programmers or other users of the function need to know.
Return Values	Explains the value returned by the function. Most commonly, it lists error codes that the function returns.

See Also If present, lists the names of functions which perform related tasks.

Audience for This Manual

The manual is intended for the developers of signal processing applications and libraries, as well as cross-domain applications. The audience is expected to be experienced in using C and to have a working knowledge of the vocabulary and principles of signal processing.

Online Version

This manual is available in an electronic format (Portable Document Format, or PDF). To obtain a hard copy of the manual, print the file using the printing capability of Adobe Acrobat*, the tool used for the online presentation of the document.

Related Publications

For more information about signal processing concepts and algorithms, refer to the books and papers listed in the Bibliography.

Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code;
- Signal name conventions;
- Naming conventions for different items.

Font Conventions

The following font conventions are used throughout the manual:

`THIS TYPE STYLE` Used in the text for Intel IPP constant identifiers; for example, `IPP_MAX_64S`.

`This type style`	Mixed with the uppercase in structure names as in `IppLibraryVersion`; also used in function names, code examples and call statements; for example, `void ippsFree()`.
`This type style`	Variables in arguments and parameters description; for example, *`val`*, *`srcLen`*.

Signal Name Conventions

In this manual, vectors and arrays are commonly used to represent a discrete 1D signal. The notation $x(n)$ refers to a conceptual signal, while the notation $x[n]$ refers to an actual vector. Both of these are annotated to indicate a specific finite range of values:

$x[n], \quad 0 \le n < len$.

Typically, the number of elements in vectors is denoted by *len*. Vector names contain square brackets as distinct from vector elements with current index *n*.

For example, the expression *pDst*[*n*] = *pSrc*[*n*] + *val* implies that each element *pDst*[*n*] of the vector *pDst* is computed for each *n* in the range from 0 to *len*-1. Special cases are regarded and described separately.

Naming Conventions

The following naming conventions for different items are used by the Intel IPP software:

- Constant identifiers are in uppercase; for example, `IPP_MIN_64S`.
- All structures and enumerators, specific for the signal processing domain have the `Ipps` prefix, while those common for entire Intel IPP software have the `Ipp` prefix; for example, `IppsROI`, `IppLibraryVersion`.
- All names of the functions used for signal processing have the `ipps` prefix. In code examples, you can distinguish the Intel IPP interface functions from the application functions by this prefix.

NOTE. In this manual, the `ipps` prefix in function names is always used in the code examples. In the text, this prefix is usually omitted when referring to the function group.

- Each new part of a function name starts with an uppercase character, without underscore; for example, `ippsAddAllRowSum`.

For the detailed description of function name structure in Intel IPP, see “Function Naming” on page 2-2.

Intel® Integrated Performance Primitives Concepts

2

This chapter explains the purpose and structure of the Intel® Integrated Performance Primitives (Intel® IPP) software and looks over some of the basic concepts used in the signal processing part of Intel IPP. It also describes the supported data formats and operation modes, and defines function naming conventions in the manual.

Basic Features

The Intel Integrated Performance Primitives, like other members of the Intel® Performance Libraries, is a collection of high-performance code that performs domain-specific operations. It is distinguished by providing a low-level, stateless interface.

Based on experience in developing and using Intel Performance Libraries, Intel IPP has the following major distinctive features:

- The Intel IPP provides basic low-level functions for creating applications in several different domains, such as signal processing, image and video processing, operations on small matrices, and cryptography applications;
- Intel IPP functions follow the same interface conventions including uniform naming rules and similar composition of prototypes for primitives that refer to different application domains;
- Intel IPP functions use abstraction level which is best suited to achieve superior performance figures by the application programs.

To speed up performance, Intel IPP functions are optimized to use all benefits of Intel® architecture processors. Besides that, most of Intel IPP functions do not use complicated data structures, which helps reduce overall execution overhead.

Intel IPP is well-suited for cross-platform applications. For example, the functions developed for IA-32 platform can be readily ported to Intel® Itanium®-based platforms or systems based on the Intel® IXP4XX product line.

For more information on platform compatibility, see Cross-Architecture Alignment section in chapter 1.

Function Naming

Naming conventions for the Intel IPP functions are similar for all covered domains. You can distinguish signal processing functions by the `ipps` prefix, while image and video processing functions have `ippi` prefix, and functions that are specific for operations on small matrices have `ippm` prefix in their names.

Function names in Intel IPP have the following general format:

```
ipp<data-domain><name>_<datatype>[_<descriptor>](<parameters>);
```

The elements of this format are explained in the sections that follow.

Data-Domain

The *data-domain* is a single character that expresses the subset of functionality to which a given function belongs. The current version of Intel IPP supports the following data-domains:

- `s` for signals (expected data type is a 1D signal);
- `i` for images and video (expected data type is a 2D image);
- `m` for matrices (expected data type is a matrix).
- `r` for realistic rendering functionality and 3D data processing (expected data type depends on supported rendering techniques).

For example, function names that begin with `ipps` signify that respective functions are used for signal processing.

Name

The *name* is an abbreviation for the core operation that the function really does, for example `Set`, `Copy`, followed in some cases by a function-specific modifier:

```
<name> = <operation>[_modifier]
```

This modifier, if present, denotes a slight modification or variation of the given function. For example, the modifier `CToC` in the function `ippsFFTInv_CToC_32fc` signifies that the inverse fast Fourier transform operates on complex data, performing Complex-To-Complex (`CToC`) transform.

Data Types

The *datatype* field indicates data types used by the function, in the following format:

```
<bit depth><bit interpretation> ,
```

where

```
bit depth = <1|8|16|32|64>
```

and

```
bit interpretation = <u|s|f>[c]
```

Here *u* indicates “unsigned integer”, *s* indicates “signed integer”, *f* indicates “floating point”, and *c* indicates “complex”.

The current version of Intel IPP supports the data types of the source and destination for signal processing functions listed in Table 2-1.

NOTE. In the lists of function parameters, the `Ipp` prefix is written in the data type. For example, the 8-bit signed data is denoted as `Ipp8s` type. These Intel IPP -specific data types are defined in the respective library header files.

Table 2-1 Data Types Supported by Intel IPP for Signal Processing

Type	Usual C Type	Intel IPP Type
8u	unsigned char	Ipp8u
8s	signed char	Ipp8s
16u	unsigned short	Ipp16u
16s	signed short	Ipp16s
16sc	complex short	Ipp16sc
32u	unsigned int	Ipp32u
32s	signed int	Ipp32s
32f	float	Ipp32f
32fc	complex float	Ipp32fc
64s	__int64 (Windows) or long long (Linux)	Ipp64s
64f	double	Ipp64f
64fc	complex double	Ipp64fc

For functions that operate on a single data type, the *datatype* field contains only one of the values listed above.

If a function operates on source and destination signals that have different data types, the respective data type identifiers are listed in the function name in order of source and destination as follows:

<datatype> = <src1Depth>[src2Depth][dstDepth]

For example, the function ippsDotProd_16s16sc_Sfs computes the dot product of 16-bit short and 16-bit complex short source vectors and stores the result in a 16-bit complex short destination vector. The *dstDepth* modifier is not present in the name because the second operand and the result are of the same type. The result is scaled and saturated.

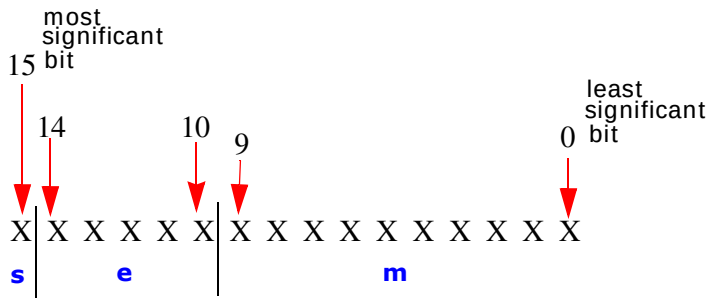
There are several data types, namely 24u, 24s and 16f, that are not supported by the Intel IPP but can be readily converted to supported data types for further processing by the library functions.

For the unsigned `24u` data, each vector element consists of three consecutive bytes represented as `Ipp8u` data types. It has a little-endian byte order when a lower order byte is at the lower address. These data may be converted to and from `32u` or `32f` data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

For the signed `24s` data, each vector element consists of three consecutive bytes represented as `Ipp8u` data types. It has a little-endian byte order when a lower order byte is at the lower address. The sign is represented by the most significant bit of the highest order byte. These data may be converted to and from `32s` or `32f` data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

For the `16f` format, 16-bit floating point data (half type) can represent positive and negative numbers, whose magnitude is between roughly $6.1e^{-5}$ and $6.5e^{4}$, with a relative error of $9.8e^{-4}$; numbers smaller than $6.1e^{-5}$ can be represented with an absolute error of $6.0e^{-8}$. All integers from `-2048` to `+2048` can be represented exactly.

The figure below illustrates the bit-layout for a half number:



s is the sign-bit, **e** is the exponent, and **m** is the significand.

These data may be converted to and from `16s` and `32f` data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

Descriptor

The *`descriptor`* field further describes the data associated with the operation. It may contain implied parameters and/or indicate additional required parameters. To minimize the number of code branches in the function and thus reduce potentially unnecessary

execution overhead, most of the general functions were split into separate primitive functions, with some of their parameters entering the primitive function name as descriptors.

However, some functions may still have parameters that determine internal operations (for example, `ippsThreshold`).

The following descriptors are used in signal processing functions:

`I`	Operation is in-place (default is not-in-place).
`Sfs`	Saturation and fixed scaling mode (default is saturation and no scaling).
`Dx`	Signal is x-dimensional (default is `D1`).
`L`	One pointer is used for each row (in `D2` case).

The abbreviations of descriptors in function names are always presented in alphabetical order.

Not all functions have every abbreviation listed above. For example, in-place mode makes no sense for a copy operation.

Parameters

The *parameters* field specifies the function parameters (arguments).

The order of parameters is as follows:

1. All source operands. Constants follow vectors.
2. All destination operands. Constants follow vectors.
3. Other, operation-specific parameters.

The parameter name have the following conventions:

- All parameters defined as pointers start with *p*, for example, *pPhase*, *pSrc*, *pSeed*; and parameters defined as values start with a lowercase letter, for example, *val*, *src*, *srcLen*.
- Each new part of a parameter name starts with an uppercase character, without underscore; for example, *pSrc*, *lenSrc*, *pDlyLine*.
- Each parameter name specifies its functionality. Source parameters are named *pSrc* or *src*, sometimes followed by names or numbers, e.g., *pSrc2*, *srcLen*. Output parameters are named *pDst* or *dst* followed by names or numbers, e.g., *pDst*, *dstLen*. For in-place operations, the input/ output parameter contains the name *pSrcDst*.

Structures and Enumerators

This section describes the structures and enumerators used by the Intel IPP for signal processing.

Library Version Structure

The `IppLibraryVersion` structure describes the current Intel IPP software version. The main fields of this structure are:

- integer fields *major* and *minor*, which store version numbers;
- string field *Name* storing the Intel IPP version name, e.g., “ippsa6”;
- string field *Version*, storing the version description, e.g., “v0.0 Alpha 0.0.3.3”.

Complex Data Structures

Complex numbers in Intel IPP are described by the structures, containing two numbers of the respective data type which are real and imaginary parts of the complex number.
For example, the single precision complex number is described by the `Ipp32fc` structure as follows:

```
typedef struct {
    Ipp32f re;
    Ipp32f im;
} Ipp32fc;
```

The following complex data types are defined: `Ipp16sc`, `Ipp32fc`, `Ipp64fc`.

Function Context Structures

Some Intel IPP functions that perform such operations as filtering, Fourier and wavelet transforms, use context structures to store function-specific information.

For example, the `IppsFFTSpec` structure stores twiddle factors and bit reverse indexes needed in the fast Fourier transform.

Two different kinds of structures are used: structures with `Spec` suffix in the name, which are not modified during function operation, and structures with `State` suffix in the name, which are modified during operation.

These context-related structures are not defined in public headers, and you can not access the structure fields. It was done because the function context interpretation is processor dependent. Thus, you may only use context-related functions and may not create a function context as an automatic variable.

Enumerators

The `IppStatus` constant enumerates the status values returned by the Intel IPP functions, indicating whether the operation was error-free or not. See section Error Reporting in this chapter for more information on the set of valid status values and corresponding error messages for signal processing functions.

The `IppCmpOp` enumeration defines the type of relational operator to be used by threshold functions:

```
typedef enum {
    ippCmpLess,
    ippCmpLessEq,
    ippCmpEq,
    ippCmpGreaterEq,
    ippCmpGreater
} IppCmpOp;
```

The `IppRoundMode` enumeration defines the rounding mode to be used by conversion functions:

```
typedef enum {
    ippRndZero,
    ippRndNear,
    ippRndFinancial
} IppRoundMode;
```

The `IppHintAlgorithm` enumeration defines the type of code to be used in some transform operations, i. e. faster but less accurate, or vice-versa, more accurate but slower. For more information on using this enumerator, see "Flag and Hint Arguments".

```
typedef enum {
    ippAlgHintNone,
    ippAlgHintFast,
    ippAlgHintAccurate
} IppHintAlgorithm;
```

The `IppCpuType` enumerates processor types returned by the `ippGetCpuType` function:

```
typedef enum {
 /* Enumeration:           Processor:                                */
    ippCpuUnknown = 0x0,
    ippCpuPP,            /* Intel(R) Pentium(R) processor             */
    ippCpuPMX,           /* Pentium(R) processor
                            with MMX(TM) technology                   */
    ippCpuPPR,           /* Pentium(R) Pro processor                  */
    ippCpuPII,           /* Pentium(R) II  processor                  */
    ippCpuPIII,          /* Pentium(R) III processor
                            and Pentium(R)III Xeon(R) processor       */
    ippCpuP4,            /* Pentium(R) 4 processor
                            and Intel(R) Xeon(R) processor            */
    ippCpuP4HT,          /* Pentium(R) 4 processor with HT Technology */
    ippCpuP4HT2,         /* Pentium(R) 4 processor with
                                      Streaming SIMD Extensions 3     */
    ippCpuCentrino,      /* Intel(R) Centrino(TM) mobile technology   */
    ippCpuCoreSolo,      /* Intel(R) Core(TM)Solo processor           */
    ippCpuCoreDuo,       /* Intel(R) Core(TM) Duo processor           */
    ippCpuITP    = 0x10, /* Intel(R) Itanium(R) processor             */
    ippCpuITP2           /* Intel(R) Itanium(R) 2 processor           */
    ippCpuEM64T = 0x20,  /* Intel(R) 64 Instruction Set
                                         Architecture(ISA)            */
    ippCpuC2D,           /* Intel(R) Core(TM) 2 Duo processor         */
    ippCpuC2Q,           /* Intel(R) Core(TM) 2 Quad processor        */
    ippCpuNext
    ippCpuSSE    = 0x40, /* Processor supports Pentium(R) III
                            processor instruction set                 */
    ippCpuSSE2           /* Processor supports Streaming SIMD
                            Extensions 2 instruction set              */
    ippCpuSSE3           /* Processor supports Streaming SIMD
                             Extensions 3 instruction set             */
    ippCpuSSSE3          /* Processor supports Supplemental Streaming
                            SIMD Extensions 3 instruction set         */
    ippCpuX866           /* Processor supports 64 bit extension       */
} IppCpuType;
```

The `IppWinType` enumeration defines the type of window to be used by the FIR Filter Coefficient Generating Functions:

```
typedef enum {
    ippWinBartlett,
    ippWinBlackman,
    ippWinHamming,
    ippWinHann,
    ippWinRect
} IppWinType;
```

The `IppLZ77ComprLevel` enumeration defines the compression level to be used by the ZLIB data compression functions:

```
typedef enum {
    IppLZ77FastCompr,
    IppLZ77AverageCompr,
    IppLZ77BestCompr
} IppLZ77ComprLevel;
```

The `IppLZ77Chcksm` enumeration defines what algorithm is used to compute the checksum by the ZLIB data compression functions:

```
typedef enum {
    IppLZ77NoChcksm,
    IppLZ77Adler32,
    IppLZ77CRC32
} IppLZ77Chcksm;
```

The `IppLZ77Flush` enumeration defines what encoding mode is used by the ZLIB data compression functions:

```
typedef enum {
    IppLZ77NoFlush,
    IppLZ77SyncFlush,
    IppLZ77FullFlush,
    IppLZ77FinishFlush
} IppLZ77Flush;
```

The `IppLZ77DeflateStatus` enumeration defines the encoding status that is used by the ZLIB data compression functions:

```
typedef enum {
    IppLZ77StatusInit,
    IppLZ77StatusLZ77Process,
    IppLZ77StatusHuffProcess,
    IppLZ77StatusFinal
} IppLZ77DeflateStatus;
```

The `IppLZ77InflateStatus` enumeration defines the decoding status that is used by the ZLIB data compression functions:

```
typedef enum {
    IppLZ77InflateStatusInit,
     IppLZ77InflateStatusHuffProcess,
     IppLZ77InflateStatusLZ77Process,
     IppLZ77InflateStatusFinal
```

```
} IppLZ77InflateStatus;
```

The `IppLZ77HuffMode` enumeration defines the encoding mode that is used by the ZLIB data compression functions:

```
typedef enum {
    IppLZ77UseFixed,
    IppLZ77UseDynamic,
    IppLZ77UsedStored,
} IppLZ77HuffMode;
```

The `IppGITStrategyHint` enumeration defines which strategy of encoding is used in some operations by the GIT data compression functions:

```
typedef enum {
    ippGITNoStrategy,
    ippGITLeftReorder,
    ippGITRightReorder,
    ippGITFixedOrder
} IppGITStrategyHint;
```

Data Ranges

The range of values that can be represented by each data type lies between the lower and upper bounds. The following table lists data ranges and constant identifiers used in Intel IPP to denote the respective range bounds:

Table 2-2 Data Types and Ranges

Data Type	Lower Bound		Upper Bound	
	Identifier	Value	Identifier	Value
8s	IPP_MIN_8S	-128	IPP_MAX_8S	127
8u		0	IPP_MAX_8U	255
16s	IPP_MIN_16S	-32768	IPP_MAX_16S	32767
16u		0	IPP_MAX_16U	65535
32s	IPP_MIN_32S	-2^{31}	IPP_MAX_32S	$2^{31}-1$
32u		0	IPP_MAX_32U	$2^{32}-1$
32f †	IPP_MINABS_32F	$1.175494351e^{-38}$	IPP_MAXABS_32F	$3.402823466e^{38}$
64s	IPP_MIN_64S	-2^{63}	IPP_MAX_64S	$2^{63}-1$
64f †	IPP_MINABS_64F	$2.2250738585072014e^{-308}$	IPP_MAXABS_64F	$1.7976931348623158e^{308}$

† The range for absolute values

Data Alignment

The Intel IPP is built using the compiler option `/Zp16`, which aligns the structure fields on the field size or 16 bytes if the size is greater than 16.

The Intel IPP allows also to align the allocated memory pointer on 32 bytes by means of using the function `ippsMalloc`.

Integer Scaling

Some signal processing functions operating on integer data use scaling of the internally computed output results by the integer *scaleFactor*, which is specified as one of the function parameters. These functions have the `Sfs` descriptor in their names.

The scale factor can be negative, positive, or zero. Scaling is applied because internal computations are generally performed with a higher precision than the data types used for input and output signals.

NOTE. The result of integer operations is always saturated to the destination data type range even when scaling is used.

Scaling of an integer result is done by multiplying the output vector values by $2^{-scaleFactor}$ before the function returns. This helps retain either the output data range or its precision. Usually the scaling with a positive factor is performed by the shift operation. The result is rounded off to the nearest integer number.

For example, the integer `Ipp16s` result of the square operation `ippsSqr` for the input value 200 is equal to 32767 instead of 40000, that is, the result is saturated and the exact value can not be restored.

The scaling of the output value with the factor *scaleFactor* = 1 yields the result 20000 which is not saturated, and the exact value can be restored as 20000*2. Thus, the output data range is retained.

The following example shows how the precision can be partially retained by means of scaling.

The integer square root operation `ippsSqrt` (without scaling) for the input value 2 gives the result equal to 1 instead of 1.414. Scaling of the internally computed output value with the factor *scaleFactor* = -3 gives the result 11, and permits to restore the more precise value as $11*2^{-3} = 1.375$.

Error Reporting

The Intel IPP functions return the status of the performed operation to report errors and warnings to the calling program. Thus, it is up to the application to perform error-related actions and/or recover from the error. The last value of the error status is not stored, and the user is to decide whether to check it or not as the function returns. The status values are of `IppStatus` type and are global constant integers.

Table 2-3 lists status codes and corresponding messages reported by the Intel IPP for signal processing.

Table 2-3 Error Status Values and Messages

Status	Message
`ippStsCpuNotSupportedErr`	The target cpu is not supported
`ippStsRoundModeNotSupportedErr`	Unsupported round mode
`ippStsMaxLenHuffCodeErr`	Huff: Max length of Huffman code is more than expected one
`ippStsCodeLenTableErr`	Huff: Invalid codeLenTable
`ippStsFreqTableErr`	Huff: Invalid freqTable
`ippStsRegExpOptionsErr`	RegExp: Options for pattern are incorrect
`ippStsRegExpErr`	RegExp: The structure pRegExpState contains wrong data
`ippStsRegExpMatchLimitErr`	RegExp: The match limit has been exhausted
`ippStsRegExpQuantifierErr`	RegExp: wrong quantifier
`ippStsRegExpGroupingErr`	RegExp: wrong grouping
`ippStsRegExpBackRefErr`	RegExp: wrong back reference
`ippStsRegExpChClassErr`	RegExp: wrong character class
`ippStsRegExpMetaChErr`	RegExp: wrong metacharacter
`ippStsMP3FrameHeaderErr`	Error in fields IppMP3FrameHeader structure
`ippStsMP3SideInfoErr`	Error in fields IppMP3SideInfo structure
`ippStsAacPrgNumErr`	AAC: Invalid number of elements for one program
`ippStsAacSectCbErr`	AAC: Invalid section codebook

Table 2-3 Error Status Values and Messages (continued)

ippStsAacSfValErr	AAC: Invalid scalefactor value
ippStsAacCoefValErr	AAC: Invalid quantized coefficient value
ippStsAacMaxSfbErr	AAC: Invalid coefficient index
ippStsAacPredSfbErr	AAC: Invalid predicted coefficient index
ippStsAacPlsDataErr	AAC: Invalid pulse data attributes
ippStsAacGainCtrErr	AAC: Gain control not supported
ippStsAacSectErr	AAC: Invalid number of sections
ippStsAacTnsNumFiltErr	AAC: Invalid number of TNS filters
ippStsAacTnsLenErr	AAC: Invalid TNS region length
ippStsAacTnsOrderErr	AAC: Invalid order of TNS filter
ippStsAacTnsCoefResErr	AAC: Invalid bit-resolution for TNS filter coefficients
ippStsAacTnsCoefErr	AAC: Invalid TNS filter coefficients
ippStsAacTnsDirectErr	AAC: Invalid TNS filter direction
ippStsAacTnsProfileErr	AAC: Invalid TNS profile
ippStsAacErr	AAC: Internal error
ippStsAacBitOffsetErr	AAC: Invalid current bit offset in bitstream
ippStsAacAdtsSyncWordErr	AAC: Invalid ADTS syncword
ippStsAacSmplRateIdxErr	AAC: Invalid sample rate index
ippStsAacWinLenErr	AAC: Invalid window length (not short or long)
ippStsAacWinGrpErr	AAC: Invalid number of groups for current window length
ippStsAacWinSeqErr	AAC: Invalid window sequence range
ippStsAacComWinErr	AAC: Invalid common window flag
ippStsAacStereoMaskErr	AAC: Invalid stereo mask
ippStsAacChanErr	AAC: Invalid channel number
ippStsAacMonoStereoErr	AAC: Invalid mono-stereo flag
ippStsAacStereoLayerErr	AAC: Invalid this Stereo Layer flag
ippStsAacMonoLayerErr	AAC: Invalid this Mono Layer flag
ippStsAacScalableErr	AAC: Invalid scalable object flag
ippStsAacObjTypeErr	AAC: Invalid audio object type
ippStsAacWinShapeErr	AAC: Invalid window shape
ippStsAacPcmModeErr	AAC: Invalid PCM output interleaving indicator
ippStsVLCUsrTblHeaderErr	VLC: Invalid header inside table

Table 2-3 Error Status Values and Messages (continued)

ippStsVLCUsrTblUnsupportedFmtErr	VLC: Unsupported table format
ippStsVLCUsrTblEscAlgTypeErr	VLC: Unsupported Ecs-algorithm
ippStsVLCUsrTblEscCodeLengthErr	VLC: Incorrect Esc-code length inside table header
ippStsVLCUsrTblCodeLengthErr	VLC: Unsupported code length inside table
ippStsVLCInternalTblErr	VLC: Invalid internal table
ippStsVLCInputDataErr	VLC: Invalid input data
ippStsVLCAACEscCodeLengthErr	VLC: Invalid AAC-Esc code length
ippStsIncorrectLSPErr	Incorrect Linear Spectral Pair values
ippStsNoRootFoundErr	No roots are found for equation
ippStsLengthErr	Wrong value of string length
ippStsFBankFreqErr	Incorrect value of the filter bank frequency parameter
ippStsFBankFlagErr	Incorrect value of the filter bank parameter
ippStsFBankErr	Filter bank is not correctly initialized
ippStsNegOccErr	Negative occupation count
ippStsCdbkFlagErr	Incorrect value of the codebook flag parameter
ippStsSVDCnvgErr	No convergence of SVD algorithm
ippStsToneMagnErr	Tone magnitude is less than or equal to zero
ippStsToneFreqErr	Tone frequency is negative, or greater than or equal to 0.5
ippStsTonePhaseErr	Tone phase is negative, or greater than or equal to 2*PI
ippStsTrnglMagnErr	Triangle magnitude is less than or equal to zero
ippStsTrnglFreqErr	Triangle frequency is negative, or greater than or equal to 0.5
ippStsTrnglPhaseErr	Triangle phase is negative, or greater than or equal to 2*PI
ippStsTrnglAsymErr	Triangle asymmetry is less than -PI, or greater than or equal to PI
ippStsHugeWinErr	Incorrect size of the Kaiser window
ippStsJaehneErr	Magnitude value is negative
ippStsStepErr	Step value is not valid
ippStsDlyLineIndexErr	Invalid value of the delay line sample index
ippStsStrideErr	Stride value is less than the row length
ippStsEpsValErr	Negative epsilon value error
ippStsScaleRangeErr	Scale bounds are out of range

Table 2-3 Error Status Values and Messages (continued)

ippStsThresholdErr	Invalid threshold bounds
ippStsWtOffsetErr	Invalid offset value of the wavelet filter
ippStsAnchorErr	Anchor point is outside the mask
ippStsMaskSizeErr	Invalid mask size
ippStsShiftErr	Shift value is less than zero
ippStsSampleFactorErr	Sampling factor is less than or equal to zero
ippStsSamplePhaseErr	Phase value is out of range, $0 \le phase < factor$
ippStsFIRMRFactorErr	MR FIR sampling factor is less than or equal to zero
ippStsFIRMRPhaseErr	MR FIR sampling phase parameter is negative, or greater than or equal to the sampling factor
ippStsRelFreqErr	Relative frequency value is out of range
ippStsFIRLenErr	Length of a FIR filter is less than or equal to zero
ippStsIIROrderErr	Order of an IIR filter is less than or equal to zero
ippStsResizeFactorErr	Resize factor(s) is less than or equal to zero
ippStsDivByZeroErr	An attempt to divide by zero
ippStsInterpolationErr	Invalid interpolation mode
ippStsMirrorFlipErr	Invalid flip mode
ippStsMoment00ZeroErr	Moment value M(0,0) is too small to continue calculations
ippStsThreshNegLevelErr	Negative value of the level in the threshold operation
ippStsContextMatchErr	Context parameter doesn't match the operation
ippStsFftFlagErr	Invalid value of the FFT flag parameter
ippStsFftOrderErr	Invalid value of the FFT order parameter
ippStsMemAllocErr	Not enough memory allocated for the operation
ippStsNullPtrErr	Null pointer error
ippStsSizeErr	Wrong value of the data size
ippStsBadArgErr	Function argument/parameter is bad
ippStsErr	Unknown/unspecified error
ippStsNoErr	No error, it's OK
ippStsNoOperation	No operation has been executed
ippStsMisalignedBuf	Misaligned pointer in operation in which it must be aligned
ippStsSqrtNegArg	Negative value(s) of the argument in the function Sqrt
ippStsInvByZero	Inf result. Zero value was met by InvThresh with zero level

Table 2-3 Error Status Values and Messages (continued)

ippStsEvenMedianMaskSize	Even size of the Median Filter mask was replaced by the odd one
ippStsDivByZero	Zero value(s) of the divisor in the function Div
ippStsLnZeroArg	Zero value(s) of the argument in the function Ln
ippStsLnNegArg	Negative value(s) of the argument in the function Ln
ippStsNanArg	Not a Number (NaN) argument value warning
ippStsResFloor	All result values are floored
ippStsOverflow	Overflow occurred in the operation
ippStsZeroOcc	Zero occupation count
ippStsUnderflow	Underflow occurred in the operation
ippStsSingularity	Singularity occurred in the operation
ippStsDomain	Argument is out of the function domain
ippStsNotIntelCpu	The target cpu is not Genuine Intel
ippStsCpuMismatch	The library for given cpu cannot be set
ippStsNotIppFunctionFound	Application does not contain IPP functions calls
ippStsDllNotFoundBestUsed	The newest version of IPP DLL's not found by dispatcher
ippStsNoOperationInDll	The function does nothing in the dynamic version of the library
ippStsOvermuchStrings	Number of destination strings is more than expected
ippStsOverlongString	Length of one of the destination strings is more than expected
ippStsSrcSizeLessExpected	DC: The size of source buffer is less than expected one
ippStsDstSizeLessExpected	DC: The size of destination buffer is less than expected one
ippStsNotSupportedCpu	The CPU is not supported
ippStsUnkhownCacheSize	The CPU is supported, but the size of the cache is unknown.

The status codes ending with `Err` (except for the `ippStsNoErr` status) indicate an error; the integer values of these codes are negative. When an error occurs, the function execution is interrupted.
All other status codes indicate warnings. When a specific case is encountered, the function execution is completed and the corresponding warning status is returned.

For example, if the integer function `ippsDiv_8u` meets an attempt to divide a positive value by zero, the function execution is not aborted. The result of the operation is set to the maximum value that can be represented by the source data type, and the function

returns the warning status `ippStsDivByZero`. This is the case for the vector-vector operation `ippsDiv`. For the vector-scalar division operation `ippsDivC`, the function behavior is different: if the constant divisor is zero, then the function stops execution and returns immediately with the error status `ippStsDivByZeroErr`.

Code Examples

The manual contains a number of code examples that use Intel IPP functions and serve to demonstrate both some particular features of the primitives and how the primitives can be called. Many of these code examples output result data together with status code and associated messages in case when error or warning condition was met.

To keep the example code simpler, special definitions of print statements are used that get output strings look exactly the way is needed for better representation of both real and complex results of different format, as well as print status codes and messages.

The code definitions given below make it possible to build the examples contained in the manual by straightforward copying and pasting the example code fragments.

```
/// the functions providing simple output of the result
/// they are for real and complex data

#define genPRINT(TYPE,FMT) \
void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus st ) { \
   int n; \
   if( st > ippStsNoErr ) \
      printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \
   else if( st < ippStsNoErr ) \
      printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
   printf(" %s ", msg ); \
   for( n=0; n<len; ++n ) printf( FMT, buf[n] ); \
   printf("\n" ); \
}

#define genPRINTcplx(TYPE,FMT) \
void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus st ) { \
   int n; \
   if( st > ippStsNoErr ) \
```

```
      printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \
   else if( st < ippStsNoErr ) \
      printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
   printf(" %s ", msg ); \
   for( n=0; n<len; ++n ) printf( FMT, buf[n].re, buf[n].im ); \
   printf("\n" ); \
}
genPRINT( 64f, " %f" )
genPRINT( 32f, " %f" )
genPRINT( 16s, " %d" )

genPRINTcplx( 64fc, " {%f,%f}" )
genPRINTcplx( 32fc, " {%f,%f}" )
genPRINTcplx( 16sc, " {%d,%d}" )
```

Support Functions

3

This chapter describes Intel® IPP support functions that are used to:

- retrieve information about the current Intel IPP software version
- allocate and free memory that is needed for the operation of other Intel IPP functions.
- perform specific auxiliary operations

The full list of these functions is given in Table 3-1.

Table 3-1 Intel IPP Support Functions

Function Base Name	Operation
Version Information Functions	
ippsGetLibVersion	Returns information about the active library version.
Memory Allocation Functions	
ippsMalloc	Allocates memory aligned to 32-byte boundary.
ippsFree	Frees memory allocated by the function ippsMalloc.
Common Functions	
ippGetStatusString	Translates a status code into a message.
ippGetCpuType	Returns a processor type.
ippGetCpuClocks	Returns a current value of the time stamp counter (TSC) register.
ippGetCpuFreqMhz	Estimates the processor frequency.
ippGetMaxCacheSizeB	
ippSetFlushToZero	Enables or disables flush-to-zero mode.
ippSetDenormAreZeros	Enables or disables denormals-are-zero mode.
ippAlignPtr	Aligns a pointer to the specified number of bytes.

Table 3-1 Intel IPP Support Functions (continued)

Function Base Name	Operation
ippSetNumThreads	Sets the number of threads in the multithread environment.
ippGetNumThreads	Returns the number of existing threadsin the multithread environment.
ippMalloc	Allocates memory aligned to 32-byte boundary.
ippFree	Frees memory allocated by the function ippMalloc.
Dispatcher Control Functions	
ippStaticInit	Automatically selects the most appropriate static code.
ippStaticInitCpu	Initializes the specified version of the static code.

Version Information Functions

These functions return the version number and other information about the active Intel IPP software.

ippsGetLibVersion

Returns information about the active version of Intel IPP signal processing software.

Syntax

```
const IppLibraryVersion* ippsGetLibVersion(void);
```

Description

The function ippsGetLibVersion is declared in the ipps.h file. This function returns a pointer to a static data structure IppLibraryVersion that contains information about the current version of the Intel IPP software for signal processing. There is no need for you to release memory referenced by the returned pointer, as it points to a static variable. The following fields of the IppLibraryVersion structure are available:

major	is the major number of the current library version.
minor	the minor number of the current library version.
majorBuild	the number of builds of the major version.

build	current build number.
Name	the name of the current library version.
Version	is the library version string.
BuildDate	is the library version actual build date.

For example, if the library version is “v1.2 Beta”, library name is “ippsm6”, and build date is “Jul 20 99”, then the fields in this structure are set as:

major = 1, *minor* = 2, *Name* = “ippsm6”,
Version = “v1.2 Beta”, *BuildDate* = “Jul 20 99”

Example 3-1 shows how to use the function `ippsGetLibVersion`.

Example 3-1 Using the Function `ippsGetLibVersion`

```
void libinfo(void) {
       const IppLibraryVersion* lib = ippsGetLibVersion();
       printf("%s %s %d.%d.%d.%d\n", lib->Name, lib->Version,
           lib->major, lib->minor, lib->majorBuild, lib->build);
}

Output:
    ippsa6 v0.0 Alpha 0.0.5.5
```

NOTE. Each sub-library that is used in the signal processing domain has its own similar function to retrieve information about the active library version. These functions are: `ippGetLibVersion`, `ippacGetLibVersion`, `ippchGetLibVersion`, `ippdcGetLibVersion`, `ippscGetLibVersion`, `ippsrGetLibVersion`, `ippvmGetLibVersion`. They are declared in the following header files: `ippcore.h`, `ippac.h`, `ippch.h`, `ippdc.h`, `ippsc.h`, `ippsr.h`, `ippvm.h`, respectively, and have the same interface as the above described function.

Memory Allocation Functions

This section describes the Intel IPP signal processing functions that allocate aligned memory blocks for data of required type or free the previously allocated memory. The size of allocated memory is specified by the number of elements allocated, *len*.

NOTE. The only function to free the memory allocated by any of these functions is `ippsFree()`.

ippsMalloc

Allocates memory aligned to 32-byte boundary.

Syntax

```
Ipp8u* ippsMalloc_8u(int len);
Ipp16u* ippsMalloc_16u(int len);
Ipp32u* ippsMalloc_32u(int len);
Ipp8s* ippsMalloc_8s(int len);
Ipp16s* ippsMalloc_16s(int len);
Ipp32s* ippsMalloc_32s(int len);
Ipp64s* ippsMalloc_64s(int len);
Ipp32f* ippsMalloc_32f(int len);
Ipp64f* ippsMalloc_64f(int len);
Ipp8sc* ippsMalloc_8sc(int len);
Ipp16sc* ippsMalloc_16sc(int len);
Ipp32sc* ippsMalloc_32sc(int len);
Ipp64sc* ippsMalloc_64sc(int len);
Ipp32fc* ippsMalloc_32fc(int len);
Ipp64fc* ippsMalloc_64fc(int len);
```

Parameters

len	Number of elements to allocate.

Description

The function `ippsMalloc` is declared in the `ipps.h` file. This function allocates memory block aligned to a 32-byte boundary for elements of different data types.

Example 3-2 shows how to use the function `ippsMalloc_8u`.

Example 3-2 Using the Function `ippsMalloc`

```
void func_malloc(void)
{
    Ipp8u* pBuf = ippsMalloc_8u(8*sizeof(Ipp8u));
    if(NULL == pBuf)
      // not enough memory

    ippsFree(pBuf);
}
```

Return Values

The return value of `ippsMalloc` is a pointer to an aligned memory block. If no memory is available in the system, then the `NULL` value is returned. To free this block, use the function `ippsFree`.

ippsFree

Frees memory allocated by the function `ippsMalloc`.

Syntax

```
void ippsFree(void* ptr);
```

Parameters

ptr	Pointer to a memory block to be freed. The memory block pointed to with *ptr* has been allocated by the function `ippsMalloc`.

Description

The function `ippsFree` is declared in the `ipps.h` file. This function frees the aligned memory block allocated by the function `ippsMalloc`.

CAUTION. You can not use the function `ippsFree` to free memory allocated by standard functions like `malloc` or `calloc`, nor can the memory allocated by `ippsMalloc` be freed by `free`.

Common Functions

This section describes the Intel IPP functions that perform special operations common for all domains. For example, the function `ippGetStatusString` gets a brief description of the status code returned by the current Intel IPP software. The function `ippGetCpuType` retrieves the type of the processor in your system. The function `ippGetCpuClocks` gets the current number of processor clocks, which is widely used for operations timing. The functions `ippSetFlushToZero` and `ippSetDenormAreZeros` enable or disable special processor modes. The function `ippAlignPtr` performs pointer alignment. The functions `ippMalloc` and `ippFree` allow to allocate and free the memory block aligned to 32-byte boundary. The specific subset contains functions to control the dispatching of the merged static libraries.
All these functions are grouped in the separate sub-library called `ippcore`.

ippGetStatusString

Translates a status code into a message.

Syntax

```
const char* ippGetStatusString(IppStatus StsCode);
```

Parameters

StsCode	Code that indicates the status type (see Table 2-3).

Description

The function `ippGetStatusString` is declared in the `ippcore.h` file. This function returns a pointer to the text string associated with a status code of type `IppStatus`. Use this function to produce error and warning messages for users. The returned pointer is a pointer to an internal static buffer and need not be released.

A code Example 3-3 shows how to use the function `ippGetStatusString`.
If you call an Intel IPP function, in this case, `ippsAddC_16s_I` with a `NULL` pointer, it returns an error code `-8`. The status information function translates this code into the corresponding message "`Null Pointer Error`".

Example 3-3 Using the Function `ippGetStatusString`

```
void statusinfo(void) {
        IppStatus st = ippsAddC_16s_I (3, 0, 0);
        printf("%d : %s\n", st, ippGetStatusString(st));
}

Output:
    -8, Null Pointer Error
```

ippGetCpuType

Returns a processor type.

Syntax

```
IppCpuType ippGetCpuType (void);
```

Description

The function `ippGetCpuType` is declared in the `ippcore.h` file. This function detects the processor type used in your computer system and returns an appropriate `IppCpuType` variable value. Table 3-2 lists possible values and their meaning.

Table 3-2 Processor Types Detected in Intel IPP

Returned Variable Value	Processor Type
ippCpuPP	Intel® Pentium® processor
ippCpuPMX	Intel® Pentium® processor with MMX™ technology
ippCpuPPR	Intel® Pentium® Pro processor

Table 3-2 Processor Types Detected in Intel IPP (continued)

Returned Variable Value	Processor Type
ippCpuPII	Intel® Pentium® II processor
ippCpuPIII	Intel® Pentium® III processor or Intel® Pentium® III Xeon® processor
ippCpuP4	Intel® Pentium® 4 processor or Intel® Xeon® processor
ippCpuP4HT	Intel Pentium 4 processor with Hyper-Threading Technology
ippCpuP4HT2	Intel Pentium 4 Processor with Streaming SIMD Extensions 3
ippCpuCentrino	Intel® Centrino™ mobile technology
ippCpuCoreSolo	Intel® Core™ Solo processor
ippCpuCoreDuo	Intel® Core™ Duo processor
ippCpuITP	Intel® Itanium® processor
ippCpuITP2	Intel® Itanium® 2 processor
ippCpuEM64T	Intel® 64 instruction set architecture
ippCpuC2D	Intel® Core™ 2 Duo processor
ippCpuC2Q	Intel® Core™ 2 Quad processor
ippCpuSSE	Processor supports Intel Pentium III processor instruction set
ippCpuSSE2	Processor with Streaming SIMD Extensions 2 instruction set
ippCpuSSE3	Processor with Streaming SIMD Extensions 3 instruction set
ippCpuSSSE3	Processor with Supplemental Streaming SIMD Extensions 3 instruction set
ippCpuX8664	Processor supports 64 bit extension
ippCpuUnknown	Unknown processor

ippGetCpuClocks

Returns a current value of the time stamp counter (TSC) register.

Syntax

```
Ipp64u ippGetCpuClocks (void);
```

Description

The function `ippGetCpuClocks` is declared in the `ippcore.h` file. This function reads the current state of the time stamp counter (TSC) register and returns its value. A hardware exception is possible if TSC reading is not supported by the current chipset.

ippGetCpuFreqMhz

Estimates the processor operating frequency.

Syntax

```
IppStatus ippGetCpuFreqMhz(int* pMhz);
```

Parameters

pMhz	Pointer to the output result.

Description

The function `ippGetCpuFreqMhz` is declared in the `ippcore.h` file. This function estimates the processor operating frequency and returns its value in MHz as an integer stored in *pMhz*.

Note that no exact value of frequency is guaranteed so that the estimated value can vary depending on the processor workload.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition when the *pMhz* pointer is `NULL`.

ippGetMaxCacheSizeB

Returns maximum size of the L2 and L3 caches of the processor.

Syntax

```
IppStatus ippGetMaxCacheSizeB(int* pSizeByte)
```

Parameters

pSizeByte	Pointer to the output result.

Description

The function `ippGetMaxCacheSizeB` is declared in the `ippcore.h` file. This function finds the maximum size (in bytes) of the L2 and L3 caches of the processor used in your computer system. THe result is stored it in the *pSizeByte*.

Note the function supports only Intel® IA-32 processors including those with Intel® EM64T. If the processor is not supported, or size of cache is unknown, the result is 0, and the function returns corresponding warning message.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition when the *pSizeByte* pointer is `NULL`.
`ippStsNotSupportedCpu`	Indicates that processor is not supported.
`ippStsUnknownCacheSize`	Indicates that size of the cache is unknown.

ippSetFlushToZero

Enables or disables flush-to-zero mode.

Syntax

```
IppStatus ippSetFlushToZero(int value, unsigned int* pUMask);
```

Parameters

value	Switch to set or clear the corresponding bit of the MXCSR register.
pUMask	Pointer to the current underflow exception mask; may be set to NULL.

Description

The function ippSetFlushToZero is declared in the ippcore.h file. This function enables (when the *value* is not equal to 0) or disables (when the *value* is equal to 0) a flush-to-zero (FTZ) mode of processors that support Streaming SIMD Extensions (SSE) instructions. The FTZ mode controls the masked response to a SIMD floating-point underflow condition. The FTZ mode is provided primarily for performance reasons. At the cost of a slight precision loss, FTZ mode enables faster execution of applications where underflows are common and rounding the underflow result to zero can be tolerated.

FTZ mode is possible only when the mask register is in a certain state. The ippSetFlushToZero function checks and changes this state if necessary. After disabling the FTZ mode, you can restore the initial mask register state. To do this, you must declare a variable of unsigned integer type in your application and point to it in the *pUMask* parameter of the ippSetFlushToZero function. The initial state of mask register will be saved in this location and can be restored later. If you do not need to restore the initial mask state, then the *pUMask* pointer may be set to NULL.

Return Values

ippStsNoErr	Indicates no error.
ippStsCpuNotSupportedErr	Indicates an error condition when the FTZ mode is not supported by the processor.

ippSetDenormAreZeros

Enables or disables denormals-are-zero mode.

Syntax

```
IppStatus ippSetDenormAreZeros(int value);
```

Parameters

value	Switch to set or clear the corresponding bit of the MXCSR register.

Description

The function `ippSetDenormAreZeros` is declared in the `ippcore.h` file. This function enables (when the *value* is not equal to 0) or disables (when the *value* is equal to 0) the denormals-are-zero (DAZ) mode of processors that support Streaming SIMD Extensions (SSE) instructions. The DAZ mode controls the processor response to a SIMD floating-point denormal operand condition. When the DAZ flag is set, the processor converts all denormal source operands to zero with the sign of the original operand before performing any computations on source data. The DAZ mode is provided to improve processor performance for applications such as streaming media processing, where rounding a denormal operand to zero does not noticeably affect the quality of the processed data.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsCpuNotSupportedErr`	Indicates an error condition when the DAZ mode is not supported by the processor.

ippAlignPtr

Aligns a pointer to the specified number of bytes.

Syntax

```
void* ippAlignPtr(void* ptr, int alignBytes);
```

Parameters

ptr	Processor type.
alignBytes	Number of bytes to align. Possible values are the powers of 2, that is, 2, 4, 8, 16 and so on.

Description

The function `ippAlignPtr` is declared in the `ippcore.h` file. This function returns a pointer *`ptr`* aligned to the specified number of bytes *`alignBytes`*. Possible values of *`alignBytes`* are powers of two. The function does not check the validity of this parameter.

CAUTION. Do not free the pointer returned by the function, but free the original pointer.

ippSetNumThreads

Sets the number of threads in the multithreading environment.

Syntax

```
IppStatus ippSetNumThreads(int numThr);
```

Parameters

`numThr`	Number of threads, should be greater than 0.

Description

The function `ippSetNumThreads` is declared in the `ippcore.h` file. This function sets the number of OMP threads. A number of established threads may be less than specified *`numThr`*.

WARNING. This function operates only in the dynamic version of the library. In static version it returns a warning status `ippStsNoOperation`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsSizeErr`	Indicates an error condition if *`numThr`* is less than or equal to 0.
`ippStsNoOperation`	Indicates that there is no such operation in the static version of the library.

ippGetNumThreads

Returns the number of existing threads in the multithreading environment.

Syntax

```
IppStatus ippGetNumThreads(int* pNumThr);
```

Parameters

pNumThr	Pointer to the number of threads.

Description

The function `ippGetNumThreads` is declared in the `ippcore.h` file. This function returns the number of OMP threads specified by the user previously. If it has not been specified, the function returns the initial number of threads that depends on the number of logical processors

WARNING. This function operates only in the dynamic version of the library. In static version it returns a warning status `ippStsNoOperation`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition when the *pMhz* pointer is `NULL`.
`ippStsNoOperation`	Indicates that there is no such operation in the static version of the library.

ippMalloc

Allocates memory aligned to 32-byte boundary.

Syntax

```
void* ippMalloc(int length);
```

Parameters

len	Size (in bytes) of the allocated block.

Description

The function `ippMalloc` is declared in the `ippcore.h` file. This function allocates memory block aligned to a 32-byte boundary.

Return Values

The return value of `ippMalloc` is a pointer to an aligned memory block. To free this block, use only the function `ippFree`.

ippFree

Frees memory allocated by the function `ippMalloc`.

Syntax

```
void ippFree(void* ptr);
```

Parameters

ptr	Pointer to a memory block to be freed.

Description

The function `ippFree` is declared in the `ippcore.h` file. This function frees the aligned memory block allocated by the function `ippMalloc`.

CAUTION. You can not use the function `ippFree` to free memory allocated by any other functions like `malloc`, nor can the memory allocated by `ippMalloc` be freed by `free`.

Dispatcher Control Functions

This section describes Intel IPP functions that control the dispatchers of the merged static libraries.

ippStaticInit

Automatically selects the most appropriate static code.

Syntax

```
IppStatus ippStaticInit(void);
```

Description

The function `ippStaticInit` is declared in the `ippcore.h` file. This function detects the processor type used in the user computer system and sets the most appropriate processor-specific static code of the Intel IPP software.

The function `ippStaticInit` should not be used in the driver implementation.

WARNING. This function operates only in the static version of the library.

CAUTION. You can not use any other Intel IPP function while the function `ippStaticInit` continues execution.

Return Values

`ippStsNoErr`	Indicates that the most appropriate static code of the Intel IPP software is successfully set.
`ippStsNonIntelCpu`	Indicates that the static version of generic code for Intel® Architecture is set.
`ippStsNoOperationInDll`	Indicates that there is no such operation in the dynamic version of the library.

ippStaticInitCpu

Initializes the specified version of the static code.

Syntax

```
IppStatus ippStaticInitCpu(IppCpuType cpu);
```

Parameters

cpu	Processor type.

Description

The function `ippStaticInitCpu` is declared in the `ippcore.h` file. This function sets the processor-specific static code of the Intel IPP library in accordance with the specified processor type *cpu*.

WARNING. This function operates only in the static version of the library.

Return Values

`ippStsNoErr`	Indicates that the required processor-specific static code is successfully set.
`ippStsCpuMismatch`	Indicates that the specified processor type is not valid and the previously set code version is used.
`ippStsNoOperationInDll`	Indicates that there is no such operation in the dynamic version of the library.

Vector Initialization Functions

4

This chapter describes Intel® IPP functions that initialize vectors with either constants, the contents of other vectors, or the generated signals.

The full list of functions in this group is given in Table 4-1.

Table 4-1 Intel IPP Vector Initialization Functions

Function Base Name	Operation
Vector Initialization Functions	
Copy	Copies the contents of one vector into another.
PackBits	Packs part of data bits from the vector to the another vector.
Move	Moves the contents of one vector to another vector.
Set	Initializes vector elements to a specified common value.
Zero	Initializes a vector to zero.
Tone and Triangle Generation Functions	
ToneInitAllocQ15	Allocates memory and initializes the tone generator state
ToneFree	Frees memory allocated by the function `ippsToneInitAlloc`.
ToneGetStateSizeQ15	Computes the length of the tone generator structure.
ToneInitQ15	Initializes the tone generator specification structure.
ToneQ15	Generates a tone in accordance with tone generator specification
Tone_Direct	Generates a tone with a given frequency, phase, and magnitude.

Table 4-1 Intel IPP Vector Initialization Functions (continued)

Function Base Name	Operation
ToneQ15_Direct	Generates a tone with a given frequency, phase, and magnitude.
TriangleInitAllocQ15	Allocates memory and initializes the triangle generator state.
TriangleFree	Frees memory allocated by the function ippsTriangleInitAlloc.
TriangleGetStateSizeQ15	Computes the length of the triangle generator structure.
TriangleInitQ15	Initializes the triangle generator specification structure.
TriangleQ15	Generates a triangle in accordance with triangle generator specification
Triangle_Direct	Generates a triangle with a given frequency, phase, and magnitude.
TriangleQ15_Direct	Generates a triangle with a given frequency, phase, and magnitude
Uniform Distribution Functions	
RandUniformInitAlloc	Allocates memory and initializes a noise generator with uniform distribution.
RandUniformFree	Closes the uniform distribution generator state.
RandUniformGetSize	Computes the length of the uniform distribution generator structure
RandUniformInit	Initializes a noise generator with uniform distribution.
RandUnifrom	Generates the pseudo-random samples with a uniform distribution.
RandUniform_Direct	Generates the pseudo-random samples with a uniform distribution in direct mode.
Gaussian Distribution Functions	
RandGaussInitAlloc	Allocates memory and initializes a noise generator with Gaussian distribution.
RandGaussFree	Closes the Gaussian distribution generator state.
RandGaussGetSize	Computes the length of the Gaussian distribution generator state
RandGaussInit	Initializes a noise generator with Gaussian distribution.
RandGauss	Generates the pseudo-random samples with a Gaussian distribution.

Table 4-1 Intel IPP Vector Initialization Functions (continued)

Function Base Name	Operation
RandGauss_Direct	Generates pseudo-random samples with a Gaussian distribution in the direct mode.
Special Vector Functions	
VectorJaehne	Creates a Jaehne vector.
ippsVectorSlope	Creates a slope vector.
VectorRamp	Creates a ramp vector.

Vector Initialization Functions

This section describes functions that initialize the values of vector elements.
All vector elements can be initialized to a common zero or another specified value. They can also be initialized to respective values of a second vector elements.

Copy

Copies the contents of one vector into another.

Syntax

```
IppStatus ippsCopy_1u(const Ipp8u* pSrc, int srcBitOffset, Ipp8u* pDst,
    int dstBitOffset, int len);
IppStatus ippsCopy_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsCopy_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsCopy_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsCopy_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCopy_64s(const Ipp64s* pSrc, Ipp64s* pDst, int len );
IppStatus ippsCopy_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len );
IppStatus ippsCopy_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsCopy_32sc(const Ipp32sc* pSrc, Ipp32sc* pDst, int len);
IppStatus ippsCopy_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCopy_64sc(const Ipp64sc* pSrc, Ipp64sc* pDst, int len);
```

```
IppStatus ippsCopy_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

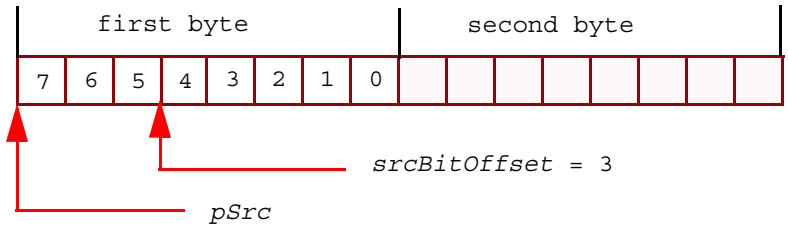
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements to copy.
srcBitOffset	Offset (in bits) in the first byte of the source vector.
dstBitOffset	Offset (in bits) in the first byte of the destination vector.

Description

The function `ippsCopy` is declared in the `ipps.h` file. This function copies the first *len* elements from a source vector *pSrc* into a destination vector *pDst*.

ippsCopy_1u. This funtion flavor performs copying operation on a vector that has a `8u` data type where each byte represents eight consecutive elements. In ths case additional parameters *srcBitOffset* and *dstBitOffset* are required to specifiy start positions in the source and destination vectors respectively. Note that the bit order in each byte is inverse relative to the element order, that is, the first element in a vector is represented by the last (7th) bit in the first byte of the vector (see Figure 4-1).

Figure 4-1 Bit Layout for the Function `ippsCopy_1u`.



CAUTION. These functions perform only copying operations described above and are not intended to move data. Their behavior is unpredictable if source and destination buffers are overlapping.
To move data, the functions `ippsMove` must be used

Example 4-1 shows how to use the function `ippsCopy`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 4-1 Using the `ippsCopy` Function

```
IppStatus copy(void) {
      char src[] = "to be copied\0";
      char dst[256];
      return ippsCopy_8u(src, dst, strlen(src)+1);
}
```

PackBits

Packs part of data bits from the vector to the bitstream.

Syntax

```
IppStatus ippsPackBits_32u8u(const Ipp32u* pSrcBit, const int* pSrcBits,
   int srcLen, Ipp8u* pDst, int dstBitOffset, int* pDstLenBit);
```

Parameters

pSrcBit	Pointer to the source vector with data.
pSrcBits	Pointer to the source vector that specifies the number of data bits.
srcLen	Number of elements in each source vector.
pDst	Pointer to the destination vector (bitstream).
dstBitOffset	Offset (in bits) in the first byte of the destination vector.
pDstLenBit	Pointer to the length in bits of the destination vector.

Description

The function `ippsPackBits` is declared in the `ipps.h` file. This function copies the certain number of bits from each element of the data source vector *pSrcBit* and stores them contiguously in the destination vector *pDst* starting from the bit position *dstBitOffset*. Each element of the source vector *pSrcBits* contains the number of bits that are copied from the corresponding element of the vector *pSrcBit*. If this number is less than 0, or greater than 32, then its value is set to 0 or 32 respectively. If such cases occur, the function returns the warning message after completing the operation.
After completing the operation the function returns the lenght of the destiantion vector in bits *pDstLenBit*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *srcLen* is less than or equal to 0, or *dstBitOffset* is less than 0.
`ippStsOverlongString`	Indicates a warning when *pSrcBits*[i] is greater than 32 or less than 0.

Move

Moves the contents of one vector to another vector.

Syntax

```
IppStatus ippsMove_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsMove_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMove_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsMove_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMove_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len );
IppStatus ippsMove_64s(const Ipp64s* pSrc, Ipp64s* pDst, int len);
IppStatus ippsMove_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsMove_32sc(const Ipp32sc* pSrc, Ipp32sc* pDst, int len);
IppStatus ippsMove_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
```

```
IppStatus ippsMove_64sc(const Ipp64sc* pSrc, Ipp64sc* pDst, int len);
IppStatus ippsMove_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector used to initialize *pDst*.
pDst	Pointer to the destination vector to be initialized.
len	Number of elements to move.

Description

The function `ippsMove` is declared in the `ipps.h` file. This function moves the first *len* elements from a source vector *pSrc* into the destination vector *pDst*. If some parts of the source and destination vectors are overlapping, then the function ensures that the original source bytes in the overlapping parts are moved (it means that they are copied before being overwritten) to the appropriate parts of the destination vector.

Example 4-3 shows how to use the function `ippsMove`.

Example 4-2 Using the `ippsMove` function

```
Ipp8u pSrc[10] = { "123456789" };
Ipp8u pDst[6];
int len = 6;
IppStatus status;

status = ippsMove_8u ( pSrc, pDst, len );
if(ippStsNoErr != status)
  printf("IPP Error: %s",ippGetStatusString(status));

result:
pSrc = 123456789
pDst = 123456
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Set

Initializes vector elements to a specified common value.

Syntax

```
IppStatus ippsSet_8u(Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsSet_16s(Ipp16s val, Ipp16s* pDst, int len);
IppStatus ippsSet_16sc(Ipp16sc val, Ipp16sc* pDst, int len);
IppStatus ippsSet_32s(Ipp32s val, Ipp32s* pDst, int len);
IppStatus ippsSet_32f(Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsSet_32sc(Ipp32sc val, Ipp32sc* pDst, int len);
IppStatus ippsSet_32fc(Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsSet_64s(Ipp64s val, Ipp64s* pDst, int len);
IppStatus ippsSet_64f(Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsSet_64sc(Ipp64sc val, Ipp64sc* pDst, int len);
IppStatus ippsSet_64fc(Ipp64fc val, Ipp64fc* pDst, int len);
```

Parameters

pDst	Pointer to the vector to be initialized.
len	Number of elements to initialize.
val	Value used to initialize the vector *pDst*.

Description

The function `ippsSet` is declared in the `ipps.h` file. This function initializes the first *len* elements of the real or complex vector *pDst* to contain the same value *val*.

Example 4-3 shows how to use the function `ippsSet`.

Example 4-3 Using the `ippsSet` Function

```
IppStatus set(void) {
      char src[] = "set";
      return ippsSet_8u('0', src, strlen(src));
}
```

Example 4-3 Using the `ippsSet` Function

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Zero

Initializes a vector to zero.

Syntax

```
IppStatus ippsZero_8u(Ipp8u* pDst, int len);
IppStatus ippsZero_16s(Ipp16s* pDst, int len);
IppStatus ippsZero_32s(Ipp32s* pDst, int len);
IppStatus ippsZero_32f(Ipp32f* pDst, int len);
IppStatus ippsZero_64s(Ipp64s* pDst, int len);
IppStatus ippsZero_64f(Ipp64f* pDst, int len);
IppStatus ippsZero_16sc(Ipp16sc* pDst, int len);
IppStatus ippsZero_32sc(Ipp32sc* pDst, int len);
IppStatus ippsZero_32fc(Ipp32fc* pDst, int len);
IppStatus ippsZero_64sc(Ipp64sc* pDst, int len);
IppStatus ippsZero_64fc(Ipp64fc* pDst, int len);
```

Parameters

pDst	Pointer to the vector to be initialized to zero.
len	Number of elements to initialize.

Description

The function `ippsZero` is declared in the `ipps.h` file. This function initializes the first *len* elements of the vector *pDst* to 0. If *pDst* is a complex vector, both real and imaginary parts are zeroed.

Example 4-4 shows how to use the function `ippsZero`.

Example 4-4 Using the ippsZero Function

```
IppStatus zero(void) {
   char src[] = "zero";
   return ippsZero_8u(src, strlen(src));
}
```

Return Values

`ippStsNoErr`	Indicates no error
`ippStsNullPtrErr`	Indicates an error when the *pDst* pointer is `NULL`
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0

Sample-Generating Functions

This section describes Intel IPP functions which generate tone samples, triangle samples, pseudo-random samples with uniform distribution, and pseudo-random samples with Gaussian distribution, as well as special test samples.

Some sample-generating functions operate with data in the fixed point format. These functions have `Q15` suffix in their name. This means that integer data are used in calculations inside the function as real numbers equal to the integer value multiplied by 2^{-15} (where "`15`" is called a scale factor).

Tone-Generating Functions

The functions described below generate a tone (or "sinusoid") of a given frequency, phase, and magnitude. Tones are fundamental building blocks for analog signals. Thus, sampled tones are extremely useful in signal processing systems as test signals and as building blocks for more complex signals.

The use of tone functions is preferable against the analogous C math library's `sin()` function for many applications, because Intel IPP functions can use information retained from the computation of the previous sample to compute the next sample much faster than standard `sin()` or `cos()`.

ToneInitAllocQ15

Allocates memory and initializes the tone generator specification structure for fixed point data.

Syntax

```
IppStatus ippsToneInitAllocQ15_16s(IppToneState_16s** pToneState, Ipp16s magn,
    Ipp16s rFreqQ15, Ipp32s phaseQ15);
```

Parameters

pToneState	Pointer to the pointer to the tone generator specification structure.
magn	Magnitude of the tone, that is, the maximum value attained by the wave.
phaseQ15	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
rFreqQ15	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].

Description

The function `ippsToneInitAllocQ15` is declared in the `ipps.h` file. This function allocates memory and initializes the tone generator structure *pToneState* with the specified frequency *rFreqQ15*, phase *phaseQ15*, and magnitude *magn*.
Input data in the fixed point format Q15 are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0, 2π) for phase.
Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.
Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Example 4-5 demonstrartes how to use the function `ippsToneInitAllocQ15`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pToneState* pointer is `NULL`.
`ippStsToneMagnErr`	Indicates an error when *magn* is less than or equal to zero.
`ippStsToneFreqErr`	Indicates an error when *rFreqQ15* is negative, or greater than 16383.

ippStsTonePhaseErr	Indicates an error when the *phaseQ15* value is negative, or greater than 205886.

ToneFree

Frees memory allocated by the function ippsToneInitAllocQ15.

Syntax

```
IppStatus ippsToneFree(IppToneState_16s* pToneState);
```

Parameters

pToneState	Pointer to the tone generator specification structure.

Description

The function ippsToneFree is declared in the ipps.h file. This function closes the tone generator state by freeing all memory associated with the structure created by ippsToneInitAllocQ15.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointer *pToneState* is NULL.

ToneGetStateSizeQ15

Computes the length of the tone generator structure.

Syntax

```
IppStatus ippsToneGetStateSizeQ15_16s(int* pToneStateSize);
```

Parameters

pToneStateSize	Pointer to the computed value of size in bytes of the generator specification structure.

Description

The function `ippsToneGetStateSizeQ15` is declared in the `ipps.h` file. This function computes the length (in bytes) *pToneStateSize* of the tone generator structure. The function `ippsToneGetStateSizeQ15` should be called before the call to the function `ippsToneInitQ15`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointer *pToneStateSize* is `NULL`.

ToneInitQ15

Initializes the tone generator specification structure for fixed point data.

Syntax

```
IppStatus ippsToneInitQ15_16s(IppToneState_16s* pToneState, Ipp16s magn,
    Ipp16s rFreqQ15, Ipp32s phaseQ15);
```

Parameters

pToneState	Pointer to the tone generator specification structure.
magn	Magnitude of the tone, that is, the maximum value attained by the wave.
phaseQ15	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
rFreqQ15	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].

Description

The function `ippsToneInitQ15` is declared in the `ipps.h` file. This function initializes the tone generator structure *pToneState* with the specified frequency *rFreqQ15*, phase *phaseQ15*, and magnitude *magn*. The structure is allocated in the external buffer, the size of which should be computed by the function `ippsToneGetStateSizeQ15`.
Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0, 2π) for phase.
Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to

present a 32-bit value in the fixed point format.
Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pToneState* pointer is NULL.
ippStsToneMagnErr	Indicates an error when *magn* is less than or equal to zero.
ippStsToneFreqErr	Indicates an error when *rFreqQ15* is negative, or greater than 16383.
ippStsTonePhaseErr	Indicates an error when the *phaseQ15* value is negative, or greater than 205886.

ToneQ15

Generates a tone with a frequency, phase, and magnitude specified in the tone generator structure.

Syntax

```
IppStatus ippsToneQ15_16s(Ipp16s* pDst, int len, IppToneState_16s* pToneState);
```

Parameters

pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.
pToneState	Pointer to the tone generator specification structure.

Description

The function ippsToneQ15 is declared in the ipps.h file. This function generates the tone with the frequency, phase, and magnitude parameters that are specified in the previously created structure *pToneState*. The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values *x*[*n*] are computed using the same formulas as in ippsToneDirect function for computing real tones.

Example 4-5 demonstrartes how to use the function ippsToneQ15.

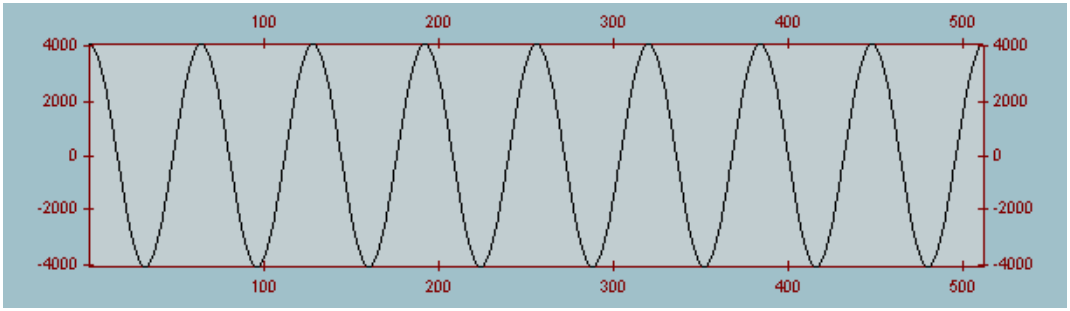
Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pToneState* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.

Example 4-5 Using Tone Generating Functions

```
void func_toneq15()
{
    IppToneState_16s *TS;
    Ipp16s magn = 4095;
    Ipp16s rFreqQ15 = 512;
    Ipp16s phaseQ15 = 0;
    Ipp16s* pDst;
    IppStatus status;

    status = ippsToneInitAllocQ15_16s(&TS,magn,rFreqQ15,phaseQ15);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));

    status = ippsToneQ15_16s(pDst,rFreqQ15,TS);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));
}
```

result:



Tone_Direct

Generates a tone with a given frequency, phase, and magnitude.

Syntax

```
IppStatus ippsTone_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, float rFreq,
    float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn, float
    rFreq, float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_32f(Ipp32f* pDst, int len, float magn, float rFreq,
    float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_32fc(Ipp32fc* pDst, int len, float magn, float rFreq,
    float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_64f(Ipp64f* pDst, int len, double magn, double rFreq,
    double* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_64fc(Ipp64fc* pDst, int len, double magn, double rFreq,
    double* pPhase, IppHintAlgorithm hint);
```

Parameters

magn	Magnitude of the tone, that is, the maximum value attained by the wave.
pPhase	Pointer to the phase of the tone relative to a cosine wave. It must be in range [0.0, 2π). The returned value may be used to compute the next continuous data block.
rFreq	Frequency of the tone relative to the sampling frequency. It must be in the interval [0.0, 0.5) for real tone and in [0.0, 1.0) for complex tone.
pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."

Description

The function `ippsTone_Direct` is declared in the `ipps.h` file. This function generates the tone with the specified frequency *rFreq*, phase *pPhase*, and magnitude *magn*. The function computes *len* samples of the tone, and stores them in the array *pDst*. For real tones, each generated value $x[n]$ is defined as:

$$x[n] = magn \cdot \cos(2\pi n \cdot rFreq + phase)$$

For complex tones, $x[n]$ is defined as:

$$x[n] = magn \cdot (\cos(2\pi n \cdot rFreq + phase) + j \cdot \sin(2\pi n \cdot rFreq + phase))$$

The *hint* argument suggests using specific code, which provides for either fast but less accurate calculation, or more accurate but slower execution. The values you can enter for the *hint* argument are listed in Table 7-3.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pPhase* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.
`ippStsToneMagnErr`	Indicates an error when *magn* is less than or equal to zero.
`ippStsToneFreqErr`	Indicates an error when *rFreq* is negative, or greater than or equal to 0.5 for real tone and to 1.0 for complex tone.
`ippStsTonePhaseErr`	Indicates an error when the *pPhase* value is negative, or greater than or equal to `IPP_2PI`.

ToneQ15_Direct

Generates a tone with a given frequency, phase, and magnitude.

Syntax

```
IppStatus ippsToneQ15_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, Ipp16s rFreqQ15,
    Ipp32s phaseQ15);
```

Parameters

pDst	Pointer to the array which stores the samples.

magn	Magnitude of the tone, that is, the maximum value attained by the wave.
rFreqQ15	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
phaseQ15	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].

Description

The function `ippsToneQ15_Direct` is declared in the `ipps.h` file. This function generates the tone with the specified frequency *rFreqQ15*, phase *pPhaseQ15*, and magnitude *magn*. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0, 2π) for phase.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.
Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values *x*[*n*] are computed using the same formulas as in `ippsTone_Direct` function for computing real tones.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.
`ippStsToneMagnErr`	Indicates an error when *magn* is less than or equal to zero.
`ippStsToneFreqErr`	Indicates an error when *rFreqQ15* is negative, or greater than 16383.
`ippStsTonePhaseErr`	Indicates an error when the *phaseQ15* value is negative, or greater than 205886.

Triangle-Generating Functions

This section describes the functions that generate a periodic signal with a triangular wave form (referred to as “triangle”) of a given frequency, phase, magnitude, and asymmetry.

Application Notes

A real periodic signal with triangular wave form $x[n]$ (referred to as a real triangle) of a given frequency *rFreq*, phase value *phase*, magnitude *magn*, and asymmetry h is defined as follows:

$$x[n] = magn \cdot \mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase), \quad n = 0, 1, 2, \ldots$$

A complex periodic signal with triangular wave form $x[n]$ (referred to as a complex triangle) of a given frequency *rFreq*, phase value *phase*, magnitude *mag*, and asymmetry h is defined as follows:

$$x[n] = magn \cdot (\mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase) + j \cdot \mathbf{st}_h(2\pi \cdot rFreq \cdot n + phase)), \quad n = 0, 1, 2, \ldots$$

The $\mathbf{ct}_h()$ function is determined as follows:

$$H = \pi + h$$

$$\mathbf{ct}_h(\alpha) = \begin{cases} -\frac{2}{H} \cdot \left(\alpha - \frac{H}{2}\right), & 0 \le \alpha \le H \\ \frac{2}{2\pi - H} \cdot \left(\alpha - \frac{2\pi + H}{2}\right), & H \le \alpha \le 2\pi \end{cases}$$

$$\mathbf{ct}_h(\alpha + k \cdot 2\pi) = \mathbf{ct}_h(\alpha), \quad k = 0, \pm 1, \pm 2, \ldots$$

When $H = \pi$, asymmetry $h = 0$, and function $\mathbf{ct}_h()$ is symmetric and a triangular analog of the cos() function. Note the following equations:

$$\mathbf{ct}_h(H/2 + k \cdot \pi) = 0, \quad k = 0, \pm 1, \pm 2, \ldots$$

$$\mathbf{ct}_h(k \cdot 2\pi) = 1, \quad k = 0, \pm 1, \pm 2, \ldots$$

$$\mathbf{ct}_h(H + k \cdot 2\pi) = -1, \quad k = 0, \pm 1, \pm 2, \ldots$$

The $\mathbf{st}_h()$ function is determined as follows:

$$\mathbf{st}_h(\alpha) = \begin{cases} \frac{2}{2\pi - H} \cdot \alpha, & 0 \le \alpha \le \frac{2\pi - H}{2} \\ -\frac{2}{H} \cdot (\alpha - \pi), & \frac{2\pi - H}{2} \le \alpha \le \frac{2\pi + H}{2} \\ \frac{2}{2\pi - H} \cdot (\alpha - 2\pi), & \frac{2\pi + H}{2} \le \alpha \le 2\pi \end{cases}$$

$$\mathbf{st}_h(\alpha + k \cdot 2\pi) = \mathbf{st}_h(\alpha), \quad k = 0, \pm 1, \pm 2, \ldots$$

When $H = \pi$, asymmetry $h = 0$, and function $\mathbf{st}_h()$ is a triangular analog of a sine function. Note the following equations:

$$\mathbf{st}_h(\alpha) = \mathbf{ct}_h(\alpha + (3\pi + h)/2)$$

$$\mathbf{st}_h(\pi k) = 0, \quad k = 0, \pm 1, \pm 2, \ldots$$

$$\mathbf{st}_h((\pi - h)/2 + 2\pi k) = 1, \quad k = 0, \pm 1, \pm 2, \ldots$$

$$\mathbf{st}_h((3\pi + h)/2 + 2\pi k) = -1, \quad k = 0, \pm 1, \pm 2, \ldots$$

TriangleInitAllocQ15

Allocates memory and initializes the triangle generator specification structure for fixed point data.

Syntax

```
IppStatus ippsTriangleInitAllocQ15_16s(IppTriangleState_16s** pTriangleState,
    Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

Parameters

pTriangleState	Pointer to the pointer to the triangle generator specification structure.
magn	Magnitude of the tone, that is, the maximum value attained by the wave.
rFreqQ15	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
phaseQ15	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].

asymQ15	Asymmetry *h* of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If *h*=0, then the triangle is symmetric and a direct analog of a tone.

Description

The function `ippsTriangleInitAllocQ15` is declared in the `ipps.h` file. This function allocates memory and initializes the triangle generator structure *pTriangleState* with the specified frequency *rFreqQ15*, phase *phaseQ15*, asymmetry *asymQ15* and magnitude *magn*.
Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2π) for the phase, and (-π, π) for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.
Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTriandleState* pointer is `NULL`.
`ippStsTriangleMagnErr`	Indicates an error when *magn* is less than or equal to zero.
`ippStsTriangleFreqErr`	Indicates an error when *rFreqQ15* is negative, or greater than 16383.
`ippStsTrianglePhaseErr`	Indicates an error when the *phaseQ15* value is negative, or greater than 205886.
`ippStsTriangleAsymErr`	Indicates an error when the *asymQ15* value is less than -102943 or greater than 102943.

TriangleFree

Frees memory allocated by the function `ippsTriangleInitAlloc`.

Syntax

```
IppStatus ippsTriangleFree(IppTriangleState_16s* pTriangleState);
```

Parameters

pTriangleState	Pointer to the triangle generator specification structure.

Description

The function `ippsTriangleFree` is declared in the `ipps.h` file. This function closes the triangle generator state by freeing all memory associated with the structure created by `ippsTriangleInitAllocQ15`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointer *pTriangleState* is `NULL`.

TriangleGetStateSizeQ15

Computes the length of the triangle generator structure.

Syntax

```
IppStatus ippsTriangleGetStateSizeQ15_16s(int* pTriangleStateSize);
```

Parameters

pTriangleStateSize	Pointer to the computed value of size in bytes of the generator specification structure.

Description

The function `ippsTriangleGetStateSizeQ15` is declared in the `ipps.h` file. This function computes the length (in bytes) *pTriangleStateSize* of the tone generator structure. The function `ippsTriangleGetStateSizeQ15` should be called before the call to the function `ippsTriangleInitQ15`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointer *pTriangleStateSize* is `NULL`.

TriangleInitQ15

Initializes the triangle generator specification structure for fixed point data.

Syntax

```
IppStatus ippsTriangleInitQ15_16s(IppTriangleState_16s* pTriangleState,
    Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

Parameters

pTriangleState	Pointer to the triangle generator specification structure.
magn	Magnitude of the tone, that is, the maximum value attained by the wave.
rFreqQ15	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
phaseQ15	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
asymQ15	Asymmetry *h* of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If *h*=0, then the triangle is symmetric and a direct analog of a tone.

Description

The function `ippsTriangleInitQ15` is declared in the `ipps.h` file. This function initializes the triangle generator structure *pTriangleState* with the specified magnitude *magn*, frequency *rFreqQ15*, phase *phaseQ15*, and asymmetry *asymQ15*. The structure is allocated in the external buffer, the size of which should be computed by the function

ippsTriangleGetStateSizeQ15.
Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2π) for the phase, and (-π, π) for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.
Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pTriandleState* pointer is NULL.
ippStsTriangleMagnErr	Indicates an error when *magn* is less than or equal to zero.
ippStsTriangleFreqErr	Indicates an error when *rFreqQ15* is negative, or greater than 16383.
ippStsTrianglePhaseErr	Indicates an error when the *phaseQ15* value is negative, or greater than 205886.
ippStsTriangleAsymErr	Indicates an error when the *asymQ15* value is less than -102943 or greater than 102943.

TriangleQ15

Generates a triangle with a frequency, phase, and magnitude specified in the triangle generator structure.

Syntax

```
IppStatus ippsTriangleQ15_16s(Ipp16s* pDst, int len, IppTriangleState_16s*
    pTriangleState);
```

Parameters

pDst	Pointer to the array which stores the generated samples.
len	Number of samples to be computed.

pTriangleState	Pointer to the triangle generator specification structure.

Description

The function `ippsTriangleQ15` is declared in the `ipps.h` file. This function generates the triangle with the frequency, phase, magnitude, and asymmetry parameters that are specified in the previously created structure *pTriangleState*. The function computes *len* samples of the triangle, and stores them in the array *pDst*. Generated values *x*[*n*] are computed using the same formulas as in `ippsTriangle_Direct` function for computing real triangles.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pToneState* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

Triangle_Direct

Generates a triangle with a given frequency, phase, and magnitude.

Syntax

```
IppStatus ippsTriangle_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, float rFreq,
    float asym, float* pPhase);
IppStatus ippsTriangle_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn, float rFreq,
    float asym, float* pPhase);
IppStatus ippsTriangle_Direct_32f(Ipp32f* pDst, int len, float magn, float rFreq,
    float asym, float* pPhase);
IppStatus ippsTriangle_Direct_32fc(Ipp32fc* pDst, int len, float magn, float rFreq,
    float asym, float* pPhase);
IppStatus ippsTriangle_Direct_64f(Ipp64f* pDst, int len, double magn, double rFreq,
    double asym, double* pPhase);
IppStatus ippsTriangle_Direct_64fc(Ipp64fc* pDst, int len, double magn, double rFreq,
    double asym, double* pPhase);
```

Parameters

rFreq	Frequency of the triangle relative to the sampling frequency. It must be in range [0.0, 0.5).
pPhase	Pointer to the phase of the triangle relative to a cosine triangular analog wave. It must be in range [0.0, 2π). The returned value may be used to compute the next continuous data block.
magn	Magnitude of the triangle, that is, the maximum value attained by the wave.
asym	Asymmetry h of a triangle. It must be in range [-π, π). If h=0, then the triangle is symmetric and a direct analog of a tone.
pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.

Description

The function `ippsTriangle` is declared in the `ipps.h` file. This function generates the triangle with the specified frequency *rFreq*, phase pointed by *pPhase*, and magnitude *magn*. The function computes *len* samples of the triangle, and stores them in the array *pDst*. For real triangle, $x[n]$ is defined as:

$$x[n] = magn \cdot \mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase), \quad n = 0, 1, 2, \ldots$$

For complex triangles, $x[n]$ is defined as:

$$x[n] = magn \cdot (\mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase) + j \cdot \mathbf{st}_h(2\pi \cdot rFreq \cdot n + phase)), \quad n = 0, 1, 2, \ldots$$

See “Application Notes” on page 4-19 for the definition of functions $\mathbf{ct}_h$ and $\mathbf{st}_h$.

Example 4-6 demonstrartes how to use the function `ippsTriangle`.

Return Values

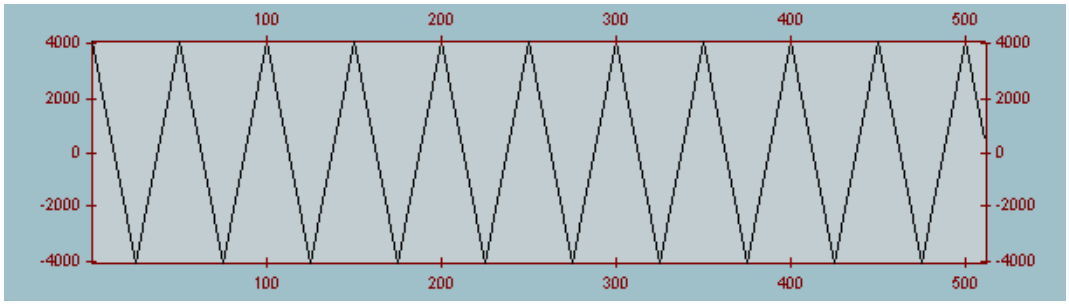
`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pPhase* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.
`ippStsTrnglMagnErr`	Indicates an error when *magn* is less than or equal to zero.
`ippStsTrnglFreqErr`	Indicates an error when *rFreq* is negative, or greater than or equal to 0.5.
`ippStsTrnglPhaseErr`	Indicates an error when the *pPhase* value is negative, or greater than or equal to `IPP_2PI`.

`ippStsTrnglAsymErr` Indicates an error when *asym* is less than `-IPP_PI`, or greater than or equal to `IPP_PI`.

Example 4-6 Using the Triangle Generating Functions

```
void func_triangle_direct()
{
    Ipp16s* pDst;
    int len = 512;
    Ipp16s magn = 4095;
    float rFreq = 0.02;
    float asym = 0.0;;
    float Phase = 0.0;
    IppStatus status;

    status = ippsTriangle_Direct_16s(pDst, len, magn, rFreq, asym,
&Phase);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));
}
```

result:



TriangleQ15_Direct

Generates a triangle with a given frequency, phase, and magnitude for fixed point data.

Syntax

```
IppStatus ippsTriangleQ15_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, Ipp16s
    rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

Parameters

pDst	Pointer to the array which stores the samples.
magn	Magnitude of the tone, that is, the maximum value attained by the wave.
rFreqQ15	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
phaseQ15	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
asymQ15	Asymmetry *h* of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If *h*=0, then the triangle is symmetric and a direct analog of a tone.

Description

The function `ippsTriangleQ15_Direct` is declared in the `ipps.h` file. This function generates the triangle with the specified magnitude *magn*, frequency *rFreqQ15*, phase *pPhaseQ15*, and asymmetry *asymQ15*. Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2π) for the phase, and [-π, π) for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format.
Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values *x*[*n*] are computed using the same formulas as in `ippsTriangle_Direct` function for computing real triangles.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.
`ippStsToneMagnErr`	Indicates an error when *magn* is less than or equal to zero.
`ippStsToneFreqErr`	Indicates an error when *rFreqQ15* is negative, or greater than 16383.
`ippStsTonePhaseErr`	Indicates an error when the *phaseQ15* value is negative, or greater than 205886.

ippStsTriangleAsymErr	Indicates an error when the *asymQ15* value is less than -102943 or greater than 102943.

Uniform Distribution Functions

This section describes the functions that generate pseudo-random samples with uniform distribution.

RandUniformInitAlloc

Allocates memory and initializes a noise generator with uniform distribution.

Syntax

```
IppStatus ippsRandUniformInitAlloc_8u(IppsRandUniState_8u** pRandUniState,
    Ipp8u low, Ipp8u high, unsigned int seed);
IppStatus ippsRandUniformInitAlloc_16s(IppsRandUniState_16s** pRandUniState,
    Ipp16s low, Ipp16s high, unsigned int seed);
IppStatus ippsRandUniformInitAlloc_32f(IppsRandUniState_32f** pRandUniState,
    Ipp32f low, Ipp32f high, unsigned int seed);
```

Parameters

pRandUniState	Pointer to the structure containing parameters for the generator of noise.
low	Lower bound of the uniform distribution range.
high	Upper bound of the uniform distribution range.
seed	Seed value used by the pseudo-random number generation algorithm.

Description

The function `ippsRandUniformInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the pseudo-random generator state *pRandUniState*. The uniform distribution range is specified by the lower and upper bounds *low* and *high*, respectively.

Example 4-7 demonstrartes how to use the function `ippsRandUniformInitAlloc`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRandUniState* pointer is NULL.
ippStsMemAllocErr	Indicates an error when there is not enough memory for the operation.

RandUniformFree

Closes the uniform distribution generator state.

Syntax

```
IppStatus ippsRandUniformFree_8u(IppsRandUniState_8u* pRandUniState);
IppStatus ippsRandUniformFree_16s(IppsRandUniState_16s* pRandUniState);
IppStatus ippsRandUniformFree_32f(IppsRandUniState_32f* pRandUniState);
```

Parameters

pRandUniState	Pointer to the structure containing parameters for the generator of noise.

Description

The function ippsRandUniformFree is declared in the ipps.h file. This function closes the noise generator state *pRandUniState* by freeing all memory allocated by the ippsRandUniInitAlloc function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRandUniState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

RandUniformInit

Initializes a noise generator with uniform distribution.

Syntax

```
IppStatus ippsRandUniformInit_16s(IppsRandUniState_16s* pRandUniState, Ipp16s low,
    Ipp16s high, unsigned int seed);
```

Parameters

pRandUniState	Pointer to the structure containing parameters for the generator of noise.
low	Lower bound of the uniform distribution range.
high	Upper bound of the uniform distribution range.
seed	Seed value used by the pseudo-random number generation algorithm.

Description

The function `ippsRandUniformInit` is declared in the `ipps.h` file. This function initializes the pseudo-random generator state *pRandUniState* in the external buffer. The size of this buffer should be computed previously by calling the function `ippsRandUniformGetSize`. The uniform distribution range is specified by the lower and upper bounds *low* and *high*, respectively.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pRandUniState* pointer is `NULL`.
`ippStsMemAllocErr`	Indicates an error when there is not enough memory for the operation.

RandUniformGetSize

Computes the length of the uniform distribution generator structure.

Syntax

```
IppStatus ippsRandUniformGetSize_16s(int* pRandUniStateSize);
```

Parameters

pRandUniStateSize	Pointer to the computed value of size in bytes of the generator specification structure.

Description

The function `ippsRandUniformGetSize` is declared in the `ipps.h` file. This function computes the length (in bytes) *pRandUniStateSize* of the uniform distribution generator structure. The function `ippsRandUniformGetSize` should be called prior to the function `ippsRandUniformInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointer *pRandUniStateSize* is `NULL`.

RandUnifrom

Generates the pseudo-random samples with a uniform distribution.

Syntax

```
IppStatus ippsRandUniform_8u(Ipp8u* pDst,  int len, IppsRandUniState_8u*
    pRandUniState);
IppStatus ippsRandUniform_16s(Ipp16s* pDst, int len, IppsRandUniState_16s*
    pRandUniState);
IppStatus ippsRandUniform_32f(Ipp32f* pDst, int len, IppsRandUniState_32f*
    pRandUniState);
```

Parameters

pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.
pRandUniState	Pointer to the structure containing parameters for the generator of noise.

Description

The function `ippsRandUniform` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a uniform distribution and stores them in the array *pDst*. Initial parameters of the generator are set in the generator state structure *pRandUniState*.
Before calling `ippsRandUniform`, you must initialize the generator state by calling the `ippsRandUniformInitAlloc` function.

Example 4-7 demonstrartes how to use the function `ippsRandUniform`.

Return Values

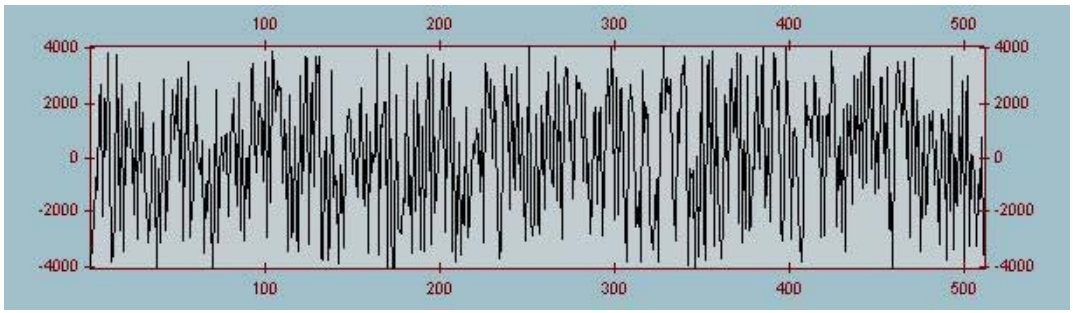
`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pRandUniState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

Example 4-7 Using the Function to Generate Pseudo-Random Signal

```
void func_randuniform()
{
    IppsRandUniState_16s* pRUS;
    Ipp16s low, high;
    low = -4096;
    high = 4095;
    unsigned int seed = 0;
    Ipp16s* pDst;
    int len = 512;
    IppStatus status;

    status = ippsRandUniformInitAlloc_16s(&pRUS,low,high,seed);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));

    status = ippsRandUniform_16s(pDst,len, pRUS);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));
}
```

Example 4-7 Using the Function to Generate Pseudo-Random Signal

```
result:
```



RandUniform_Direct

Generates the pseudo-random samples with a uniform distribution in direct mode.

Syntax

```
IppStatus ippsRandUniform_Direct_16s(Ipp16s* pDst, int len, Ipp16s low, Ipp16s high,
    unsigned int* pSeed);
IppStatus ippsRandUniform_Direct_32f(Ipp32f* pDst, int len, Ipp32f low, Ipp32f high,
    unsigned int* pSeed);
IppStatus ippsRandUniform_Direct_64f(Ipp64f* pDst, int len, Ipp64f low, Ipp64f high,
    unsigned int* pSeed);
```

Parameters

pSeed	Pointer to the seed value used by the pseudo-random number generation algorithm.
low	Lower bound of the uniform distribution range.
high	Upper bound of the uniform distribution range.
pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.

Description

The function `ippsRandUniform_Direct` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a uniform distribution and stores them in the array *pDst*. This function does not require to initialize the generator state structure in advance. All parameters of the pseudo-random number generator are set directly in the function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSeed* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Gaussian Distribution Functions

This section describes the function that generates pseudo-random samples with Gaussian distribution.

RandGaussInitAlloc

Allocates memory and initializes a noise generator with Gaussian distribution.

```
IppStatusippsRandGaussInitAlloc_8u(IppsRandGaussState_8u**pRandGaussState,Ipp8u
    mean, Ipp8u stdDev, unsigned int seed);
IppStatus ippsRandGaussInitAlloc_16s(IppsRandGaussState_16s** pRandGaussState,
    Ipp16s mean, Ipp16s stdDev, unsigned int seed);
IppStatus ippsRandGaussInitAlloc_32f(IppsRandGaussState_32f** pRandGaussState,
    Ipp32f mean, Ipp32f stdDev, unsigned int seed);
```

Parameters

pRandGaussState	Pointer to the structure containing parameters for the generator of noise.
mean	Mean of the Gaussian distribution.
stdDev	Standard deviation of the Gaussian distribution.

seed	Seed value used by the pseudo-random number generator algorithm.

Description

The function `ippsRandGaussInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the pseudo-random generator state structure *pRandGaussState*. This structure contains parameters of the required noise generator that are specified by the *mean*, *stdDev* and *seed* values.

Return Values

`ppStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pRandGaussState* pointer is `NULL`.
`ippStsMemAllocErr`	Indicates an error when there is not enough memory for the operation.

RandGaussFree

Closes the Gaussian distribution generator state.

Syntax

```
IppStatus ippsRandGaussFree_8u(IppsRandGaussState_8u* pRandGaussState);
IppStatus ippsRandGaussFree_16s(IppsRandGaussState_16s* pRandGaussState);
IppStatus ippsRandGaussFree_32f(IppsRandGaussState_32f* pRandGaussState);
```

Parameters

pRandGaussState	Pointer to the structure containing parameters for the generator of noise.

Description

The function `ippsRandGaussFree` is declared in the `ipps.h` file. This function closes the noise generator state *pRandGaussState* by freeing all memory allocated by the `ippsRandGaussInitAlloc` function.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pRandGaussState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

RandGaussGetSize

Computes the length of the Gaussian distribution generator structure.

Syntax

```
IppStatus ippsRandGaussGetSize_16s(int* pRandGaussStateSize);
```

Parameters

pRandGaussStateSize	Pointer to the computed value of size in bytes of the generator specification structure.

Description

The function ippsRandGaussGetSize is declared in the ipps.h file. This function computes the length (in bytes) *pRandGaussStateSize* of the uniform distribution generator structure. The function ippsRandGaussGetSize should be called before calling the function ippsRandGaussInit.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointer *pRandGaussStateSize* is NULL.

RandGaussInit

Initializes a noise generator with Gaussian distribution.

Syntax

```
IppStatus ippsRandGaussInit_16s(IppsRandGaussState_16s* pRandGaussState, Ipp16s
    mean, Ipp16s stdDev, unsigned int seed);
```

Parameters

pRandGaussState	Pointer to the structure containing parameters for the generator of noise.
mean	Mean of the Gaussian distribution.
stdDev	Standard deviation of the Gaussian distribution.
seed	Seed value used by the pseudo-random number generator algorithm.

Description

The function `ippsRandGaussInit` is declared in the `ipps.h` file. This function initializes the pseudo-random generator state structure *pRandGaussState* in the external buffer. The size of this buffer should be computed previously by calling the function `ippsRandGaussGetSize`. This structure contains parameters of the required noise generator that are specified by the *mean*, *stdDev* and *seed* values.

Return Values

`ppStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pRandGaussState* pointer is `NULL`.
`ippStsMemAllocErr`	Indicates an error when there is not enough memory for the operation.

RandGauss

Generates the pseudo-random samples with a Gaussian distribution.

Syntax

```
IppStatus ippsRandGauss_8u(Ipp8u* pDst,  int len, IppsRandGaussState_8u*
    pRandGaussState);
IppStatus ippsRandGauss_16s(Ipp16s* pDst, int len, IppsRandGaussState_16s*
    pRandGaussState);
IppStatus ippsRandGauss_32f(Ipp32f* pDst, int len, IppsRandGaussState_32f*
    pRandGaussState);
```

Parameters

pDst	Pointer to the array which stores the samples.
len	Number of samples to be computed.
pRandGaussState	Pointer to the structure containing parameters of the noise generator.

Description

The function `ippsRandGauss` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a Gaussian distribution and stores them in the array *pDst*. The initial parameters of the generator are set in the generator state structure *pRandGaussState*.
Before calling `ippsRandGauss`, you must initialize the generator state by calling the `ippsRandGaussInitAlloc` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pRandGaussState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

RandGauss_Direct

Generates pseudo-random samples with a Gaussian distribution in the direct mode.

Syntax

```
IppStatus ippsRandGauss_Direct_16s(Ipp16s* pDst, int len, Ipp16s mean,
    Ipp16s stdev, unsigned int* pSeed);
IppStatus ippsRandGauss_Direct_32f(Ipp32f* pDst, int len, Ipp32f mean,
    Ipp32f stdev, unsigned int* pSeed);
IppStatus ippsRandGauss_Direct_64f(Ipp64f* pDst, int len, Ipp64f mean,
    Ipp64f stdev, unsigned int* pSeed);
```

Parameters

pDst	Pointer to the array which stores the samples.
pSeed	Pointer to the seed value used by the pseudo-random number generation algorithm.
len	Number of samples to be computed.
mean	Mean of the Gaussian distribution.
stdev	Standard deviation of the Gaussian distribution.

Description

The function `ippsRandGauss_Direct` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a Gaussian distribution, and stores them in the array *pDst*.This function does not require to initialize the generator state structure in advance. All parameters of the pseudo-random number generator are set directly in the function.

Example 4-8 demonstrartes how to use the function `ippsRandGauss_Direct`.

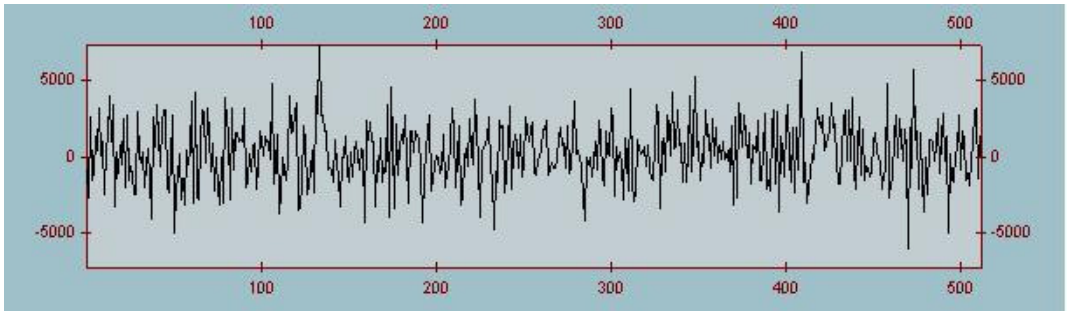
Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSeed* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 4-8 Using the Function ippsaRandGauss_Direct

```
void func_gauss_direct()
{
    Ipp16s* pDst;
    int len = 512;
    Ipp16s mean = 0;
    Ipp16s stdev = 2047;
    unsigned int Seed = 0;
    IppStatus status;

    status = ippsRandGauss_Direct_16s(pDst, len, mean, stdev, &Seed);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));
}
```

result:



Special Vector Functions

The functions described in this section create special vectors that can be used as a test signals to examine the effect of applying different signal processing functions.

VectorJaehne

Creates a Jaehne vector.

Syntax

```
IppStatus ippsVectorJaehne_8u(Ipp8u* pDst, int len, Ipp8u magn);
```

```
IppStatus ippsVectorJaehne_8s(Ipp8s* pDst, int len, Ipp8s magn);
IppStatus ippsVectorJaehne_16u(Ipp16u* pDst, int len, Ipp16u magn);
IppStatus ippsVectorJaehne_16s(Ipp16s* pDst, int len, Ipp16s magn);
IppStatus ippsVectorJaehne_32u(Ipp32u* pDst, int len, Ipp32u magn);
IppStatus ippsVectorJaehne_32s(Ipp32s* pDst, int len, Ipp32s magn);
IppStatus ippsVectorJaehne_32f(Ipp32f* pDst, int len, Ipp32f magn);
IppStatus ippsVectorJaehne_64f(Ipp64f* pDst, int len, Ipp64f magn);
```

Parameters

pDst	Pointer to the destination vector.
len	Number of elements in the vector.
magn	Magnitude of the signal to be generated.

Description

The function `ippsVectorJaehne` is declared in the `ipps.h` file. This function creates a Jaehne vector and stores the result in *pDst*. The magnitude *magn* must be positive. The function generates the sinusoid with a variable frequency. The computation is performed as follows:

$$pDst[n] = magn \cdot \sin\left(\frac{0.5\pi n^2}{len}\right), \quad 0 \le n < len$$

Example 4-9 shows how to use the function `ippsVectorJaehne`.

Example 4-9 Using the ippsVectorJaehne Function

```
IppStatus Jaehne (void) {
    Ipp16s buf[100] ;
    return ippsVectorJaehne_16s ( buf, 100, 255 );
}
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsJaehneErr	Indicates an error when *magn* is negative.

VectorSlope

Creates a slope vector.

Syntax

```
IppStatus ippsVectorSlope_8u(Ipp8u* pDst, int len, Ipp32f offset, Ipp32f
    slope);
IppStatus ippsVectorSlope_8s(Ipp8s* pDst, int len, Ipp32f offset, Ipp32f
    slope);
IppStatus ippsVectorSlope_16u(Ipp16u* pDst, int len, Ipp32f offset,
    Ipp32f slope);
IppStatus ippsVectorSlope_16s(Ipp16s* pDst, int len, Ipp32f offset,
    Ipp32f slope);
IppStatus ippsVectorSlope_32u(Ipp32u* pDst, int len, Ipp64f offset,
    Ipp64f slope);
IppStatus ippsVectorSlope_32s(Ipp32s* pDst, int len, Ipp64f offset,
    Ipp64f slope);
IppStatus ippsVectorSlope_32f(Ipp32f* pDst, int len, Ipp32f offset,
    Ipp32f slope);
IppStatus ippsVectorSlope_64f(Ipp64f* pDst, int len, Ipp64f offset,
    Ipp64f slope);
```

Parameters

pDst	Pointer to the destination vector.
len	Number of elements in the vector.
offset	Offset value.
slope	Slope coefficient.

Description

The function `ippsVectorSlope` is declared in the `ipps.h` file. This function creates a slope vector and stores the result in *pDst*. The destination vector elements are computed according to the following formula:

$pDst[n] = offset + slope*n, \quad 0 \le n < len.$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

VectorRamp

Creates a ramp vector.

Syntax

```
IppStatus ippsVectorRamp_8u(Ipp8u* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_8s(Ipp8s* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_16u(Ipp16u* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_16s(Ipp16s* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_32u(Ipp32u* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_32s(Ipp32s* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_32f(Ipp32f* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_64f(Ipp64f* pDst, int len, float offset, float slope);
```

Parameters

pDst	Pointer to the destination vector.
len	Number of elements in the vector.
offset	Offset value.
slope	Slope coefficient.

Description

The function ippsVectorRamp is declared in the ipps.h file. This function creates a ramp vector and stores the result in *pDst*. The destination vector elements are computed according to the following formula:

$pDst[n] = offset + slope*n, \quad 0 \le n < len.$

Note that this function is similar to the function ippsVectorSlope, but the linear transform coefficients *offset* and *slope* have floating-point values for all flavors of the function ippsVectorRamp. In most cases the use of the function ippsVectorSlope is more preferrable.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Essential Functions

5

This chapter describes Intel® IPP functions that perform logical and shift, arithmetic, conversion, windowing, and statistical operations.

The full list of functions in this group is given in Table 5-1.

Table 5-1 Intel IPP Essential Vector Functions

Function Base Name	Operation
Logical and Shift Functions	
AndC	Computes the bitwise AND of a scalar value and each element of a vector.
And	Computes the bitwise AND of two vectors.
OrC	Computes the bitwise OR of a scalar value and each element of a vector.
Or	Computes the bitwise OR of two vectors.
XorC	Computes the bitwise XOR of a scalar value and each element of a vector.
Xor	Computes the bitwise XOR of two vectors.
Not	Computes the bitwise NOT of the vector elements.
LShiftC	Shifts bits in vector elements to the left.
RShiftC	Shifts bits in vector elements to the right.
Arithmetic Functions	
AddC	Adds a constant value to each element of a vector.
Add	Adds the elements of two vectors.
AddProduct	Adds product of two vectors to the accumulator vector.
AddProductC	Adds product of a vector and a constant to the accumulator vector.

Table 5-1 Intel IPP Essential Vector Functions (continued)

Function Base Name	Operation
MulC	Multiplies each elements of a vector by a constant value.
Mul	Multiplies the elements of two vectors.
SubC	Subtracts a constant value from each element of a vector.
SubCRev	Subtracts each element of a vector from a constant value.
Sub	Subtracts the elements of two vectors.
DivC	Divides each element of a vector by a constant value.
DivCRev	Divides a constant value by each element of a vector.
Div	Divides the elements of two vectors.
Div_Round	Divides the elements of two vectors with rounding.
Abs	Computes absolute values of vector elements.
Sqr	Computes a square of each element of a vector.
Sqrt	Computes a square root of each element of a vector.
Cubrt	Computes cube root of each element of a vector.
Exp	Computes e to the power of each element of a vector.
Ln	Computes the natural logarithm of each element of a vector.
10Log10	Computes the decimal logarithm of each element of a vector and multiplies it by 10.
SumLn	Sums natural logarithms of each element of a vector.
Arctan	Computes the inverse tangent of each element of a vector.
Normalize	Normalizes elements of a real or complex vector using offset and division operations.
Conversion Functions	
SortAscend, SortDescend	Rearranges all elements of a vector.
SortIndexAscend, SortIndexDescend	Rearranges all elements of the vector and their indexes.
SwapBytes	Reverses byte order of a vector.

Table 5-1 Intel IPP Essential Vector Functions (continued)

Function Base Name	Operation
Convert	Converts the data type of a vector and stores the results in a second vector.
Join	Converts the floating-point data of several vectors to integer data, and stores the results in a single vector.
JoinScaled	Converts with scaling the floating-point data of several vectors to integer data, and stores the results in a single vector.
SplitScaled	Converts with scaling the integer data of a vector to floating-point data, and stores the result in several vectors.
Conj	Stores the complex conjugate values of a vector in a second vector or in-place.
ConjFlip	Computes the complex conjugate of a vector and stores the result in reverse order.
Magnitude	Computes the magnitudes of the elements of a complex vector.
MagSquared	Computes the squared magnitudes of the elements of a complex vector.
Phase	Computes the phase angles of elements of a complex vector.
PowerSpectr	Computes the power spectrum of a complex vector.
Real	Returns the real part of a complex vector in a second vector.
Imag	Returns the imaginary part of a complex vector in a second vector.
RealToCplx	Returns a complex vector constructed from the real and imaginary parts of two real vectors.
CplxToReal	Returns the real and imaginary parts of a complex vector in two respective vectors.
Threshold Threshold_LT, Threshold_GT Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal	Performs the threshold operation on the elements of a vector by limiting the element values by *level*.

Table 5-1 Intel IPP Essential Vector Functions (continued)

Function Base Name	Operation
Threshold_LTAbs, Threshold_GTAbs	Performs the threshold operation on the absolute values of elements of a vector.
Threshold_LTInv	Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.
CartToPolar	Converts the elements of a complex vector to polar coordinate form.
PolarToCart	Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.
MaxOrder	Computes the maximum order of a vector.
Preemphasize	Computes preemphasis of a single precision real signal in-place.
Flip	Reverses the order of elements in a vector.
FindNearestOne	Finds an element of the table which is closest to the specified value.
FindNearest	Finds table elements that are closest to the elements of the specified vector.
Viterbi Decoder Functions	
GetVarPointDV	Fills the array with the information about points that are closest to the received point.
CalcStatesDV	Calculates the states of the Viterbi decoder.
BuildSymblTableDV4D	Fills the array with the information about possible 4D symbols.
UpdatePathMetricsDV	Searches for the state with the minimum path metric.
Windowing Functions	
WinBartlett	Multiplies a vector by a Bartlett windowing function.
WinBlackman	Multiplies a vector by a Blackman windowing function.
WinHamming	Multiplies a vector by a Hamming windowing function.
WinHann	Multiplies a vector by a Hann windowing function.
WinKaiser	Multiplies a vector by a Kaiser windowing function.
Statistical Functions	
Sum	Computes the sum of the elements of a vector.
Max	Returns the maximum value of a vector.

Table 5-1 Intel IPP Essential Vector Functions (continued)

Function Base Name	Operation
MaxIndx	Returns the maximum value of a vector and the index of the maximum element.
MaxAbs	Returns the maximum absolute value of a vector.
MaxAbsIndx	Returns the maximum absolute value of a vector and the index of the corresponding element
Min	Returns the minimum value of a vector.
MinIndx	Returns the minimum value of a vector and the index of the minimum element.
MinAbs	Returns the minimum absolute value of a vector.
MinAbsIndx	Returns the minimum absolute value of a vector and the index of the corresponding element
MinMax	Returns the maximum and minimum values of a vector.
MinMaxIndx	Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.
Mean	Computes the mean value of a vector.
StdDev	Computes the standard deviation value of a vector.
MeanStdDev	Computes the mean value and the standard deviation value of a vector.
Norm	Computes the C, L1, L2, or L2Sqr norm of a vector.
NormDiff	Computes the C, L1, L2, or L2Sqr norm of two vectors' difference.
DotProd	Computes the dot product of two vectors.
MaxEvery, MinEvery	Computes maximum or minimum value for each pair of elements of two vectors.
CountInRange	Computes the number of elements of the vector whose values are in the specified range.
Sampling Functions	
SampleUp	Up-samples a signal, conceptually increasing its sampling rate by an integer factor.
SampleDown	Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

Logical and Shift Functions

This section describes the Intel IPP signal processing functions that perform logical and shift operations on vectors. Logical and shift functions are only defined for integer arguments.

For binary logical operations AND, OR and XOR, the following functions are provided:
`AndC`, `OrC`, `XorC` for vector-scalar operations
`And`, `Or`, `Xor` for vector-vector operations.

AndC

Computes the bitwise AND of a scalar value and each element of a vector.

Syntax

```
IppStatus ippsAndC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsAndC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsAndC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);
IppStatus ippsAndC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsAndC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsAndC_32u_I(Ipp32u val, Ipp32u* pSrcDst, int len);
```

Parameters

val	Input scalar value.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsAndC` is declared in the `ipps.h` file. This function computes the bitwise AND of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsAndC` compute the bitwise AND of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

And

Computes the bitwise AND of two vectors.

Syntax

```
IppStatus ippsAnd_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len);
IppStatus ippsAnd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsAnd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
IppStatus ippsAnd_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsAnd_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsAnd_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

Parameters

pSrc1, pSrc2	Pointers to the two source vectors.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector for the in-place operation.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsAnd` is declared in the `ipps.h` file. This function computes the bitwise AND of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsAnd` compute the bitwise AND of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

OrC

Computes the bitwise OR of a scalar value and each element of a vector.

Syntax

```
IppStatus ippsOrC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsOrC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsOrC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);
IppStatus ippsOrC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsOrC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsOrC_32u_I(Ipp16u val, Ipp32u* pSrcDst, int len);
```

Parameters

val	Input scalar value.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsOrC` is declared in the `ipps.h` file. This function computes the bitwise OR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsOrC` compute the bitwise OR of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Or

Computes the bitwise OR of two vectors.

Syntax

```
IppStatus ippsOr_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len);
IppStatus ippsOr_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsOr_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
IppStatus ippsOr_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsOr_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsOr_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

Parameters

pSrc1, pSrc2	Pointers to the two source vectors.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector for the in-place operation.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsOr` is declared in the `ipps.h` file. This function computes the bitwise OR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsOr` compute the bitwise OR of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

XorC

Computes the bitwise XOR of a scalar value and each element of a vector.

Syntax

```
IppStatus ippsXorC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsXorC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsXorC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);
IppStatus ippsXorC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsXorC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsXorC_32u_I(Ipp32u val, Ipp32u* pSrcDst, int len);
```

Parameters

val	Input scalar value.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsXorC` is declared in the `ipps.h` file. This function computes the bitwise XOR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of ippsXorC compute the bitwise XOR of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Xor

Computes the bitwise XOR of two vectors.

Syntax

```
IppStatus ippsXor_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len);
IppStatus ippsXor_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsXor_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
IppStatus ippsXor_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsXor_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsXor_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

Parameters

pSrc1, pSrc2	Pointers to the two source vectors.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector for the in-place operation.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function ippsXor is declared in the ipps.h file. This function computes the bitwise XOR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsXor` compute the bitwise XOR of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Not

Computes the bitwise NOT of the vector elements.

Syntax

```
IppStatus ippsNot_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsNot_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsNot_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsNot_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsNot_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsNot_32u_I(Ipp32u* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsNot` is declared in the `ipps.h` file. This function computes the bitwise NOT of the corresponding elements of the vectors *pSrc*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsNot` compute the bitwise NOT of the corresponding elements of the vector *pSrcDst* and store the result in the vector *pSrcDst*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LShiftC

Shifts bits in vector elements to the left.

Syntax

```
IppStatus ippsLShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int len);
IppStatus ippsLShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst, int len);
IppStatus ippsLShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst, int len);
IppStatus ippsLShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst, int len);
IppStatus ippsLShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippsLShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsLShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsLShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);
```

Parameters

val	Number of bits by which the function shifts each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function ippsLShiftC is declared in the ipps.h file. This function shifts each element of the vector *pSrc* by *val* bits to the left, and stores the result in *pDst*.

The in-place flavors of `ippsLShiftC` shift each element of the vector *pSrcDst* by *val* bits to the left and store the result in *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

RShiftC

Shifts bits in vector elements to the right.

Syntax

```
IppStatus ippsRShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int len);
IppStatus ippsRShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst, int len);
IppStatus ippsRShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst, int len);
IppStatus ippsRShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst, int len);
IppStatus ippsRShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippsRShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsRShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsRShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);
```

Parameters

val	Number of bits by which the function shifts each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsRShiftC` is declared in the `ipps.h` file. This function shifts each element of the vector *pSrc* by *val* bits to the right, and stores the result in *pDst*.

The in-place flavors of `ippsRShiftC` shift each element of the vector *pSrcDst* by *val* bits to the right and store the result in *pSrcDst*.

Note that the arithmetic shift is realized for signed data, and the logical shift for unsigned data.

Example 5-1 shows how the logical and shift functions can be used in the saturate operation. The data are converted to the unsigned char range [0...255].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 5-1 Using the Logical and Shift Functions

```
void saturate(void) {
      Ipp16s x[8] = {1000, -257, 127, 4, 5, 0, 7, 8}, lo[8], hi[8];
      IppStatus status = ippsNot_16u((Ipp16u*)x, (Ipp16u*)lo, 8);
      ippsRShiftC_16s_I(15, lo, 8);
      ippsCopy_16s(x, hi, 8);
      ippsSubCRev_16s_ISfs(255, hi, 8, 0);
      ippsRShiftC_16s_I(15, hi, 8);
      ippsAnd_16u_I((Ipp16u*)lo, (Ipp16u*)x, 8);
      ippsOr_16u_I((Ipp16u*)hi, (Ipp16u*)x, 8);
      ippsAndC_16u_I(255, (Ipp16u*)x, 8);
      printf_16s("saturate =", x, 8, status);
}
Output:
    saturate =  255 0 127 4 5 0 7 8
```

Arithmetic Functions

This section describes the Intel IPP signal processing functions that perform vector arithmetic operations on vectors. The arithmetic functions include basic element-wise arithmetic operations between vectors, as well as more complex calculations such as computing absolute values, square and square root, natural logarithm and exponential of vector elements.

Intel IPP software provides two versions of each function. One version performs the operation in-place, while the other stores the results of the operation in a different destination vector, that is, executes an out-of-place operation.

AddC

Adds a constant value to each element of a vector.

Syntax

Case 1. Not-in-place operations on floating point data.

```
IppStatus ippsAddC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsAddC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsAddC_32fc(const Ipp32fc* pSrc, Ipp32fc val,Ipp32fc* pDst, int len);
IppStatus ippsAddC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
```

Case 2. Not-in-place operations on integer data.

```
IppStatus ippsAddC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsAddC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsAddC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsAddC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len,
    int scaleFactor);
IppStatus ippsAddC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst,
    int len, int scaleFactor);
IppStatus ippsAddC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst,
    int len, int scaleFactor);
```

Case 3. In-place operations on floating point data.

```
IppStatus ippsAddC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsAddC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsAddC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsAddC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsAddC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

Case 4. In-place operations on integer data.

```
IppStatus ippsAddC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAddC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
val	Scalar value used to increment each element of the source vector
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsAddC` is declared in the `ipps.h` file. This function adds a value *val* to each element of the source vector *pSrc*, and stores the result in the destination vector *pDst*.

The in-place flavors of `ippsAddC` add a value *val* to each element of the vector *pSrcDst*, and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Add

Adds the elements of two vectors.

Syntax

Case 1. Not-in-place operations on floating point data, and integer data without scaling.

```
IppStatus ippsAdd_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int len);
IppStatus ippsAdd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsAdd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
IppStatus ippsAdd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
IppStatus ippsAdd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, int len);
IppStatus ippsAdd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
    int len);
IppStatus ippsAdd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
    int len);
IppStatus ippsAdd_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsAdd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f* pDst,
    int len);
```

Case 2. Not-in-place operations on integer data with scaling.

```
IppStatus ippsAdd_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsAdd_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, Ipp32sc*
    pDst, int len, int scaleFactor);
```

Case 3. In-place operations on floating point data, and integer data without scaling.

```
IppStatus ippsAdd_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsAdd_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsAdd_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsAdd_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsAdd_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsAdd_16s32s_I(const Ipp16s* pSrc, Ipp32s* pSrcDst, int len);
```

Case 4. In-place operations on integer data with scaling.

```
IppStatus ippsAdd_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsAdd_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAdd_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAdd_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAdd_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAdd_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len, int
    scaleFactor)
```

Parameters

pSrc1, pSrc2	Pointers to the source vectors.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector for in-place operations.
pSrcDst	Pointer to the source and destination vector for in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsAdd` is declared in the `ipps.h` file. This function adds the elements of the vector *pSrc1* to the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of `ippsAdd` add the elements of the vector *pSrc* to the elements of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Example 5-2 shows that overflow does not occur while adding big numbers due to the scaling operation.

Return Values

ippStsNoErr	Indicates no error
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0

Example 5-2 Using the ippsAdd Function

```
IppStatus add(void) {
      Ipp16s x[4] = {-1, 32767, 2, 30000};
      IppStatus st = ippsAdd_16s_ISfs(x, x, 4, 1);
      printf_16s("add =", x, 4, st);
      return st;
}
Output:
    add = -1 32767 2 30000
```

AddProduct

Adds product of two vectors to the accumulator vector.

Syntax

Case 1. Operations on floating point data.

```
IppStatusippsAddProduct_32f(constIpp32f*pSrc1,constIpp32f*pSrc2,Ipp32f*pSrcDst,
    int len);
IppStatusippsAddProduct_64f(constIpp64f*pSrc1,constIpp64f*pSrc2,Ipp64f*pSrcDst,
    int len);
IppStatus ippsAddProduct_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
    pSrcDst, int len);
IppStatus ippsAddProduct_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
    pSrcDst, int len);
```

Case 2. Operations on integer data with scaling.

```
IppStatus ippsAddProduct_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pSrcDst, int len, int scaleFactor);
IppStatus ippsAddProduct_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsAddProduct_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32s*
    pSrcDst, int len, int scaleFactor);
```

Parameters

pSrc1, pSrc2	Pointers to the source vectors.
pSrcDst	Pointer to the destination accumulator vector.
len	The number of elements in the vectors.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsAddProduct` is declared in the `ipps.h` file. This function multiplies each element of the source vector *pSrc1* by the corresponding element of the vector *pSrc2*, and adds the result to the corresponding element of the accumulator vector *pSrcDst* as given by:

$$pSrcDst[n] = pSrcDst[n] + pSrc1[n] * pSrc2[n], \quad 0 \le n < len$$

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

AddProductC

Adds product of a vector and a constant to the accumulator vector.

Syntax

```
IppStatus ippsAddProductC_32f(const Ipp32f* pSrc, const Ipp32f val,
    Ipp32f* pSrcDst, int len);
```

Parameters

pSrc	Pointers to the source vector.

val	The scalar value used to multiply each element of the source vector *pSrc*.
pSrcDst	Pointer to the destination accumulator vector.
len	The number of elements in the vectors.

The function `ippsAddProductC` is declared in the `ipps.h` file. This function multiplies each element of the source vector *pSrc* by a value *val* and adds the result to the corresponding element of the accumulator vector *pSrcDst* as given by:

$$pSrcDst[n] = pSrcDst[n] + pSrc[n] * val, \quad 0 \le n < len$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *len* is less than or equal to 0.

MulC

Multiplies each elements of a vector by a constant value.

Syntax

Case 1. Not-in-place operations without scaling.

```
IppStatus ippsMulC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsMulC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsMulC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsMulC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);

IppStatus ippsMulC_Low_32f16s(const Ipp32f* pSrc, Ipp32f val, Ipp16s*  pDst,
    int len);
```

Case 2. Not-in-place operations with scaling.

```
IppStatus ippsMulC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int
    scaleFactor);
IppStatus ippsMulC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len,
    int scaleFactor);
```

```
IppStatus ippsMulC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsMulC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len,
    int scaleFactor);
IppStatus ippsMulC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int
    len, int scaleFactor);
IppStatus ippsMulC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst, int
    len, int scaleFactor);
IppStatus ippsMulC_32f16s_Sfs(const Ipp32f* pSrc, Ipp32f val, Ipp16s*  pDst,
    int len, int scaleFactor);
```

Case 3. In-place operations without scaling.

```
IppStatus ippsMulC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsMulC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsMulC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsMulC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsMulC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

Case 4. In-place operations with scaling.

```
IppStatus ippsMulC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
val	The scalar value used to multiply each element of the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operation.
len	The number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMulC` is declared in the `ipps.h` file. This function multiplies each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsMulC` multiply each element of the vector *pSrcDst* by a value *val* and store the result in *pSrcDst*.

Function flavor with **Low** suffix in its name requires that each value of the product *pSrc***val* does not exceed the `Ipp32s` data type range.

Function flavors with `Sfs` suffix perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Mul

Multiplies the elements of two vectors.

Syntax

Case 1. Not-in-place operations on floating point data, and integer data without scaling.

```
IppStatus ippsMul_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
    int len);
IppStatus ippsMul_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
    int len);
IppStatus ippsMul_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
    int len);
IppStatus ippsMul_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
    pDst, int len);
IppStatus ippsMul_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
    pDst, int len);
```

```
IppStatus ippsMul_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp16u* pDst,
    int len);
IppStatus ippsMul_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsMul_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
    pDst, int len);
```

Case 2. Not-in-place operations on integer data with scaling.

```
IppStatus ippsMul_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsMul_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, Ipp32sc*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_16u16s_Sfs(const Ipp16u* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32s*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc* pSrc2,
    Ipp32sc* pDst, int len, int scaleFactor);

IppStatus ippsMul_Low_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
    Ipp32s* pDst, int len, int scaleFactor);
```

Case 3. In-place operations on floating point data and integer data without scaling.

```
IppStatus ippsMul_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMul_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMul_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsMul_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsMul_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsMul_32f32fc_I(const Ipp32f* pSrc, Ipp32fc* pSrcDst, int len);
```

Case 4. In-place operations on integer data with scaling.

```
IppStatus ippsMul_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int
    scaleFactor);
IppStatus, ippsMul_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsMul_32s32sc_ISfs(const Ipp32s* pSrc, Ipp32sc* pSrcDst, int len,
    int scaleFactor);
```

Parameters

pSrc1, pSrc2	Pointers to the source vectors.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector for in-place operation.
pSrcDst	Pointer to the source and destination vector for in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMul` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2* and stores the result in *pDst*.

The in-place flavors of `ippsMul` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst* and store the result in *pSrcDst*.

Function flavors with **`Sfs`** suffix perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Function flavor with **`Low`** suffix in its name requires that each value of the product does not exceed the `Ipp32s` data type range.

Return Values

ippStsNoErr	Indicates no error
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0

SubC

Subtracts a constant value from each element of a vector.

Syntax

Case 1. Not-in-place operations on floating point data.

```
IppStatus ippsSubC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsSubC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int
    len);
IppStatus ippsSubC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsSubC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int
    len);
```

Case 2. Not-in-place operations on integer data.

```
IppStatus ippsSubC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsSubC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int
    len, int scaleFactor);
IppStatus ippsSubC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int
    len, int scaleFactor);
IppStatus ippsSubC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int
    len, int scaleFactor);
IppStatus ippsSubC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst,
    int len, int scaleFactor);
IppStatus ippsSubC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst,
    int len, int scaleFactor);
```

Case 3. In-place operations on floating point data.

```
IppStatus ippsSubC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsSubC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsSubC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

Case 4. In-place operations on integer data.

```
IppStatus ippsSubC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
val	Scalar value used to decrement each element of the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSubC` is declared in the `ipps.h` file. This function subtracts a value *val* from each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsSubC` subtract a value *val* from each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

SubCRev

Subtracts each element of a vector from a constant value.

Syntax

Case 1. Not-in-place operations on floating point data.

```
IppStatus ippsSubCRev_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int
    len);
IppStatus ippsSubCRev_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int
    len);
IppStatus ippsSubCRev_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int
    len);
IppStatus ippsSubCRev_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int
    len);
```

Case 2. Not-in-place operations on integer data.

```
IppStatus ippsSubCRev_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int
    len, int scaleFactor);
IppStatus ippsSubCRev_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int
    len, int scaleFactor);
IppStatus ippsSubCRev_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int
    len, int scaleFactor);
IppStatus ippsSubCRev_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int
    len, int scaleFactor);
IppStatus ippsSubCRev_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst,
    int len, int scaleFactor);
IppStatus ippsSubCRev_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst,
    int len, int scaleFactor);
```

Case 3. In-place operations on floating point data.

```
IppStatus ippsSubCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubCRev_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsSubCRev_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubCRev_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

Case 4. In-place operations on integer data.

```
IppStatus ippsSubCRev_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubCRev_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubCRev_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubCRev_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubCRev_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubCRev_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
    scaleFactor);
```

Parameters

val	Scalar value from which vector elements are subtracted.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the vector whose elements are to be subtracted from the value *val* in case of the in-place operation. The destination vector which stores the result of the subtraction *val* - *pSrcDst*[*n*].
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSubCRev` is declared in the `ipps.h` file. This function subtracts each element of the vector *pSrc* from a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsSubCRev` subtract each element of the vector *pSrcDst* from a value *val* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Sub

Subtracts the elements of two vectors.

Syntax

Case 1. Not-in-place operations on floating point data, and integer data without scaling.

```
IppStatus ippsSub_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
    int len);
IppStatus ippsSub_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
    int len);
IppStatus ippsSub_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
    int len);
IppStatus ippsSub_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
    pDst, int len);
IppStatus ippsSub_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
    pDst, int len);
IppStatus ippsSub_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f*
    pDst, int len);
```

Case 2. Not-in-place operations on integer data with scaling.

```
IppStatus ippsSub_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsSub_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
    pDst, int len, int scaleFactor);
IppStatus ippsSub_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsSub_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pDst, int len, int scaleFactor);
IppStatus ippsSub_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc*
    pDst, int len, int scaleFactor);
```

```
IppStatus ippsSub_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, Ipp32sc*
    pDst, int len, int scaleFactor);
```

Case 3. In-place operations on floating point data and integer data without scaling.

```
IppStatus ippsSub_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsSub_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsSub_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsSub_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsSub_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
```

Case 4. In-place operations on integer data with scaling.

```
IppStatus ippsSub_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSub_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len, int
    scaleFactor);
```

Parameters

pSrc1	Pointer to the source vector-subtrahend, whose elements are to be subtracted.
pSrc2	Pointer to the source vector-minuend from whose elements the elements of *pSrc1* are to be subtracted.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector-subtrahend for in-place operation.
pSrcDst	Pointer to the source vector-minuend and destination vector for in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function ippsSub is declared in the ipps.h file. This function subtracts the elements of the vector *pSrc1* from the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of ippsSub subtract the elements of the vector *pSrc* from the elements of a vector *pSrcDst* and store the result in *pSrcDst*.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

ippStsNoErr	Indicates no error
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0

DivC

Divides each element of a vector by a constant value.

Syntax

Case 1. Not-in-place operations on floating point data.

```
IppStatus ippsDivC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsDivC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsDivC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsDivC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
```

Case 2. Not-in-place operations on integer data with scaling.

```
IppStatus ippsDivC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len,
    int ScaleFactor);
IppStatus ippsDivC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int
    len, int scaleFactor);
IppStatus ippsDivC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int
    len, int ScaleFactor);
IppStatus ippsDivC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst,
    int len, int ScaleFactor);
```

Case 3. In-place operations on floating point data.

```
IppStatus ippsDivC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsDivC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsDivC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsDivC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

Case 4. In-place operations on integer data with scaling.

```
IppStatus ippsDivC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int ScaleFactor);
IppStatus ippsDivC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsDivC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int ScaleFactor);
IppStatus ippsDivC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int ScaleFactor);
```

Parameters

val	Scalar value used as a divisor.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDivC` is declared in the `ipps.h` file. This function divides each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsDivC` divide each element of the vector *pSrcDst* by a value *val* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

ippStsDivByZeroErr	Indicates an error when *val* is equal to 0.

DivCRev

Divides a constant value by each element of a vector.

Syntax

```
IppStatus ippsDivCRev_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsDivCRev_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsDivCRev_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsDivCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

Parameters

val	Constant value used as a dividend in the operation.
pSrc	Pointer to the source vector whose elements are used as divisors.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operation.
len	Number of elements in the vector

Description

The function ippsDivCRev is declared in the ipps.h file. This function divides the constant value *val* by each element of the vector *pSrc* and stores the results in *pDst*.

The in-place flavors of ippsDivC divide the constant value *val* by each element of the vector *pSrcDst* and store the results in *pSrcDst*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZeroErr	Indicates an error when *val* is equal to 0.

Div

Divides the elements of two vectors.

Syntax

Case 1. Not-in-place operations on integer data.

```
IppStatus ippsDiv_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsDiv_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
    pDst, int len, int scaleFactor);
IppStatus ippsDiv_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsDiv_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
    pDst, int len, int scaleFactor);
IppStatus ippsDiv_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippsDiv_32s16s_Sfs(const Ipp16s* pSrc1, const Ipp32s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
```

Case 2. Not-in-place operations on floating point data.

```
IppStatus ippsDiv_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
    int len);
IppStatus ippsDiv_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
    int len);
IppStatus ippsDiv_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
    int len);
IppStatus ippsDiv_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
    int len);
```

Case 3. In-place operations on integer data.

```
IppStatus ippsDiv_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int
    ScaleFactor);
IppStatus ippsDiv_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsDiv_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    ScaleFactor);
```

```
IppStatus ippsDiv_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
    ScaleFactor);
IppStatus ippsDiv_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    ScaleFactor);
```

Case 4. In-place operations on floating point data

```
IppStatus ippsDiv_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsDiv_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsDiv_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsDiv_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
```

Parameters

pSrc1	Pointer to the vector whose elements are used as divisors.
pSrc2	Pointer to the vector whose elements are used as dividends.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector whose elements are used as divisors for in-place operations.
pSrcDst	Pointer to the sourve and destination vector for in-place operations.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDiv` is declared in the `ipps.h` file. This function divides the elements of the vector *pSrc2* by the elements of the vector *pSrc1*, and stores the result in *pDst*.

The in-place flavors of `ippsDiv` divide the elements of the vector *pSrcDst* by the elements of the vector *pSrc* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

If the function `ippsDiv` encounters a zero-valued divisor vector element, it returns a warning status and continues execution with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

Example 5-3 shows that the use of the scaling factor in the integer functions increases operation accuracy. Example 5-4 considers division by zero exceptions (x / 0, 0 / 0).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZero	Indicates a warning for zero-valued divisor vector element.

Example 5-3 Using the ippsDiv_16s_ISfs Function

```
IppStatus div16s( void ) {
      Ipp16s x[4] = { -3, 2, 0, 300 };
      Ipp16s y[4] = { -2, 2, 0, 0 };
      IppStatus st = ippsDiv_16s_ISfs( y, x, 4, -1 );
      printf_16s("div16s =", x, 4, st );
      return st;
}
Output:
    -- warning 6, Zero value(s) in the divisor of the function Div
    div16s =  3 2 0 32767
```

Example 5-4 Using the ippsDiv_32f_I Function

```
IppStatus div32f( void ) {
      Ipp32f x[4] = { -3, 2, 0, 300 };
      Ipp32f y[4] = { -2, 2, 0, 0 };
      IppStatus st = ippsDiv_32f_I( y, x, 4 );
      printf_32f( "div32f =", x, 4, st );
      return st;
}
Output:
    -- warning 6, Zero value(s) in the divisor of the function Div
    div32f =  1.500000 1.000000 1.#IND00 1.#INF00
```

Div_Round

Divides the elements of two vectors with rounding.

Syntax

Case 1. Not-in-place operations on integer data.

```
IppStatus ippsDiv_Round_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
    Ipp8u* pDst, int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsDiv_Round_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u*
    pSrc2, Ipp16u* pDst, int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsDiv_Round_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pDst, int len, IppRoundMode rndMode, int scaleFactor);
```

Case 2. In-place operations on integer data.

```
IppStatus ippsDiv_Round_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int
    len, IppRoundMode rndMode, int scaleFactor)
IppStatus ippsDiv_Round_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst,
    int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsDiv_Round_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
    int len, IppRoundMode rndMode, int scaleFactor);
```

Parameters

pSrc1	Pointer to the vector whose elements are used as divisors.
pSrc2	Pointer to the vector whose elements are used as dividends.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector whose elements are used as divisors for in-place operations.
pSrcDst	Pointer to the sourve and destination vector for in-place operations.
len	Number of elements in the vector
roundMode	Rounding mode, the following values are possible:
	`ippRndZero` — Specifies that floating-point values must be truncated toward zero.
	`ippRndNear` — Specifies that floating-point values must be rounded to the nearest even integer.

ippRndFinancial	Specifies that floating-point values must be rounded down to the nearest integer if decimal value is less than 0.5, or rounded up to the nearest integer if decimal value is equal or greater than 0.5.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function ippsDiv_Round is declared in the ipps.h file. This function divides the elements of the vector *pSrc2* by the elements of the vector *pSrc1*, the result is rounded using the rounding method specified by the parameter *roundMode* and stored in the vector *pDst*.

The in-place flavors of ippsDiv_Round divide the elements of the vector *pSrcDst* by the elements of the vector *pSrc*, the result is rounded using the rounding method specified by the parameter *roundMode* and stored in the vector *pSrcDst*.

Functions perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

If the function ippsDiv_Round encounters a zero-valued divisor vector element, it returns a warning status and continues execution with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZero	Indicates a warning for zero-valued divisor vector element. The function execution is continued.
ippStsStsRoundModeNotSupportedErr	Indicates an error condition if the *roundMode* has an illegal value.

Abs

Computes absolute values of vector elements.

Syntax

```
IppStatus ippsAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAbs_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsAbs_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsAbs_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsAbs_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
len	Number of elements in the vector.

Description

The function `ippsAbs` is declared in the `ipps.h` file. This function computes the absolute values of each element of the vector *pSrc* and stores the result in *pDst*.
The in-place flavors of `ippsAbs` compute the absolute values of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

To compute the absolute values of complex data, use the function `ippsMagnitude`.

Example 5-5 shows how to call the function `ippsAbs_32f_I`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 5-5 Using the ippsAbs Function

```
void abs32f(void) {
      Ipp32f x[4] = {-1, 1, 0, 0};
      x[3] *= (-1);
      ippsAbs_32f_I(x, 4);
      printf_32f("abs =", x, 4, ippStsNoErr);
}
Output:
    abs =  1.000000 1.000000 0.000000 0.000000
```

Sqr

Computes a square of each element of a vector.

Syntax

```
IppStatus ippsSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqr_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqr_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqr_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqr_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqr_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqr_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16sc_ISfs(Ipp16sc* pSrcDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSqr` is declared in the `ipps.h` file. This function computes the square of each element of the vector *pSrc*, and stores the result in *pDst*. The computation is performed as follows:

$pDst[n] = pSrc[n]^2$

The in-place flavors of `ippsSqr` compute the square of each element of the vector *pSrcDst* and store the result in *pSrcDst*. The computation is performed as follows:

$pSrcDst[n] = pSrcDst[n]^2$

When computing the square of an integer number, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. Example 5-6 shows how to get the value of 200^2. Without scaling this result is clipped to 32767. Scaling retains the output data range but results in the precision loss in low-order bits.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 5-6 Using the `ippsSqr` Function

```
IppStatus sqr(void) {
      Ipp16s x[4] = {-3, 2, 30, 200};
      IppStatus st = ippsSqr_16s_ISfs(x, 4, 1);
      printf_16s("sqr =", x, 4, st);
      return st;
}
Output:
    sqr =  4 2 450 20000
```

Sqrt

Computes a square root of each element of a vector.

Syntax

```
IppStatus ippsSqrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqrt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqrt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqrt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqrt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqrt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqrt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqrt_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqrt_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsSqrt_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSqrt_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len, int scaleFactor);
IppStatus ippsSqrt_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, int
    scaleFactor);
IppStatus ippsSqrt_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len, int scaleFactor);
IppStatus ippsSqrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsSqrt_64s16s_Sfs(const Ipp64s* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsSqrt_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16sc_ISfs(Ipp16sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.

pSrcDst	Pointer to the source and destination vector for in-place operations.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSqrt` is declared in the `ipps.h` file. This function computes the square root of each element of the vector *pSrc*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = \sqrt{pSrc[n]}$$

The in-place flavors of `ippsSqrt` compute the square root of each element of the vector *pSrcDst* and store the result in *pSrcDst*. The computation is performed as follows:

$$pSrcDst[n] = \sqrt{pSrcDst[n]}$$

The square root of complex vector elements is computed as follows:

$$\sqrt{a + j \cdot b} = \sqrt{\frac{\sqrt{a^2 + b^2} + a}{2}} + j \cdot sign(b) \cdot \sqrt{\frac{\sqrt{a^2 + b^2} - a}{2}}$$

Application Notes

If the function `ippsSqrt` encounters a negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

To increase precision of an integer output, use the scale factor.

Example 5-7 shows how to call the function `ippsSqrt_16s_ISfs`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsSqrtNegArg`	Indicates a warning that a source element has a negative value.

Example 5-7 Using the ippsSqrt Function

```
IppStatus sqrt(void) {
      Ipp16s x[4] = {-3, 2, 30, 300};
      IppStatus st = ippsSqrt_16s_ISfs(x, 4, -1);
      printf_16s("sqrt =", x, 4, st);
      return st;
}
Output:
    -- warning 3, Negative value(s) in the argument of the function Sqrt
    sqrt = 0 3 11 35
```

Cubrt

Computes cube root of each element of a vector.

Syntax

```
IppStatus ippsCubrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCubrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function ippsCubrt is declared in the ipps.h file. This function computes cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

$$pDst[n] = \sqrt[3]{pSrc[n]},\ 0 \le n < len.$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Exp

Computes e *to the power of each element of a vector.*

Syntax

```
IppStatus ippsExp_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExp_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsExp_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsExp_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsExp_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len, int scaleFactor);
IppStatus ippsExp_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector *pSrcDst* for the in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Description

The function ippsExp is declared in the ipps.h file. This function computes the exponential function of each element of the vector *pSrc*, and stores the result in *pDst*.

The computation is performed as follows:

$$pDst[n] = e^{pSrc[n]}$$

The in-place flavors of ippsExp compute the exponential function of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

The computation is performed as follows:

$$pSrcDst[n] = e^{pSrcDst[n]}$$

When an overflow occurs, the function continues operation with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

When computing the exponent of an integer number, the output result can exceed the data range and become saturated. The scaling retains the output data range but results in precision loss in low-order bits. The function ippsExp_32f64f computes the output result in a higher precision data range.

Example 5-8 shows how to call the function ippsExp_16s_ISfs. Example 5-9 shows how to call the function ippsExp_64f_I.

Application Notes

For the functions ippsExp and ippsLn the result is rounded to the nearest integer after scaling.

Example 5-8 Using the ippsExp_16s_ISfs Function

```
IppStatus exp16s(void) {
      Ipp16s x[4] = {-1, 2, 30, 0};
      IppStatus st = ippsExp_16s_ISfs(x, 4, -1);
      printf_16s("exp16s =", x, 4, st);
      return st;
}
Output:
    exp16s =  1 15 32767 2
```

Example 5-9 Using the ippsExp_64f_I Function

```
IppStatus exp64f(void) {
      Ipp64f x[4] = {-1, 2, 1, log(1.234567)};
      IppStatus st = ippsExp_64f_I(x, 4);
      printf_64f("exp64f =", x, 4, st);
      return st;
}
Output:
    exp64f =  0.367879 7.389056 2.718282 1.234567
```

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Ln

Computes the natural logarithm of each element of a vector.

Syntax

```
IppStatus ippsLn_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsLn_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsLn_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsLn_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsLn_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsLn_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsLn_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function ippsLn is declared in the ipps.h file. This function computes the natural logarithm of each element of the vector *pSrc* and stores the result in *pDst* as given by

$$pDst[n] = \log_e(pSrc[n])$$

The in-place flavors of `ippsLn` compute the natural logarithm of each element of the vector *pSrcDst* and store the result in *pSrcDst* as given by

$pSrcDst[n] = \log_e (pSrcDst[n])$

If the function `ippsLn` encounters a zero or negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

Example 5-10 shows how to call the function `ippsLn_32f_I`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsLnZeroArg`	Indicates a warning for zero-valued input vector elements.
`ippStsLnNegArg`	Indicates a warning for negative input vector elements.

Example 5-10 Using the `ippsLn` Function

```
IppStatus ln32f(void) {
      Ipp32f x[4] = {-1, (float)IPP_E, 0, (float)(exp(1.234567))};
      IppStatus st = ippsLn_32f_I(x, 4);
      printf_32f("Ln =", x, 4, st);
      return st;
}
Output:
    -- warning 8, Negative value(s) of argument in the Ln function
    Ln = -1.#IND00 1.000000 -1.#INF00 1.234567
```

10Log10

Computes the decimal logarithm of each element of a vector and multiplies it by 10.

Syntax

```
IppStatus ipps10Log10_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int
    scaleFactor);
IppStatus ipps10Log10_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ipps10Log10` is declared in the `ipps.h` file. This function computes the decimal logarithm of each element of the vector *pSrc*, multiplies it by 10, and stores the result in *pDst* as given by

$$pDst[n] = 10*\log_{10}(pSrc[n])$$

The in-place flavor of `ipps10Log10` computes the decimal logarithm of each element of the vector *pSrcDst*, multiplies it by 10, and stores the result in *pSrcDst* as given by

$$pSrcDst[n] = 10*\log_{10}(pSrcDst[n])$$

If the function `ipps10Log10` encounters a zero or negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsLnZeroArg`	Indicates a warning for zero-valued input vector elements.
`ippStsLnNegArg`	Indicates a warning for negative input vector elements.

SumLn

Sums natural logarithms of each element of a vector.

Syntax

```
IppStatus ippsSumLn_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum);
IppStatus ippsSumLn_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_32f64f(const Ipp32f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pSum);
```

Parameters

pSrc	Pointer to the source vector.
pSum	Pointer to the output result.
len	Number of elements in the vector.

Description

The function `ippsSumLn` is declared in the `ipps.h` file. This function computes the sum of natural logarithms of each element of the vector *pSrc* and stores the result value in *pSum*. The summation is given by:

$$sum = \sum_{n=0}^{len-1} \ln(pSrc[n])$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pSum* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsLnZeroArg`	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to `-Inf`.
`ippStsLnNegArg`	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to `NaN`.

Arctan

Computes the inverse tangent of each element of a vector.

Syntax

```
IppStatus ippsArctan_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsArctan_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsArctan_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsArctan_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector *pSrcDst* for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsAtan` is declared in the `ipps.h` file. This function computes the inverse tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

$$pDst[n] = \arctan(pSrc[n]),\ 0 \le n < len.$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Normalize

Normalizes elements of a real or complex vector using offset and division operations.

Syntax

```
IppStatus ippsNormalize_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f vSub,
    Ipp32f vDiv);
IppStatus ippsNormalize_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f vSub,
    Ipp64f vDiv);
IppStatus ippsNormalize_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, Ipp32fc
    vSub, Ipp32f vDiv);
IppStatus ippsNormalize_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, Ipp64fc
    vSub, Ipp64f vDiv);
IppStatus ippsNormalize_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
    vSub, int vDiv, int scaleFactor);
IppStatus ippsNormalize_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, Ipp16sc
    vSub, int vDiv, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
vSub	Subtrahend value.
vDiv	Denominator value.
pDst	Pointer to the vector which stores the normalized elements.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsNormalize` is declared in the `ipps.h` file. This function subtracts *vSub* from elements of the input vector *pSrc*, divides the differences by *vDiv*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = \frac{pSrc[n] - vsub}{vDiv}$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZeroErr	Indicates an error when *vDiv* is equal to 0 or less than the minimum floating-point positive number.

Conversion Functions

The functions described in this section perform the following conversion operations for vectors:

- Sorting all elements of a vector
- Data type conversion (including floating-point to integer and integer to floating-point)
- Joining several vectors
- Extracting components from a complex vector and constructing a complex vector
- Computing the complex conjugates of vectors
- Cartesian to polar and polar to Cartesian coordinate conversion.

This section also describes the Intel IPP functions that extract real and imaginary components from a complex vector or construct a complex vector using its real and imaginary components.
The functions ippsReal and ippsImag return the real and imaginary parts of a complex vector in a separate vector, respectively.
The function ippsRealToCplx constructs a complex vector from real and imaginary components stored in two respective vectors.
The function ippsCplxToReal returns the real and imaginary parts of a complex vector in two respective vectors.
The function ippsMagnitude computes the magnitude of a complex vector elements.

Additionally this section describes functions that perform the Viterbi decoding for V34 receiver.

SortAscend, SortDescend

Sorts all elements of a vector.

Syntax

```
IppStatus ippsSortAscend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortAscend_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSortAscend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortAscend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortAscend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortAscend_64f_I(Ipp64f* pSrcDst, int len);

IppStatus ippsSortDescend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortDescend_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSortDescend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortDescend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortDescend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortDescend_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

pSrcDst	Pointer to the source and destination vector.
len	Number of elements in the vector.

Description

The functions `ippsSortAscend` and `ippsSortDescend` are declared in the `ipps.h` file. These functions rearrange all elements of the specified vector *pSrcDst* in the ascending or descending order, respectively, and store the result in the destination vector *pSrcDst*.

Example 5-11 shows how to call the function `ippsSortAscend_8u_I`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst* is `NULL`.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-11 Using the Function ippsSortAscend

```
void func_sort()
{
    Ipp8u vec[10] = {0,2,4,5,1,8,9,4,6,7};
    IppStatus status;

    status = ippsSortAscend_8u_I(vec,10);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));
}

Result
      0 1 2 4 4 5 6 7 8 9
```

SortIndexAscend, SortIndexDescend

Rearranges elements of the vector and their indexes.

Syntax

```
IppStatus ippsSortIndexAscend_8u_I(Ipp8u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_16u_I(Ipp16u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_16s_I(Ipp16s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_32s_I(Ipp32s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_32f_I(Ipp32f* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_64f_I(Ipp64f* pSrcDst, int* pDstIdx, int len);

IppStatus ippsSortIndexDescend_8u_I(Ipp8u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_16u_I(Ipp16u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_16s_I(Ipp16s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_32s_I(Ipp32s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_32f_I(Ipp32f* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_64f_I(Ipp64f* pSrcDst, int* pDstIdx, int len);
```

Parameters

pSrcDst	Pointer to the source and destination vector.
pDstIdx	Pointer to the destination vector containing indexes.
len	Number of elements in the vector.

Description

The functions `ippsSortIndexAscend` and `ippsSortIndexDescend` are declared in the `ipps.h` file. These functions rearrange all elements of the specified vector *pSrcDst* in the ascending or descending order, respectively, and store the elements in the destination vector *pSrcDst*, and their indexes in the destionation vector *pDstIdx*. If some elements are identical, their indexes are not ordered.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

SwapBytes

Reverses the byte order of a vector.

Syntax

```
IppStatus ippsSwapBytes_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsSwapBytes_24u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsSwapBytes_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsSwapBytes_64u(const Ipp64u* pSrc, Ipp64u* pDst, int len);
IppStatus ippsSwapBytes_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSwapBytes_24u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSwapBytes_32u_I(Ipp32u* pSrcDst, int len);
IppStatus ippsSwapBytes_64u_I(Ipp64u* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsSwapBytes` is declared in the `ipps.h` file. This function reverses the endian order (byte order) of the source vector *pSrc* (*pSrcDst* for the in-place operation) and stores the result in *pDst* (*pSrcDst*).
When the low-order byte is stored in memory at the lowest address, and the high-order byte at the highest address, the little-endian order is implemented.When the high-order byte is stored in memory at the lowest address, and the low-order byte at the highest address, the big-endian order is implemented. The function `ippsSwapBytes` allows to switch from one order to the other in either direction.

Example 5-12 shows how to call the function `ippsSwapBytes_16u_I`.

Example 5-12 Using the Function `ippsSwapBytes`

```
void func_swap()
{
    Ipp16u vec[2] = {0x1234,0x5678};
    IppStatus status;

    status = ippsSwapBytes_16u_I(vec, 2);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));
}
result
vec[0] = 0x3412
vec[1] = 0x7856
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Convert

Converts the data type of a vector and stores the results in a second vector.

Syntax

```
IppStatus ippsConvert_8s16s(const Ipp8s* pSrc,Ipp16s* pDst, int len);
IppStatus ippsConvert_8s32f(const Ipp8s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16u32f(const Ipp16u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s16s(const Ipp32s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsConvert_32s32f(const Ipp32s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s64f(const Ipp32s* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32f_Sfs(const Ipp16s* pSrc, Ipp32f* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_16s64f_Sfs(const Ipp16s* pSrc, Ipp64f* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32s32f_Sfs(const Ipp32s* pSrc, Ipp32f* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32s64f_Sfs(const Ipp32s* pSrc, Ipp64f* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32f8s_Sfs(const Ipp32f* pSrc, Ipp8s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f8u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16s_Sfs(const Ipp32f* pSrc, Ipp16s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16u_Sfs(const Ipp32f* pSrc, Ipp16u* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
```

```
IppStatus ippsConvert_32f32s_Sfs(const Ipp32f* pSrc, Ipp32s* pDst, int len, IppRoundMode
    rndMode, int scaleFactor);
IppStatus ippsConvert_64s32s_Sfs(const Ipp64s* pSrc, Ipp32s* pDst, int len, IppRoundMode
    rndMode, int scaleFactor);
IppStatus ippsConvert_64f16s_Sfs(const Ipp64f* pSrc, Ipp16s* pDst, int len, IppRoundMode
    rndMode, int scaleFactor);
IppStatus ippsConvert_64f32s_Sfs(const Ipp64f* pSrc, Ipp32s* pDst, int len, IppRoundMode
    rndMode, int scaleFactor);

IppStatus ippsConvert_24u32u(const Ipp8u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsConvert_24u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32u24u_Sfs(const Ipp32u* pSrc, Ipp8u* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32f24u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len, int
    scaleFactor);

IppStatus ippsConvert_24s32s(const Ipp8u* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_24s32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s24s_Sfs(const Ipp32s* pSrc, Ipp8u* pDst, int len, int
    scaleFactor);
IppStatus ippsConvert_32f24s_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len, int
    scaleFactor);

IppStatus ippsConvert_16s16f(const Ipp16s* pSrc, Ipp16f* pDst, int len,
    IppRoundMode rndMode);
IppStatus ippsConvert_32f16f(const Ipp32f* pSrc, Ipp16f* pDst, int len,
    IppRoundMode rndMode);
IppStatus ippsConvert_16f16s_Sfs(const Ipp16f* pSrc, Ipp16s* pDst, int len,
    IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_16f32f(const Ipp16f* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
rndMode	Rounding mode, possible values:

	ippRndZero	Specifies that floating-point values must be truncated toward zero.
	ippRndNear	Specifies that floating-point values must be rounded to the nearest even integer.
	ippRndFinancial	Specifies that floating-point values must be rounded down to the nearest integer if decimal value is less than 0.5, or rounded up to the nearest integer if decimal value is equal or greater than 0.5.
len	Number of elements in the vector	
scaleFactor	Refer to Integer Scaling in Chapter 2.	

Description

The function `ippsConvert` is declared in the `ipps.h` file. This function converts the type of data contained in the vector *pSrc* and stores the results in *pDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. The converted result is saturated if it exceeds the output data range.

Functions that operate with `16f` data do not support the `ippRndFinancial` rounding mode.

Example 5-13 shows how to use the function `ippsConvert`.

Example 5-13 Using the Function `ippsConvert`.

```
Ipp32s src32s[2] = { 33000, -33000 };
Ipp32f src32f[2] = { 126.6, -125.4 };
Ipp32f src_32f[2] = { 113.12, -113.6 };
Ipp32f src1_32f[5] = { -2.5, -2.4, 1.4, 1.5, 1.6};
Ipp8s src_8s[2] = { 125, -125 };
Ipp8u src8[1] = { 255 };
Ipp32f dst32f[1];
Ipp16s dst16[2];
Ipp16s dst_16s[2];
Ipp8u dst8u[2];
Ipp8s dstN8[2];
Ipp8s dstZ8[2];

int scaleFactor = 0;  // no scaling

ippsConvert_8s16s ( src_8s, dst_16s, 2 );
ippsConvert_8u32f ( src8, dst32f, 1 );
ippsConvert_32s16s ( src32s, dst16, 2 );
ippsConvert_32f8s_Sfs ( src32f, dstN8, 2, ippRndNear, scaleFactor );
ippsConvert_32f8s_Sfs ( src32f, dstZ8, 2, ippRndZero, scaleFactor );
ippsConvert_32f8u_Sfs ( src_32f, dst8u, 2, ippRndNear, scaleFactor );
ippsConvert_32f8s_Sfs ( src1_32f, dstF8, 5, ippRndFinancial, scaleFactor );
```

Example 5-13 Using the Function `ippsConvert`.

```
result:
8s16s >>        dst_16s = { 125, -125}
8u32f >>         dst32f  = { 255.0 }
32s16s >>         dst16  =  { 32767, -32768}    // max, min  16s values

// results for scaleFactor = 0
32f8s_Sfs >>    dstN8  =  { 127, -125 }       // scaleFactor = 0
32f8s_Sfs >>    dstZ8  =  { 126, -125 }       // scaleFactor = 0
32f8s_Sfs >>    dstF8  =  { -3, -2, 1, 2, 2 } // scaleFactor = 0
32f8u_Sfs >>    dst8u  =  { 113, 0 }          // scaleFactor = 0

// results for scaleFactor = 2
32f8s_Sfs >>    dstN8  =  { 32, -31 }         // scaleFactor = 2
32f8s_Sfs >>    dstZ8  =  { 31, -31 }         // scaleFactor = 2
32f8u_Sfs >>    dst8u  =  { 28, 0 }           // scaleFactor = 2
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsRoundModeNotSupportedErr`	Indicates an error when the specified rounding mode is not supported.

Join

Converts the floating-point data of several vectors to integer data, and stores the results in a single vector.

Syntax

```
IppStatus ippsJoin_32f16s_D2L(const Ipp32f** pSrc, int nChannels, int chanLen, Ipp16s*
    pDst);
```

Parameters

pSrc	Pointer to an array of pointers to the input vectors.
pDst	Pointer to the output vector.
nChannels	Number of input vectors.
chanLen	Number of elements in each input vector.

Description

The function `ippsJoin` is declared in the `ipps.h` file. This function converts floating-point data of *nChannels* input vectors stored in the array *pSrc* to integer data type and writes the results to the destination vector *pDst* in the following order: first element of first vector, first element of second vector, ..., first element of the last vector in the array; second element of first vector, second element of second vector, and so on.
The converted value is saturated if it exceeds the output data range.

Example 5-14 shows how to use the function `ippsJoin`.

Example 5-14 Using the Function `ippsJoin`

```
Ipp32f** pSrc;
Ipp16s pDst[8];
int nChannels = 2;
int chanLen = 4;
int bufSize;
float k = 1.0;

pSrc = (Ipp32f**) ippiMalloc_32f_C1 ( 2, 1, &bufSize );
for(int I = 0; I < 2; i++) {
    pSrc[i] = ( Ipp32f*) ippiMalloc_32f_C1 ( 4, 1, &bufSize );
    for(int j = 0; j < 4; j++)
        pSrc[i][j] = k++;
}

ippsJoin_32f16s_D2L ( (const Ipp32f**) pSrc, nChannels, chanLen,
(Ipp16s*) pDst );

Result:

1.0 2.0 3.0 4.0
5.0 6.0 7.0 8.0         pSrc

1 5 2 6 3 7 4 8       pDst
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error condition if *nChannels* or *chanLen* is less than or equal to 0.

JoinScaled

Converts with scaling the floating-point data of several vectors to integer data and stores the results in a single vector.

Syntax

```
IppStatus ippsJoinScaled_32f16s_D2L(const Ipp32f** pSrc, int nChannels, int chanLen,
    Ipp16s* pDst);
IppStatus ippsJoinScaled_32f24s_D2L(const Ipp32f** pSrc, int nChannels, int chanLen,
    Ipp8u* pDst);
```

Parameters

pSrc	Pointer to an array of pointers to the input vectors.
pDst	Pointer to the output vector.
nChannels	Number of input vectors.
chanLen	Number of elements in each input vector.

Description

The function `ippsJoinScaled` is declared in the `ipps.h` file. This function converts floating-point data of *nChannels* input vectors stored in the array *pSrc* to integer data type with scaling. Input data must be in the range [-1.0, 1.0] that is mapped onto output data type range [*dst_Min*..*dst_Max*] (see Data Ranges in Chapter 2 for more information on data types and ranges). If the input data exceed the range [-1.0, 1.0], they are saturated.
The function uses the following formula for scaling:

$$pDst[n] = dst_Min + k * [pSrc[n] - (-1)]$$

where $k = (dst_Max - dst_Min) / (1 - (-1))$

Mapped values are rounded to the nearest integer.

The function `ippsJoinScaled` writes the results to the destination vector *pDst* in the following order: first element of first vector, first element of second vector, ..., first element of the last vector in the array; second element of first vector, second element of second vector, and so on.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates a then error condition if at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error condition if *nChannels* or *chanLen* is less than or equal to 0.

SplitScaled

Converts the integer data of a vector to floating-point data with scaling, and stores the result in several vectors.

Syntax

```
IppStatus ippsSplitScaled_16s32f_D2L(const Ipp16s* pSrc, Ipp32f** pDst, int
    nChannels, int chanLen);
IppStatus ippsSplitScaled_24s32f_D2L(const Ipp8u* pSrc, Ipp32f** pDst, int
    nChannels, int chanLen);
```

Parameters

pSrc	Pointer to the input vector.
pDst	Pointer to an array of pointers to the output vectors.
nChannels	Number of output vectors.
chanLen	Number of elements in each output vector.

Description

The function ippsSplitScaled is declared in the ipps.h file. This function converts integer data of the input vector *pSrc* to the floating-point data type with scaling. The whole range of the input data [*src_Min*..*src_Max*] (see Data Ranges in Chapter 2 for more information on data types and ranges) is mapped onto the output data range [-1.0, 1.0] The function uses the following formula for scaling:

$$pDst[n] = (-1) + k * (pSrc[n] - src_Min)$$

where $k = (1 - (-1)) / (src_Max - src_Min)$

The function `ippsSplitScaled` writes the results to the *nChannels* destination vectors *pDst* in the following order: first element of the input vector becomes the first element of the first output vector, second element of the input vector becomes the first element of the second output vector, and so on.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates a then error condition if at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error condition if *nChannels* or *chanLen* is less than or equal to 0.

Conj

Stores the complex conjugate values of a vector in a second vector or in-place.

Syntax

```
IppStatus ippsConj_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsConj_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsConj_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsConj_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsConj_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsConj_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsConj` is declared in the `ipps.h` file. This function stores in *pDst* the element-wise conjugation of the complex vector *pSrc*. The element-wise conjugation of the vector is defined as follows:

pDst[n].re = pSrc[n].re

pDst[n].im = - pSrc[n].im

The in-place flavors of `ippsConj` store in *pSrcDst* the element-wise conjugation of the complex vector *pSrcDst*.

The element-wise conjugation of the vector is defined as follows:

pSrcDst[n].re = pSrcDst[n].re

pSrcDst[n].im = - pSrcDst[n].im

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

ConjFlip

Computes the complex conjugate of a vector and stores the result in reverse order.

Syntax

```
IppStatus ippsConjFlip_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsConjFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsConjFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.

Description

The function `ippsConjFlip` is declared in the `ipps.h` file. This function computes the conjugate of the vector *pSrc* and stores the result, in reverse order, in *pDst*.

The complex conjugate, stored in reverse order, is defined as follows:

pDst[*n*] = *conj*(*pSrc*[*len* - *n* - 1])

Note that if *pSrc* and *pDst* overlap in memory, the function returns unpredictable results.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Magnitude

Computes the magnitudes of the elements of a complex vector.

Syntax

```
IppStatus ippsMagnitude_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
    pDst, int len);
IppStatus ippsMagnitude_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
    pDst, int len);
IppStatus ippsMagnitude_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsMagnitude_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp32f* pDst, int len);
IppStatus ippsMagnitude_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsMagnitude_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsMagnitude_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len,
    int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the vector with the real parts of complex elements.
pSrcIm	Pointer to the vector with the imaginary parts of complex elements.
pDst	Pointer to the destination vector.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMagnitude` is declared in the `ipps.h` file. The complex flavor of this function computes the element-wise magnitude of the complex vector *pSrc* and stores the result in *pDst*. The element-wise magnitude is defined by the formula:

$$magn[n] = \sqrt{pSrc[n].re^2 + pSrc[n].im^2}$$

The real flavor of the function `ippsMagnitude` computes the element-wise magnitude of the complex vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the result in *pDst*. The element-wise magnitude is defined by the formula:

$$magn[n] = \sqrt{pSrcRe[n]^2 + pSrcIm[n]^2}$$

Example 5-15 verifies the identity $\sin^2 x + \cos^2 x = 1$.

Example 5-15 Using the Function `ippsMagnitude`

```
void magn(void) {
      Ipp64f x[6], magn[4];
      int n;
      for (n = 0; n<6; ++n) x[n] = sin(IPP_2PI * n / 8);
      ippsMagnitude_64f(x, x+2, magn, 4);
      printf_64f("magn =", magn, 4, ippStsNoErr);
}
Output:
    magn =  1.000000 1.000000 1.000000 1.000000
Matlab* Analog:
    >> n = 0:9; x = sin(2*pi*n/8); z = [x(1:8)+j*x(3:10)]; abs(z(1:4))
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MagSquared

Computes the squared magnitudes of the elements of a complex vector.

Syntax

```
IppStatus ippsMagSquared_32sc32s_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len,
    int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMagSquared` is declared in the `ipps.h` file. This function computes the element-wise squared magnitude of the complex vector *pSrc* and stores the result in *pDst*. The element-wise squared magnitude is defined by the formula:

$$magn[n] = pSrc[n].re^2 + pSrc[n].im^2$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Phase

Computes the phase angles of elements of a complex vector.

Syntax

```
IppStatus ippsPhase_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPhase_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsPhase_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len, int
    scaleFactor);
IppStatus ippsPhase_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f* pDst,
    int len);
IppStatus ippsPhase_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f* pDst,
    int len);
IppStatus ippsPhase_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp32f* pDst,
    int len);
IppStatus ippsPhase_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16s*
    pDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components.
pSrcIm	Pointer to the source vector which stores the imaginary components.
pDst	Pointer to the vector which stores the phase (angle) components of the elements in radians. Phase values are in the range $(-\pi,\pi]$.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsPhase` is declared in the `ipps.h` file. This function returns the phase angles of elements of the complex input vector *pSrc*, or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the result in the vector *pDst*. Phase values are returned in radians and are in the range (-π, π].

Example 5-16 shows how to call the function `ippsPhase_32f`.

Example 5-16 Using the `ippsPhase` Function

```
void func_phase()
{
    Ipp32f re[4]= {0.0, 1.0, 0.0, 1.0};
    Ipp32f im[4]= {0.0, 0.0, 1.0, 1.0};
    Ipp32f pDst[4];

    ippsPhase_32f(re,im,pDst, 4);
}
Result
pDst -> {0.0, 0.0, 1.6 0.8)
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

PowerSpectr

Computes the power spectrum of a complex vector.

Syntax

```
IppStatus ippsPowerSpectr_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPowerSpectr_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPowerSpectr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsPowerSpectr_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsPowerSpectr_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
    pDst, int len);
IppStatus ippsPowerSpectr_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
    pDst, int len);
IppStatus ippsPowerSpectr_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsPowerSpectr_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp32f*
    pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components.
pSrcIm	Pointer to the source vector which stores the imaginary components.
pDst	Pointer to the vector which stores the spectrum components of the elements.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsPowerSpectr` is declared in the `ipps.h` file. This function returns the power spectrum of the complex input vector *pSrc*, or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the results in the vector *pDst*. The power spectrum elements are squares of the magnitudes of the complex input vector elements:

$pDst[n] = (pSrc[n].re)^2 + (pSrc[n].im)^2$, or
$pDst[n] = (pSrcRe[n])^2 + (pSrcIm[n])^2$.

To compute magnitudes, use the function `ippsMagnitude`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Real

Returns the real part of a complex vector in a second vector.

Syntax

```
IppStatus ippsReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, int len);
IppStatus ippsReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, int len);
IppStatus ippsReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, int len);
```

Parameters

pSrc	Pointer to the complex source vector.
pDstRe	Pointer to the destination vector with real parts.
len	Number of elements in the vector.

Description

The function `ippsReal` is declared in the `ipps.h` file. This function returns the real part of the complex vector *pSrc* in the vector *pDstRe*.

Example 5-17 shows how to call the function `ippsReal_32fc`.

Example 5-17 Using the Function `ippsReal`

```
void func_real()
{
    Ipp32fc pSrc[5] = {{1.2, 3.2}, {0.0, 0.3}, {4.3, 6.7}, {4.2,
0.0},{1.2, 1.2}};
    Ipp32f pDstRe[5];

    ippsReal_32fc(pSrc,pDstRe,5);
}
Result
pDestRe ->(1.2, 0.0, 4.3, 4.2, 1.2)
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDstRe* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Imag

Returns the imaginary part of a complex vector in a second vector.

Syntax

```
IppStatus ippsImag_16sc(const Ipp16sc* pSrc, Ipp16s* pDstIm, int len);
IppStatus ippsImag_32fc(const Ipp32fc* pSrc, Ipp32f* pDstIm, int len);
IppStatus ippsImag_64fc(const Ipp64fc* pSrc, Ipp64f* pDstIm, int len);
```

Parameters

pSrc	Pointer to the complex source vector.
pDstIm	Pointer to the destination vector with imaginary parts.
len	Number of elements in the vector.

Description

The function `ippsImag` is declared in the `ipps.h` file. This function returns the imaginary part of a complex vector *pSrc* in the vector *pDstIm*.

Example 5-18 shows how to call the function `ippsImag_32fc`.

Example 5-18 Using the Function `ippsImag`

```
void func_imag()
{
    Ipp32fc pSrc[5] = {{1.2, 3.2}, {0.0, 0.3}, {4.3, 6.7}, {4.2, 0.0},
{1.2, 1.2}};
    Ipp32f pDstIm[5];

    ippsImag_32fc(pSrc, pDstIm, 5);
}
Result
pDstIm ->(3.2, 0.3, 6.7, 0.0, 1.2)
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDstIm* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

RealToCplx

Returns a complex vector constructed from the real and imaginary parts of two real vectors.

Syntax

```
IppStatus ippsRealToCplx_16s(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16sc*
    pDst, int len);
IppStatus ippsRealToCplx_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32fc*
    pDst, int len);
IppStatus ippsRealToCplx_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64fc*
    pDst, int len);
```

Parameters

pSrcRe	Pointer to the vector with real parts of complex elements.
pSrcIm	Pointer to the vector with imaginary parts of complex elements.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.

Description

The function `ippsRealToCplx` is declared in the `ipps.h` file. This function returns a complex vector *pDst* constructed from the real and imaginary parts of the input vectors *pSrcRe* and *pSrcIm*.

If *pSrcRe* is `NULL`, the real component of the vector is set to zero.

If *pSrcIm* is `NULL`, the imaginary component of the vector is set to zero.

Note that the pointers can not be both `NULL`.

Example 5-19 shows how to call the function `ippsRealToCplx_32f`.

Example 5-19 Using the Function `ippsRealToCplx`

```
void func_realtocplx()
{
    Ipp32f pSrcRe[6] = {12.2, -2.1, 4.3, 1.1, 2.2, 0.0};
    Ipp32f pSrcIm[6] = {-9.3, 2.3, -5.2, 0.0, -1.0, -1.2};
    Ipp32fc pDst[6];

    ippsRealToCplx_32f(pSrcRe, pSrcIm, pDst, 6);
```

Example 5-19 Using the Function ippsRealToCplx

```
}
Result
pDst ->{(12.2, -9.3), (-2.1, 2.3), (4.3, -5.2), (1.1, 0.0), (2.2, -1.0),
(0.0, 10.0}
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* pointer is NULL. The pointer *pSrcRe* or *pSrcIm* can be NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

CplxToReal

Returns the real and imaginary parts of a complex vector in two respective vectors.

Syntax

```
IppStatus ippsCplxToReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, Ipp16s* pDstIm,
    int len);
IppStatus ippsCplxToReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, Ipp32f* pDstIm,
    int len);
IppStatus ippsCplxToReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, Ipp64f* pDstIm,
    int len);
```

Parameters

pSrc	Pointer to the complex vector *pSrc*.
pDstRe	Pointer to the output vector with real parts.
pDstIm	Pointer to the output vector with imaginary parts.
len	Number of elements in the vector.

Description

The function ippsCplxToReal is declared in the ipps.h file. This function returns the real and imaginary parts of a complex vector *pSrc* in two vectors *pDstRe* and *pDstIm*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the data vector pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Threshold

Performs the threshold operation on the elements of a vector by limiting the element values by *level*.

Syntax

```
IppStatus ippsThreshold_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
    level, IppCmpOp relOp);
IppStatus ippsThreshold_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    level, IppCmpOp relOp);
IppStatus ippsThreshold_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f
    level, IppCmpOp relOp);
IppStatus ippsThreshold_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    Ipp16s level, IppCmpOp relOp);
IppStatus ippsThreshold_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level, IppCmpOp
    relOp);
IppStatus ippsThreshold_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level, IppCmpOp
    relOp);
IppStatus ippsThreshold_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level, IppCmpOp
    relOp);
IppStatus ippsThreshold_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level,
    IppCmpOp relOp);
IppStatus ippsThreshold_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level,
    IppCmpOp relOp);
```

```
IppStatus ippsThreshold_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level,
    IppCmpOp relOp);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.
relOp	Values of this argument specify which relational operator to use and whether *level* is an upper or lower bound for the input. The *relOp* must have one of the following values:
	ippCmpLess — Specifies the “less than” operator and *level* is a lower bound.
	ippCmpGreater — Specifies the “greater than” operator and *level* is an upper bound.

Description

The function `ippsThreshold` is declared in the `ipps.h` file. This function performs the threshold operation on the vector *pSrc* by limiting each element by the threshold value *level*. Function operation is similar to that of the functions `ippsThreshold_LT`, `ippsThreshold_GT` but its interface contains the *relop* argument that specifies the type of the comparison operation to perform.

The in-place flavors of `ippsThreshold` perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

The *relOp* argument specifies which relational operator to use: “greater than” or “less than,” and determines whether *level* is an upper or lower bound for the input, respectively. The formula for `ippsThreshold` called with the *ippCmpLess* flag is:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold`, the *level* argument is always real. The formula for complex `ippsThreshold` called with the *ippCmpLess* flag is:

$$pDst[n] = \begin{cases} \dfrac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the function `ippsThreshold`. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the "less than" operation (with the *ippCmpLess* flag) the coordinates are rounded to the infinity (`+Inf` for positive coordinates, and `-Inf` for negative), and for the "greater than" operation (with the *ippCmpGreater* flag) the coordinates are rounded to 0. See Example 5-20 for more information about using the "complex" function `ippsThreshold_16sc_I`.

Example 5-21 shows how to use the "real" function `ippsTreshold_16s_I`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsThreshNegLevelErr`	Indicates an error when *level* for the complex version is negative (see appendix A "Handling of Special Cases" for more information).

Example 5-20 Using the Complex Version of the `ippsThreshold` Function

```
IppStatus cmplx_threshold(void) {
     Ipp16sc x[4] = {{2,3}, {3,3}, {4,3}, {4,2}};
     /// level is near to the point {2,3} = 3.6
     /// the point {2,3} is to be replaced
     /// the computed coordinates are {2.2188,3.3282}
     /// the point used is {3,4};
     /// notice that it is the point with the phase,
     /// nearest to the source
     IppStatus st = ippsThreshold_16sc_I(x, 4, 4, ippCmpLess);
     printf_16sc("complex threshold result =", x, 4, st);
     return st;
```

Example 5-20 Using the Complex Version of the `ippsThreshold` Function

```
}
Output:
    complex threshold result = {3, 4} {3, 3} {4, 3} {4, 2}
```

Example 5-21 Using the Real Version of the `ippsThreshold` Function

```
IppStatus threshold( void ) {
     Ipp16s x[4] = { -1, 0, 2, 3 };
     IppStatus st = ippsThreshold_16s_I( x, 4, 2, ippCmpLess );
     printf_16s("threshold result =", x, 4, st );
     return st;
}
Output:
    threshold result =  2 2 2 3
```

Threshold_LT, Threshold_GT

Performs the threshold operation on the elements of a vector by limiting the element values by `level`.

Syntax

```
IppStatus ippsThreshold_LT_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level);
IppStatus ippsThreshold_LT_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    Ipp32s level);
IppStatus ippsThreshold_LT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_LT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_LT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_LT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_LT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    Ipp16s level);
IppStatus ippsThreshold_LT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
```

```
IppStatus ippsThreshold_LT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level);
IppStatus ippsThreshold_GT_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    Ipp32s level);
IppStatus ippsThreshold_GT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_GT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_GT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_GT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_GT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    Ipp16s level);
IppStatus ippsThreshold_GT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_GT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.

Description

The functions ippsThreshold_LT, ippsThreshold_GT are declared in the ipps.h file. They implement thresholding of the vector *pSrc* by limiting each element by the threshold value *level*. These functions perform the similar operation to the ippsThreshold function but are designed for the fixed type of the compare operation to use: ippsThreshold_LT is for the "less than" comparison, while ippsThreshold_GT is for the "greater than" comparison.

The in-place flavors perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

ippsThreshold_LT. The ippsThreshold_LT function performs the operation “less than”, and *level* is a lower bound for the input. The formula for ippsThreshold_LT is the following:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_LT, the *level* argument is always real.

The formula for complex ippsThreshold_LT is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_GT. The function ippsThreshold_GT performs the operation “greater than” and *level* is an upper bound for the input.

The formula for ippsThreshold_GT is the following:

$$pDst[n] = \begin{cases} level, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_GT, the *level* argument is always real.

The formula for complex ippsThreshold_GT is:

$$pDst[n] = \begin{cases} \dfrac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the threshold functions. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the "less than" operation (the `ippsThreshold_LT` function) the coordinates are rounded to the infinity (`+Inf` for positive coordinates, and `-Inf` for negative), and for the "greater than" operation (the `ippsThreshold_GT` function) the coordinates are rounded to 0.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0
`ippStsThreshNegLevelErr`	Indicates an error when *level* for the complex version is negative (see appendix A "Handling of Special Cases" for more information).

Threshold_LTAbs, Threshold_GTAbs

Performs the threshold operation on the absolute values of elements of a vector.

Syntax

```
IppStatus ippsThreshold_LTAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level);
```

```
IppStatus ippsThreshold_LTAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    Ipp32s level);
IppStatus ippsThreshold_LTAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_LTAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_LTAbs_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_LTAbs_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LTAbs_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTAbs_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);

IppStatus ippsThreshold_GTAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level);
IppStatus ippsThreshold_GTAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    Ipp32s level);
IppStatus ippsThreshold_GTAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_GTAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_GTAbs_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GTAbs_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_GTAbs_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GTAbs_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of source vector.This argument can not be negative.

Description

The functions `ippsThreshold_LTAbs`, `ippsThreshold_GTAbs` are declared in the `ipps.h` file. They implement thresholding of the vector *pSrc* by limiting absolute value of each element by the threshold value *level*. These functions perform the compare operation of the fixed type: `ippsThreshold_LTAbs` is for the "less than" comparison, while `ippsThreshold_GTAbs` is for the "greater than" comparison. Elements of the result vector *pDst* have the same sign that the source elements.

The in-place flavors perform the threshold operation on the vector *pSrcDst*.

ippsThreshold_LTAbs. The `ippsThreshold_LTAbs` function performs the operation “less than”, and *level* is a lower bound for the input. The formula for `ippsThreshold_LTAbs` is the following:

$$pDst[n] = \begin{cases} level \text{ if } abs(pSrc[n]) < level, & pSrc[n] \geq 0 \\ -level \text{ if } abs(pSrc[n]) < level, & pSrc[n] < 0 \\ pSrc[n], \quad otherwise \end{cases}$$

ippsThreshold_GTAbs. The function `ippsThreshold_GTAbs` performs the operation “greater than” and *level* is an upper bound for the input.The formula for `ippsThreshold_GTAbs` is the following:

$$pDst[n] = \begin{cases} level \text{ if } abs(pSrc[n]) > level, & pSrc[n] \geq 0 \\ -level \text{ if } abs(pSrc[n]) > level, & pSrc[n] < 0 \\ pSrc[n], \quad otherwise \end{cases}$$

Example 5-22 shows how to use the function `ippsThreshold_LTAbs_64f_I`.

Example 5-22 Using the `ippThreshold_LTAbs` function

```
IppStatus thresholdLTAbs( void ) {
    Ipp64f      vec[7] = { -4.01, -4.0, -3.9, 0.0, 2.5, 4.0, 4.5 };
    Ipp64f      level = 4.0;
    IppStatus   st = ippsThreshold_LTAbs_64f_I( vec, 7, level );
    printf("threshold abs   =   %f  %f  %f  %f\n",
        vec[0],vec[1],vec[2],vec[3]);
    printf("                    %f   %f   %f\n",
        vec[4],vec[5],vec[6]);
    return st;
}
Output:
threshold abs = -4.010000 -4.000000 -4.000000 4.000000
                 4.000000 4.000000 4.500000
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error if *len* is less than or equal to 0.
ippStsThreshNegLevelErr	Indicates an error if *level* is negative.

Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal

Performs the threshold operation on the elements of a vector by limiting the element values by level.

Syntax

```
IppStatus ippsThreshold_LTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_LTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_LTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level,
    Ipp16s value);
IppStatus ippsThreshold_LTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level,
    Ipp32f value);
IppStatus ippsThreshold_LTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level,
    Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level,
    Ipp16sc value);
```

```
IppStatus ippsThreshold_LTVal_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level,
    Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level,
    Ipp64fc value);
IppStatus ippsThreshold_GTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_GTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_GTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_GTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_GTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_GTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_GTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level,
    Ipp16s value);
IppStatus ippsThreshold_GTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level,
    Ipp32f value);
IppStatus ippsThreshold_GTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level,
    Ipp64f value);
IppStatus ippsThreshold_GTVal_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level,
    Ipp16sc value);
IppStatus ippsThreshold_GTVal_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level,
    Ipp32fc value);
IppStatus ippsThreshold_GTVal_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level,
    Ipp64fc value);
IppStatus ippsThreshold_LTValGTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);
IppStatus ippsThreshold_LTValGTVal_32s(const Ipp32s* pSrc, Ipp32s* pDst, int
    len, Ipp32s levelLT, Ipp32s valueLT, Ipp32s levelGT, Ipp32s valueGT);
IppStatus ippsThreshold_LTValGTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len, Ipp32f levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);
IppStatus ippsThreshold_LTValGTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
    len, Ipp64f levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);
IppStatus ippsThreshold_LTValGTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s
    levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);
```

```
IppStatus ippsThreshold_LTValGTVal_32s_I(Ipp32s* pSrcDst, int len, Ipp32s
    levelLT, Ipp32s valueLT, Ipp32s levelGT, Ipp32s valueGT);
IppStatus ippsThreshold_LTValGTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f
    levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);
IppStatus ippsThreshold_LTValGTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f
    levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.
levelLT	Low bound used to limit each element of *pSrc* or *pSrcDst* for the `ippsThreshold_LTValGTVal` function.
levelGT	Upper bound used to limit each element of *pSrc* or *pSrcDst* for the `ippsThreshold_LTValGTVal` function.
value	Value to be assigned to vector elements which are “less than” or “greater than” *level*.
valueLT	Value to be assigned to vector elements which are less than *levelLT* for the `ippsThreshold_LTValGTVal` function.
valueGT	Value to be assigned to vector elements which are greater than *levelGT* for the `ippsThreshold_LTValGTVal` function.

Description

These functions are declared in the `ipps.h` file. They perform the threshold operation on the vector *pSrc* by limiting each element by the threshold value.

The in-place flavors of the function perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value.

ippsThreshold_LTVal. The `ippsThreshold_LTVal` function performs the operation “less than” and *level* is a lower bound for the input. The vector elements less than *level* are set to *value*.

The formula for `ippsThreshold_LTVal` is:

$$pDst[n] = \begin{cases} value, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold_LTVal`, the *level* argument is always real.

The formula for complex `ippsThreshold_LTVal` is:

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

`ippsThreshold_GTVal`. The `ippsThreshold_GTVal` function performs the operation "greater than" and *level* is an upper bound for the input. The vector elements greater than *level* are set to *value*.

The formula for `ippsThreshold_GtVal` is:

$$pDst[n] = \begin{cases} value, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold_GTVal`, the *level* argument is always real.

The formula for complex `ippsThreshold_GTVal` is:

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

`ippsThreshold_LTValGTVal`. The `ippsThreshold_LTValGTVal` function checks both the "less than" and "greater than" conditions. Argument *levelLT* is a lower bound and argument *levelGT* is an upper bound for the input. The source vector elements less than *levelLT* are set to *valueLT*, and the source vector elements greater than *levelGT* are set to *valueGT*. The value of *levelLT* should be less than or equal to *levelGT*.

The formula for `ippsThreshold_LTValGTVal` is:

$$pDst[n] = \begin{cases} valueLT, & pSrc[n] < levelLT \\ pSrc[n], & levelLT \le pSrc[n] \le levelGT \\ valueGT, & pSrc[n] > levelGT \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represent a magnitude.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsThresholdErr	Indicates an error when *levelLT* is greater than *levelGT*.
ippStsThreshNegLevelErr	Indicates an error when *level* for the complex version is negative (see appendix A "Handling of Special Cases" for more information).

Threshold_LTInv

Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.

Syntax

```
IppStatus ippsThreshold_LTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_LTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level);
IppStatus ippsThreshold_LTInv_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real and positive.

Description

The function `ippsThreshold_LTInv` is declared in the `ipps.h` file. This function computes the inverse of elements of the vector *pSrc* and stores the result in *pDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The in-place flavors of `ippsThreshold_LTInv` compute the inverse of elements of the vector *pSrcDst* and store the result in *pSrcDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The threshold operation is performed to avoid division by zero. Since *level* represents a magnitude, it is always real and must be positive. The formula for `ippsThreshold_LTInv` is the following:

$$pDst[n] = \begin{cases} \frac{1}{level}, & abs(pSrc[n]) = 0 \\ \frac{abs(pSrc[n])}{pSrc[n] \cdot level}, & 0 < abs(pSrc[n]) < level \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

If the function encounters zero-valued vector elements and *level* is also 0 (see appendix A "Handling of Special Cases"), the output value is set to `Inf` (infinity), but operation execution is not aborted:

$$pDst[n] = \begin{cases} Inf, & pSrc[n] = 0 \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

Example 5-23 shows how to use the function `ippsThreshold_LTInv_32f_I`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsThreshNegLevelErr	Indicates an error when *level* is negative.
ippStsInvZero	Indicates a warning when *level* and a vector element are equal to 0. Operation execution is not aborted. The value of the destination vector element is Inf.

Example 5-23 Using the ippsThreshold_LTInv Function

```
IppStatus invThreshold(void) {
     Ipp32f x[4] = {-1, 0, 2, 3};
     IppStatus st = ippsThreshold_LTInv_32f_I(x, 4, 0);
     printf_32f("inv threshold =", x, 4, st);
     return st;
}
Output:
    -- warning 4, INF result. Zero value met by invThreshold with zero
level
    inv threshold = -1.000000 1.#INF00 0.500000 0.333333
```

CartToPolar

Converts the elements of a complex vector to polar coordinate form.

Syntax

```
IppStatus ippsCartToPolar_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstMagn, Ipp32f* pDstPhase, int len);
IppStatus ippsCartToPolar_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstMagn, Ipp64f* pDstPhase, int len);
IppStatus ippsCartToPolar_32fc(const Ipp32fc* pSrc, Ipp32f* pDstMagn, Ipp32f*
    pDstPhase, int len);
IppStatus ippsCartToPolar_64fc(const Ipp64fc* pSrc, Ipp64f* pDstMagn, Ipp64f*
    pDstPhase, int len);
IppStatus ippsCartToPolar_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDstMagn,
    Ipp16s* pDstPhase, int len, int magnScaleFactor, int phaseScaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components of Cartesian X,Y pairs.

pSrcIm	Pointer to the source vector which stores the imaginary components of Cartesian X,Y pairs.
pDstMagn	Pointer to the vector which stores the magnitude (radius) component of the elements of the vector *pSrc*.
pDstPhase	Pointer to the vector which stores the phase (angle) component of the elements of the vector *pSrc* in radians. Phase values are in the range (-π,π].
len	Number of elements in the vector.
magnScaleFactor	Integer scale factor for the magnitude component. Refer to Integer Scaling in Chapter 2.
phaseScaleFactor	Integer scale factor for the phase component. Refer to Integer Scaling.

Description

The function `ippsCartToPolar` is declared in the `ipps.h` file. This function converts the elements of a complex input vector *pSrc* or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, to polar coordinate form, and stores the magnitude (radius) component of each element in the vector *pDstMagn* and the phase (angle) component of each element in the vector *pDstPhase*.

Example 5-24 verifies that points are lying in the unit radius circle.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 5-24 Using the `ippsCartToPolar` Function

```
IppStatus cart2polar( void ) {
     Ipp64f cart[6], magn[4], phase[4];
     int n;
     for (n=0; n<6; ++n) cart[n] = sin(IPP_2PI * n / 8);
     IppStatus st = ippsCartToPolar_64f( cart, cart+2, magn, phase, 4 );
     printf_64f( "magn =", magn, 4, st );
     return st;
}

Output:
    magn =  1.000000 1.000000 1.000000 1.000000
```

PolarToCart

Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.

Syntax

```
IppStatus ippsPolarToCart_32f(const Ipp32f* pSrcMagn, const Ipp32f* pSrcPhase,
    Ipp32f* pDstRe, Ipp32f* pDstIm, int len);
IppStatus ippsPolarToCart_64f(const Ipp64f* pSrcMagn, const Ipp64f* pSrcPhase,
    Ipp64f* pDstRe, Ipp64f* pDstIm, int len);
IppStatus ippsPolarToCart_32fc(const Ipp32f* pSrcMagn, const Ipp32f* pSrcPhase,
    Ipp32fc* pDst, int len);
IppStatus ippsPolarToCart_64fc(const Ipp64f* pSrcMagn, const Ipp64f* pSrcPhase,
    Ipp64fc* pDst, int len);
IppStatus ippsPolarToCart_16sc(const Ipp16s* pSrcMagn, const Ipp16s* pSrcPhase,
    int phaseFixedPoint, Ipp16sc* pDst, int len);
IppStatus ippsPolarToCart_32sc(const Ipp32s* pSrcMagn, const Ipp32s* pSrcPhase,
    int phaseFixedPoint, Ipp32sc* pDst, int len);
IppStatus ippsPolarToCart_16sc_Sfs(const Ipp16s* pSrcMagn, const Ipp16s*
    pSrcPhase, Ipp16sc* pDst, int len, int magnScaleFactor, int
    phaseScaleFactor);
```

Parameters

pSrcMagn	Pointer to the source vector which stores the magnitude (radius) components of the elements in polar coordinate form.
pSrcPhase	Pointer to the vector which stores the phase (angle) components of the elements in polar coordinate form in radians. Phase values are in the range (-π,π].
pDst	Pointer to the resulting vector which stores the complex pairs in Cartesian coordinates (X + iY).
pDstRe	Pointer to the resulting vector which stores the real components of Cartesian X,Y pairs.
pDstIm	Pointer to the resulting vector which stores the imaginary components of Cartesian X,Y pairs.
len	Number of elements in the vectors.
phaseFixedPoint	Specified the position of the decimal fixed point for the phase.

magnScaleFactor	Integer scale factor for the magnitude component. Refer to Integer Scaling in Chapter 2.
phaseScaleFactor	Integer scale factor for the phase component. Refer to Integer Scaling.

Description

The function `ippsPolarToCart` is declared in the `ipps.h` file. This function converts the polar form magnitude/phase pairs stored in the input vectors *pSrcMagn* and *pSrcPhase* into a complex vector and stores the results in the vector *pDst*, or stores the real components of the result in the vector *pDstRe* and the imaginary components in the vector *pDstIm*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MaxOrder

Computes the maximum order of a vector.

Syntax

```
IppStatus ippsMaxOrder_16s(const Ipp16s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32s(const Ipp32s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32f(const Ipp32f* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_64f(const Ipp64f* pSrc, int len, int* pOrder);
```

Parameters

pSrc	Pointer to the source vector.
len	Number of elements in the vector.
pOrder	Pointer to the result value.

Description

The function `ippsMaxOrder` is declared in the `ipps.h` file. This function finds the maximum binary number in elements of the exponent vector *pSrc*, and stores the result in *pOrder*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pOrder* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsNanArg`	Indicates a warning when `NaN` is encountered in the input data vector.

Preemphasize

Computes preemphasis of a single precision real signal in-place.

Syntax

```
IppStatus ippsPreemphasize_16s(Ipp16s* pSrcDst, int len, Ipp32f val);
IppStatus ippsPreemphasize_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
```

Parameters

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
val	Multiplier factor used in the difference signal preemphasis equation.

Description

The in-place function `ippsPreemphasize` is declared in the `ipps.h` file. This function computes preemphasis of a real signal *pSrcDst*. The computation is performed according to the difference signal preemphasis equation:

$$y(n) = x(n) - val \cdot x(n - 1),$$

where *y(n)* is the preemphasized output, *x(n)* is the input, and *val* is the multiplier factor.

Note that usually *val* = 0.95 for speech signals.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Flip

Reverses the order of elements in a vector.

Syntax

```
IppStatus ippsFlip_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsFlip_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsFlip_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsFlip_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsFlip_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsFlip_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsFlip_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsFlip_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsFlip_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsFlip_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsFlip` is declared in the `ipps.h` file. This function stores the elements of a source vector *pSrc* to a destination vector *pDst* in reverse order according to the following formula:

pDst[*n*] = *pSrc*[*len*-*n*-1], *n*=0..*len*-1

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

FindNearestOne

Finds an element of the table which is closest to the specified value.

Syntax

```
IppStatus ippsFindNearestOne_16u(Ipp16u inpVal, Ipp16u* pOutVal, int* pOutIndex,
    const Ipp16u *pTable, int tblLen);
```

Parameters

inpVal	Reference value.
pOutVal	Pointer to the output value.
pOutIndx	Pointer to the output index.
pTable	Pointer to the table for searching.
tblLen	Number of elements in the table.

Description

The function `ippsFindNearestOne` is declared in the `ipps.h` file. This function searches through the table *pTable* for an element which is closest to the specified reference value *inpVal*. The resulting element and its index are stored in *pOutVal* and *pOutIndex*, respectively.

The table elements should satisfy the condition *pTable*[*i*] ≤*pTable*[*i+1*].
The function uses the following distance criterion for determining the closest element: min(|*inpVal*-*pTable*[*i*]|).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *tblLen* is less than or equal to 0.

FindNearest

Finds table elements that are closest to the elements of the specified vector.

Syntax

```
IppStatusippsFindNearest_16u(constIpp16u*pVals,Ipp16u*pOutVals,int*pOutIndexes,
    int len, const Ipp16u *pTable, int tblLen);
```

Parameters

pVals	Pointer to the vector containing reference values.
pOutVals	Pointer to the output vector.
pOutIndexes	Pointer to the array that stores output indexes.
len	Number of elements in the input vector.
pTable	Pointer to the table for searching.
tblLen	Number of elements in the table.

Description

The function ippsFindNearest is declared in the ipps.h file. This function searches through the table *pTable* for elements which are closest to the reference elements of the input vector *pVals*. The resulting elements and their indexes are stored in *pOutVals* and *pOutIndexes*, respectively.
The table elements should satisfy the condition *pTable*[*i*] ≤*pTable*[*i+1*].

The function uses the following distance criterion for determining the table element closest to *pVal*[*k*] :

min(|*pVals[k]*-*pTable*[i]|).

Example 5-23 shows how to use the function ippsFindNearest.

Example 5-25 Using the Function ippsFindNearest

```
Ipp16u pVals[] = { 12, 6, 78 };
Ipp16u pTable[] = { 1, 3, 5, 7, 18, 24, 35, 48 };
Ipp16u pOutVals[3];
int pOutIndexes[3];

ippsFindNearest_16u ( pVals, pOutVals, pOutIndexes, 3, pTable, 8 );

Result
pOutVals = { 7, 5, 48 }
pOutIndexes = { 3, 2, 7 }
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *tblLen* or *len* is less than or equal to 0.

Viterbi Decoder Functions

This section describes the functions that perform operations of Viterbi decoding in the V34 receiver. The encoding is performed in accordance with the algorithm that is described in the section 9.6.3 of the ITU-T Recommendation V.34 (see [ITUV34]).

GetVarPointDV

Fills the array with the information about points that are closest to the received point.

Syntax

```
IppStatus ippsGetVarPointDV_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, Ipp16sc*
    pVariantPoint, const Ipp8u* pLabel, int state);
```

Parameters

pSrc	Pointer to the reference point in format 9:7
pDst	Pointer to the closest to the reference point left and bottom complex point in format 9:7
pVariantPoint	Pointer to the array where the point information is stored
pLabel	Pointer to the table that stores the labels.
state	Number of states of a convolution coder.

Description

The function `ippsGetVarPointDV` is declared in the `ipps.h` file. This function fills the specified array *pVariantPoint* with the information about 2D complex points that are closest to the reference point *pSrc*. This information includes the corresponding labels from the offset table and calculated errors. The number of possible states may be 16, 32, and 64. The number of points in the array depends on the number of states: if *state* =16 , then 4 points will be referenced in the array, if *state* =32 or 64, then 8 points will be referenced.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.

CalcStatesDV

Calculates the states of the Viterbi decoder.

Syntax

```
IppStatus ippsCalcStatesDV_16sc(const Ipp16u* pathError, const Ipp8u* pNextState,
    Ipp16u* pBranchError, const Ipp16s* pCurrentSubsetPoint, Ipp16s* pPathTable,
    int state, int presentIndex);
```

Parameters

pPathError	Pointer to the table of path error metrics.
pNextState	Pointer to the next state table.
pBranchError	Pointer to the branch error table.

pCurrentSubsetPoint	Pointer to the current 4D subset.
pPathTable	Pointer to the Viterbi path table.
state	Number of states of a convolutional encoder.
presentIndex	Start index in Viterbi path table.

Description

The function `ippsCalcStatesDV` is declared in the `ipps.h` file. This function computes the possible states of the Viterbi decoder and fills the table of the accumulated errors *pPathError* and the working Viterbi path table *pPathTable*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.

BuildSymblTableDV4D

Fills the array with the information about possible 4D symbols.

Syntax

```
IppStatus ippsBuildSymblTableDV4D_16sc(const Ipp16sc* pVariantPoint, Ipp16sc*
    pCurrentSubsetPoint, int state, int bitInversion);
```

Parameters

pVariantPoint	Pointer to the array of possible 2D points.
pCurrentSubsetPoint	Pointer to the array with information about possible 4D symbols.
state	Number of states of a convolutional encoder.
bitInversion	Bit inversion.

Description

The function `ippsBuildSymblTableDV4D` is declared in the `ipps.h` file. This function fills the array *pCurrentSubsetPoint* with information about possible 4D symbols.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.

UpdatePathMetricsDV

Searches for the state with the minimum path metric.

Syntax

```
IppStatus ippsUpdatePathMetricsDV_16u(Ipp16u* pBranchError, Ipp16u* pMinPathError,
    Ipp8u* pMinSost, Ipp16u* pPathError, int state);
```

Parameters

pBranchError	Pointer to the branch error table.
pMinPathError	Pointer to the current minimum path error metric.
pMinSost	Pointer to the number of states with the minimum metric.
pPathError	Pointer to the table of accumulated path errors.
state	Number of states of a convolutional encoder.

Description

The function ippsBuildSymblTableDV4D is declared in the ipps.h file. This function searches for the state with the minimum path error metric and stores its number in the *pMinSost*. For all states, the minimum metric is subtracted from the path metric. The branch errors for all states are set to be infinitely large as is required for the next step of the Viterbi decoding.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.

Windowing Functions

This chapter describes several of the windowing functions commonly used in signal processing. A window is a mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

Understanding Window Functions

The Intel IPP provides the following functions to generate window samples:

- Bartlett windowing function
- Blackman family of windowing functions
- Hamming windowing function
- Hann windowing function
- Kaiser windowing function

These functions generate the window samples and multiply them into an existing signal. To obtain the window samples themselves, initialize the vector argument to the unity vector before calling the window function.

If you want to multiply different frames of a signal by the same window multiple times, it is better to first calculate the window by calling one of the windowing functions (`ippsWinHann`, for example) on a vector with all elements set to 1.0. Then use one of the vector multiplication functions (`ippsMul`, for example) to multiply the window into the signal each time a new set of input samples is available. This avoids repeatedly calculating the window samples. This is illustrated in Example 5-26.

Example 5-26 Window and FFT Many Frames of a Signal

```
void multiFrameWin( void ) {
     Ipp32f win[LEN], x[LEN], X[LEN];
     IppsFFTSpec_R_32f* ctx;
     ippsSet_32f( 1, win, LEN );
     ippsWinHann_32f_I( win, LEN );
     /// ... initialize FFT context
     while(1 ){
        /// ... get x signal
        ///
        ippsMul_32f_I( win, x, LEN );
        ippsFFTFwd_RToPack_32f( x, X, ctx, 0 );
     }
}
```

Related Topics

For more information on windowing, see: [Jack89], section 7.3, *Windows in Spectrum Analysis*; [Jack89], section 9.1, *Window-Function Technique*; and [Mit93], section 16-2, *Fourier Analysis of Finite-Time Signals*.
For more information on these references, see also the Bibliography at the end of this manual.

WinBartlett

Multiplies a vector by a Bartlett windowing function.

Syntax

```
IppStatus ippsWinBartlett_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBartlett_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBartlett_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBartlett_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinBartlett_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinBartlett_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinBartlett_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBartlett_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBartlett_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBartlett_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBartlett_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBartlett_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsWinBartlett` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Bartlett (triangle) window, and stores the result in *pDst*.

The in-place flavors of `ippsWinBartlett` multiply the *pSrcDst* by the Bartlett (triangle) window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window.

The Bartlett window is defined as follows:

$$w_{bartlett}(n) = \begin{cases} \frac{2n}{len-1}, & 0 \le n \le \frac{len-1}{2} \\ 2 - \frac{2n}{len-1}, & \frac{len-1}{2} < n \le len-1 \end{cases}$$

Example 5-27 shows how to use the function `ippsWinBartlett_32f_I`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than 3.

Example 5-27 Using the ippsWinBartlett Function

```
void bartlett(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBartlett_32f_I(x, 8);
      printf_32f("bartlett (half) =", x, 4, ippStsNoErr);
}
Output:
    bartlett (half) =  0.000000 0.285714 0.571429 0.857143
Matlab* Analog:
    >> b = bartlett(8); b(1:4)'
```

WinBlackman

Multiplies a vector by a Blackman windowing function.

Syntax

```
IppStatus ippsWinBlackmanQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
    alphaQ15);
IppStatus ippsWinBlackmanQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, int
    alphaQ15);
IppStatus ippsWinBlackman_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, float alpha);
IppStatus ippsWinBlackman_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, float
    alpha);
IppStatus ippsWinBlackman_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, float alpha);
IppStatus ippsWinBlackman_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, float
    alpha);
IppStatus ippsWinBlackman_64f(Ipp64f* pSrc, Ipp64f* pDst, int len, double alpha);
IppStatus ippsWinBlackman_64fc(Ipp64fc* pSrc, Ipp64fc* pDst, int len, double alpha);
IppStatus ippsWinBlackmanQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinBlackmanQ15_16s_ISfs(Ipp16s* pSrcDst, int len, int alphaQ15, int
    scaleFactor);
IppStatus ippsWinBlackmanQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinBlackman_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_32f_I(Ipp32f* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_64f_I(Ipp64f* pSrcDst, int len, double alpha);
IppStatus ippsWinBlackman_64fc_I(Ipp64fc* pSrcDst, int len, double alpha);
IppStatus ippsWinBlackmanStd_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBlackmanStd_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinBlackmanStd_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBlackmanStd_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinBlackmanStd_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBlackmanStd_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

```
IppStatus ippsWinBlackmanStd_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBlackmanOpt_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinBlackmanOpt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBlackmanOpt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinBlackmanOpt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBlackmanOpt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinBlackmanOpt_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
alpha	Adjustable parameter associated with the Blackman windowing equation.
alphaQ15	Scaled version of *alpha*. The *scaleFactor* value is 15.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The `ippsWinBlackman` family of functions are declared in the `ipps.h` file. These functions multiply the vector *pSrc* by the Blackman window, and store the result in *pDst*.

The in-place flavors of ippsWinBlackman multiply the vector *pSrcDst* by the Blackman window, and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The functions for the Blackman family of windows are defined below.

ippsWinBlackman. The function ippsWinBlackman allows the application to specify *alpha*. The Blackman window is defined as follows:

$$w_{blackman}(n) = \frac{alpha+1}{2} - 0.5\cos\left(\frac{2\pi n}{len-1}\right) - \frac{alpha}{2}\cos\left(\frac{4\pi n}{len-1}\right)$$

ippsWinBlackmanQ15. The function ippsWinBlackmanQ15 multiplies a vector by a Blackman window with *alphaQ15* scaled according to the factor 15.

ippsWinBlackmanStd. The standard Blackman window is provided by the function ippsWinBlackmanStd, which simply multiplies a vector by a Blackman window with the standard value of *alpha* shown below:

alpha = -0.16

ippsWinBlackmanOpt. The function ippsWinBlackmanOpt provides a modified window that has a 30 dB/octave roll-off by multiplying a vector by a Blackman window with the optimal value of *alpha* shown below:

$$alpha = -\frac{0.5}{1+\cos\frac{2\pi}{len-1}}$$

The minimum *len* is equal to 4. For large *len*, the optimal *alpha* converges asymptotically to the asymptotic *alpha*; the application can use the asymptotic value of *alpha* shown below:

alpha = -0.25

Example 5-28 shows how to use the function ippsWinBlackmanStd_32f_I

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 4 for the function ippsWinBlackmanOpt and less than 3 for all other functions of the family.

Example 5-28 Using the ippsWinBlackmanStd Function

```
void blackman(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBlackmanStd_32f_I(x, 8);
      printf_32f("blackman (half) =", x, 4, ippStsNoErr);
}
Output:
    blackman(half) =  0.000000 0.090453 0.459183 0.920364
Matlab* Analog:
    >> b = blackman(8)'; b(1:4)
```

WinHamming

Multiplies a vector by a Hamming windowing function.

Syntax

```
IppStatus ippsWinHamming_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHamming_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHamming_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHamming_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHamming_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHamming_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHamming_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHamming_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHamming_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHamming_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHamming_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHamming_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsWinHamming` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Hamming window and stores the result in *pDst*.

The in-place flavors of `ippsWinHamming` multiply the vector *pSrcDst* by the Hamming window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hamming window is defined as follows:

$$w_{hamming}(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{len-1}\right)$$

Example 5-29 shows how to use the function `ippsWinHamming_32f_I`

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than 3.

Example 5-29 Using the ippsWinHamming Function

```
void hamming(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinHamming_32f_I(x, 8);
      printf_32f("hamming(half) =", x, 4, ippStsNoErr);
}
Output:
    hamming(half) =  0.080000 0.253195 0.642360 0.954446
Matlab* Analog:
    >> b = hamming(8); b(1:4)'
```

WinHann

Multiplies a vector by a Hann windowing function.

Syntax

```
IppStatus ippsWinHann_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHann_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHann_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHann_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHann_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHann_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHann_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHann_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHann_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHann_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHann_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHann_64fc_I(Ipp64fc* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

The function `ippsWinHann` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Hann window and stores the result in *pDst*.

The in-place flavors of `ippsWinHann` multiply the vector *pSrcDst* by the Hann window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hann window is defined as follows:

$$w_{hann}(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{len-1}\right)$$

Example 5-30 shows how to use the function `ippsWinHann_32f_I`

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than 3.

Example 5-30 Using the ippsWinHann Function

```
void hann(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinHann_32f_I(x, 8);
      printf_32f("hann(half) =", x, 4, ippStsNoErr);
}
Output:
    hann(half) =  0.000000 0.188255 0.611260 0.950484
Matlab* Analog:
    >> N = 8; n = 0:N-1; 0.5*(1-cos(2*pi*n/(N-1)))
```

WinKaiser

Multiplies a vector by a Kaiser windowing function.

Syntax

```
IppStatus ippsWinKaiser_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, float alpha);
IppStatus ippsWinKaiser_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, float alpha);
IppStatus ippsWinKaiser_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, float alpha);
IppStatus ippsWinKaiser_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, float alpha);
IppStatus ippsWinKaiser_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, float alpha);
IppStatus ippsWinKaiser_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, float alpha);
IppStatus ippsWinKaiserQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, int alphaQ15);
```

```
IppStatus ippsWinKaiserQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, int
    alphaQ15);
IppStatus ippsWinKaiser_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32f_I(Ipp32f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_64f_I(Ipp64f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_64fc_I(Ipp64fc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiserQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
alpha	Adjustable parameter associated with the Kaiser windowing equation.
alphaQ15	Scaled version of *alpha*. The *scaleFactor* value is 15.
len	Number of elements in the vector.

Description

The function `ippsWinKaiser` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Kaiser window, and stores the result in *pDst*.

The in-place flavors of `ippsWinKaiser` multiply the vector *pSrcDst* by the Kaiser window and store the result in *pSrcDst*.

ippsWinKaiser. The function `ippsWinKaiser` allows the application to specify *alpha*. The function multiplies both real and imaginary parts of the complex vector by the same window. The Kaiser family of windows are defined as follows:

$$w_{kaiser}(n) = \frac{I_0\left(alpha\sqrt{\left(\frac{len-1}{2}\right)^2 - \left(n - \left(\frac{len-1}{2}\right)\right)^2}\right)}{I_0\left(alpha\left(\frac{len-1}{2}\right)\right)}$$

Here $I_0()$ is the modified zero-order Bessel function of the first kind.

ippsWinKaiserQ15. The function `ippsWinKaiserQ15` multiplies a vector by a Kaiser window with *alphaQ15* scaled according to the factor 15.

Example 5-31 shows how to use the function `ippsWinKaiser_32f_I`

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst*, *pSrc*, or *pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than 1.
`ippStsHugeWinErr`	Indicates an error when the Kaiser window is too big.

Example 5-31 Using the ippsWinKaiser Function

```
void kaiser(void) {
     Ipp32f x[8];
     IppStatus st;
     ippsSet_32f(1, x, 8);
     st = ippsWinKaiser_32f_I( x, 8, 1.0f );
     printf_32f("kaiser(half) =", x, 4, ippStsNoErr);
}
Output:
   kaiser(half) =  0.135534 0.429046 0.755146 0.970290
Matlab* Analog:
    >> kaiser(8,7/2)'
```

Statistical Functions

This section describes the Intel IPP functions that compute the vector measure values: maximum, minimum, mean, and standard deviation.

Sum

Computes the sum of the elements of a vector.

Syntax

```
IppStatusippsSum_32f(constIpp32f*pSrc,intlen,Ipp32f*pSum,IppHintAlgorithmhint);
```

```
IppStatus ippsSum_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pSum, IppHintAlgorithm
    hint);
IppStatus ippsSum_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSum_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pSum);
IppStatus ippsSum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pSum, int scaleFactor);
IppStatus ippsSum_32s_Sfs(const Ipp32s* pSrc, int len, Ipp32s* pSum, int scaleFactor);
IppStatus ippsSum_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pSum, int
    scaleFactor);
IppStatus ippsSum_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pSum, int
    scaleFactor);
IppStatus ippsSum_16sc32sc_Sfs(const Ipp16sc* pSrc, int len, Ipp32sc* pSum, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pSum	Pointer to the output result.
len	Number of elements in the vector.
hint	Suggests using specific code. The values for the *hint* argument are described in Flag and Hint Arguments.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsSum` is declared in the `ipps.h` file. This function computes the sum of the elements of the vector *pSrc* and stores the result in *pSum*.

The sum of the elements of *pSrc* is defined by the formula:

$$sum = \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

When computing the sum of integer numbers, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. The scaling is performed in accordance with the *scaleFactor* value.

Example 5-32 shows how to use the function `ippsSum`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSum* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 5-32 Using the `ippsSum` Function

```
void sum(void) {
      Ipp16s x[4] = {-32768, 32767, 32767, 32767}, sm;
      ippsSum_16s_Sfs(x, 4, &sm, 1);
      printf_16s("sum =", &sm, 1, ippStsNoErr);
}
Output:
    sum = 32766
Matlab* Analog:
    >> x = [-32768, 32767, 32767, 32767]; sum(x)/2
```

Max

Returns the maximum value of a vector.

Syntax

```
IppStatus ippsMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax);
IppStatus ippsMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax);
IppStatus ippsMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax);
IppStatus ippsMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax);
```

Parameters

pSrc	Pointer to the source vector.
pMax	Pointer to the output result.
len	Number of elements in the vector

Description

The function `ippsMax` is declared in the `ipps.h` file. This function returns the maximum value of the input vector *pSrc*, and stores the result in *pMax*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMax* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MaxIndx

Returns the maximum value of a vector and the index of the maximum element.

Syntax

```
IppStatus ippsMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax, int* pIndx);
IppStatus ippsMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax, int* pIndx);
IppStatus ippsMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax, int* pIndx);
IppStatus ippsMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax, int* pIndx);
```

Parameters

pSrc	Pointer to the source vector.
pMax	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the maximum element.

Description

The function `ippsMaxIndx` is declared in the `ipps.h` file. This function returns the maximum value of the input vector *pSrc*, and stores the result in *pMax*. If *pIndx* is not a `NULL` pointer, the function returns the index of the maximum element and stores it in *pIndx*. If there are several equal maximum elements, the first index from the beginning is returned. The code Example 5-33 demonstrates how to use the function `ippiMaxIndx`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-33 Using the function ippsMaxIndx

```
Ipp16s src[] = { 1, -2, 3, 8, -6 };
Ipp16s max;
int len = 5;
int  indx;

ippsMaxIndx_16s ( src, len, &max, &indx );

result: max = 8  indx = 3
```

MaxAbs

Returns the maximum absolute value of a vector.

Syntax

```
IppStatus ippsMaxAbs_16s(const Ipp16s* pSrc, int len, Ipp16s* pMaxAbs);
IppStatus ippsMaxAbs_32s(const Ipp32s* pSrc, int len, Ipp32s* pMaxAbs);
```

Parameters

pSrc	Pointer to the source vector.
pMaxAbs	Pointer to the output result.
len	Number of elements in the vector.

Description

The function ippsMaxAbs is declared in the ipps.h file. This function returns the maximum absolute value of the input vector *pSrc*, and stores the result in *pMaxAbs*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMaxAbs* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-34 shows how to use the function `ippsMaxAbs_16s`.

Example 5-34 Using the Function `ippsMaxAbs`

```
Ipp16s src[5] = { 2, -8, -3, -1, 7 };
Ipp16s maxAbs;

ippsMaxAbs_16s ( src, 5, &maxAbs );

Result
maxAbs = 8
```

MaxAbsIndx

Returns the maximum absolute value of a vector and the index of the corresponding element.

Syntax

```
IppStatus ippsMaxAbsIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMaxAbs, int* pIndx);
IppStatus ippsMaxAbsIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMaxAbs, int* pIndx);
```

Parameters

pSrc	Pointer to the source vector.
pMaxAbs	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the maximum element.

Description

The function `ippsMaxAbsIndx` is declared in the `ipps.h` file. This function returns the maximum absolute value *pMaxAbs* of the input vector *pSrc*, and the index of the corresponding element *pIndx*. If there are several elements with the equal maximum absolute value, the first index from the beginning is returned.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Min

Returns the minimum value of a vector.

Syntax

```
IppStatus ippsMin_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin);
IppStatus ippsMin_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin);
IppStatus ippsMin_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin);
IppStatus ippsMin_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin);
```

Parameters

pSrc	Pointer to the source vector.
pMin	Pointer to the output result.
len	Number of elements in the vector.

Description

The function `ippsMin` is declared in the `ipps.h` file. This function returns the minimum value of the input vector *pSrc*, and stores the result in *pMin*.

Example 5-35 shows how to use the function `ippsMin`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMin* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 5-35 Using the function `ippsMin`

```
Ipp16s src = { 1, -2, 3, 8, -6};
Ipp16s min;
int len = 5;

ippsMin_16s (src, len, &min );

result: min = -6
```

MinIndx

Returns the minimum value of a vector and the index of the minimum element.

Syntax

```
IppStatus ippsMinIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, int* pIndx);
IppStatus ippsMinIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, int* pIndx);
IppStatus ippsMinIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, int* pIndx);
IppStatus ippsMinIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, int* pIndx);
```

Parameters

pSrc	Pointer to the source vector.
pMin	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the minimum element.

Description

The function `ippsMinIndx` is declared in the `ipps.h` file. This function returns the minimum value of the input vector *pSrc* and stores the result in *pMin*. If *pIndx* is not a `NULL` pointer, the function returns the index of the minimum element and stores it in *pIndx*. If there are several equal minimum elements, the first index from the beginning is returned.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMin* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MinAbs

Returns the minimum absolute value of a vector.

Syntax

```
IppStatus ippsMinAbs_16s(const Ipp16s* pSrc, int len, Ipp16s* pMinAbs);
IppStatus ippsMinAbs_32s(const Ipp32s* pSrc, int len, Ipp32s* pMinAbs);
```

Parameters

pSrc	Pointer to the source vector.
pMinAbs	Pointer to the output result.
len	Number of elements in the vector.

Description

The function `ippsMinAbs` is declared in the `ipps.h` file. This function returns the minimum absolute value of the input vector *pSrc*, and stores the result in *pMinAbs*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMinAbs* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MinAbsIndx

Returns the minimum absolute value of a vector and the index of the corresponding element.

Syntax

```
IppStatus ippsMinAbsIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMinAbs, int* pIndx);
IppStatus ippsMinAbsIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMinAbs, int* pIndx);
```

Parameters

pSrc	Pointer to the source vector.

pMinAbs	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the corresponding element.

Description

The function `ippsMinAbsIndx` is declared in the `ipps.h` file. This function returns the minimum absolute value *pMinAbs* of the input vector *pSrc*, and the index of the corresponding element *pIndx*. If there are several elements with equal maximum absolute value, the first index from the beginning is returned.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MinMax

Returns the maximum and minimum values of a vector.

Syntax

```
IppStatus ippsMinMax_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, Ipp8u* pMax);
IppStatus ippsMinMax_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin, Ipp16u* pMax);
IppStatus ippsMinMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, Ipp16s* pMax);
IppStatus ippsMinMax_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin, Ipp32u* pMax);
IppStatus ippsMinMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, Ipp32s* pMax);
IppStatus ippsMinMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, Ipp32f* pMax);
IppStatus ippsMinMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, Ipp64f* pMax);
```

Parameters

pSrc	Pointer to the source vector.
pMin	Pointer to the minimum value.
pMax	Pointer to the maximum value.
len	Number of elements in the vector.

Description

The function `ippsMinMax` is declared in the `ipps.h` file. This function returns the minimum and maximum values of the input vector *pSrc*, and stores the results in *pMin* and *pMax*, respectively.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMin* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MinMaxIndx

Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.

Syntax

```
IppStatus ippsMinMaxIndx_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, int*
    pMinIndx, Ipp8u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin, int*
    pMinIndx, Ipp16u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, int*
    pMinIndx, Ipp16s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin, int*
    pMinIndx, Ipp32u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, int*
    pMinIndx, Ipp32s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, int*
    pMinIndx, Ipp32f* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, int*
    pMinIndx, Ipp64f* pMax, int* pMaxIndx);
```

Parameters

pSrc	Pointer to the source vector.
pMin	Pointer to the minimum value.

pMax	Pointer to the maximum value.
len	Number of elements in the vector.
pMinIndx	Pointer to the index value of the minimum element.
pMaxIndx	Pointer to the index value of the maximum element.

Description

The function `ippsMinMaxIndx` is declared in the `ipps.h` file. This function returns the minimum and maximum values of the input vector *pSrc* and stores the result in *pMin* and *pMax*, respectively. The function also returns the indexes of the minimum and maximum elements and stores them in *pMinIndx* and *pMaxIndx*, respectively. If there are several equal minimum or maximum elements, the first index from the beginning is returned.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Mean

Computes the mean value of a vector.

Syntax

```
IppStatus ippsMean_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean,
    IppHintAlgorithm hint);
IppStatus ippsMean_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pMean,
    IppHintAlgorithm hint);
IppStatus ippsMean_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean);
IppStatus ippsMean_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pMean);
IppStatus ippsMean_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean, int
    scaleFactor);
IppStatus ippsMean_32s_Sfs(const Ipp32s* pSrc,int len,Ipp32s* pMean,int
    scaleFactor);
IppStatus ippsMean_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pMean, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pMean	Pointer to the output result.
len	Number of elements in the vector
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments".
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMean` is declared in the `ipps.h` file. This function computes the mean (average) of the vector *pSrc*, and stores the result in *pMean*. The mean of *pSrc* is defined by the formula:

$$mean = \frac{1}{len} \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Example 5-36 shows how to use the function `ippsMean_32f`

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMean* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 5-36 Using the ippsMean Function

```
void mean(void) {
      Ipp32f *x = ippsMalloc_32f(1000), mean;
      int i;
      for(i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsMean_32f(x, 1000, &mean, ippAlgHintFast);
      printf_32f("mean =", &mean, 1, ippStsNoErr);
      ippsFree(x);
}
Output:
    mean = 0.492591
Matlab* Analog:
    >> x = rand(1,1000); mean(x)
```

StdDev

Computes the standard deviation value of a vector.

Syntax

```
IppStatus ippsStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pStdDev,
    IppHintAlgorithm hint);
IppStatus ippsStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pStdDev);
IppStatus ippsStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pStdDev,
    int scaleFactor);
IppStatus ippsStdDev_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pStdDev, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pStdDev	Pointer to the output result.
len	Number of elements in the vector
hint	Suggests using specific code. The values for the *hint* argument are described in Flag and Hint Arguments.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsStdDev` is declared in the `ipps.h` file. This function computes the standard deviation of the input vector *pSrc*, and stores the result in *pStdDev*. The vector length can not be less than 2. The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$stdDev = \sqrt{\frac{\sum_{n=0}^{len-1} (pSrc[n] - mean(pSrc))^2}{len-1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Example 5-37 shows how to use the function `ippsStdDev_32f`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pStdDev* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 1.

Example 5-37 Using the `ippsStdDev` Function

```
void stdev(void) {
      Ipp32f *x = ippsMalloc_32f(1000), stdev;
      int i;
      for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsStdDev_32f(x, 1000, &stdev, ippAlgHintFast);
      printf_32f("stdev =", &stdev, 1, ippStsNoErr);
      ippsFree(x);
}
Output:
    stdev = 0.286813
Matlab* Analog:
    >> x = rand(1,1000); std(x)
```

MeanStdDev

Computes the mean value and the standard deviation value of a vector.

Syntax

```
IppStatus ippsMeanStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean,
    Ipp32f* pStdDev, IppHintAlgorithm hint);
IppStatus ippsMeanStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean,
    Ipp64f* pStdDev);
IppStatus ippsMeanStdDev_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean,
    Ipp16s* pStdDev, int scaleFactor)
IppStatus ippsMeanStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pMean,
    Ipp32s* pStdDev, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pMean	Pointer to the output result - mean value.
pStdDev	Pointer to the output result - standard deviation.
len	Number of elements in the vector
hint	Suggests using specific code. The values for the *hint* argument are described in Flag and Hint Arguments.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMeanStdDev` is declared in the `ipps.h` file. This function computes both the mean value and the standard deviation of the input vector *pSrc*, and stores the results in *pMean and pStdDev* respectively. The vector length can not be less than 2. The mean of *pSrc* is defined by the formula:

$$mean = \frac{1}{len}\sum_{n=0}^{len-1} pSrc[n]$$

The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$stdev = \sqrt{\frac{\sum_{n=0}^{len-1} (pSrc[n] - mean(pSrc))^2}{len-1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 1.

Norm

Computes the C, L1, L2, or L2Sqr norm of a vector.

Syntax

```
IppStatus ippsNorm_Inf_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_32fc32f(const Ipp32fc* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm,
    int scaleFactor);

IppStatus ippsNorm_L1_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_32fc64f(const Ipp32fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);
```

```
IppStatus ippsNorm_L1_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm,
    int scaleFactor);
IppStatus ippsNorm_L1_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s* pNorm, int
    scaleFactor);

IppStatus ippsNorm_L2_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_32fc64f(const Ipp32fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm, int
    scaleFactor);

IppStatus ippsNorm_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s* pNorm,
    int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector *pSrc*.
pNorm	Pointer to the output result.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsNorm` is declared in the `ipps.h` file. This function computes the `C`, `L1`, `L2`, or `L2Sqr` norm of the source vector *pSrc* and stores the result in *pNorm*.

ippsNorm_Inf. The function `ippsNorm_Inf` computes the `C` norm defined by the formula:

$$Norm_C = \max_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L1. The function `ippsNorm_L1` computes the `L1` norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L2. The function `ippsNorm_L2` computes the `L2` norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc[n]|^2}$$

ippsNorm_L2Sqr. The function ippsNorm_L2Sqr computes the L2Sqr norm defined as square of the L2 norm.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pNorm* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

NormDiff

Computes the C, L1, L2, or L2Sqr norm of two vectors' difference.

Syntax

```
IppStatus ippsNormDiff_Inf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
    Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_32fc32f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L1_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
    Ipp64f* pNorm);
```

```
IppStatus ippsNormDiff_L1_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_32fc64f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNormDiff_L1_16s64s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp64s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L2_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
    Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_32fc64f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int
    len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp32s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp64s* pNorm, int scaleFactor);
```

Parameters

pSrc1, pSrc2	Pointers to the two source vectors; *pSrc2* can be NULL.
pNorm	Pointer to the output result.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function ippsNorm is declared in the ipps.h file. This function computes the C, L1, L2, or L2Sqr norm of the source vectors' difference, and stores the result in *pNorm*.

ippsNormDiff_Inf. The function ippsNormDiff_Inf computes the C norm defined by the formula:

$$Norm_{Inf} = \max_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L1. The function ippsNormDiff_L1 computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L2. The function ippsNormDiff_L2 computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|^2}$$

ippsNormDiff_L2Sqr. The function ippsNormDiff_L2Sqr computes the L2Sqr norm defined as square of the L2 norm.

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value.

Example 5-38 shows how to use the function ippsNormDiff.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pNorm* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-38 Using the ippsNorm Function

```
int norm( void ) {
      Ipp16s x[LEN];
      Ipp32f Norm[3];
      IppStatus st;
      int i;
      for( i=0; i<LEN; ++i ) x[i] = (Ipp16s)rand();
      ippsNormDiff_Inf_16s32f( x, 0, LEN, Norm );
      ippsNormDiff_L1_16s32f( x, 0, LEN, Norm+1 );
      st = ippsNormDiff_L2_16s32f( x, 0, LEN, Norm+2 );
      printf_32f("Norm (oo,L1,L2) =", Norm, 3, st );
      return Norm[2] <= Norm[1] && Norm[1] <= LEN*Norm[0];
}
Output:
    Norm (oo,L1,L2) =  31993.000000 1526460.000000 180270.781250
Matlab* analog:
    >> x = 32767*rand(1,100);norm(x,inf),norm(x,1),norm(x,2)
```

DotProd

Computes the dot product of two vectors.

Syntax

```
IppStatus ippsDotProd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp32f* pDp);
IppStatus ippsDotProd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int len,
    Ipp32fc* pDp);
IppStatus ippsDotProd_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp32fc* pDp);
IppStatus ippsDotProd_32f64f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
    Ipp64f* pDp);
IppStatus ippsDotProd_32fc64fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
    len, Ipp64fc* pDp);
IppStatus ippsDotProd_32f32fc64fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2,
    int len, Ipp64fc* pDp);
```

```
IppStatus ippsDotProd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
    Ipp64f* pDp);
IppStatus ippsDotProd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int len,
    Ipp64fc* pDp);
IppStatus ippsDotProd_64f64fc(const Ipp64f* pSrc1, const Ipp64fc* pSrc2, int
    len, Ipp64fc* pDp);
IppStatus ippsDotProd_16s64s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp64s* pDp);
IppStatus ippsDotProd_16sc64sc(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp64sc* pDp);
IppStatus ippsDotProd_16s16sc64sc(const Ipp16s* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp64sc* pDp);
IppStatus ippsDotProd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp32f* pDp);
IppStatus ippsDotProd_16sc32fc(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp32fc* pDp);
IppStatus ippsDotProd_16s16sc32fc(const Ipp16s* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp32fc* pDp);
IppStatus ippsDotProd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
    Ipp16s* pDp, int scaleFactor);
IppStatus ippsDotProd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp16sc* pDp, int scaleFactor);
IppStatus ippsDotProd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, int len,
    Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, int
    len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
    len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_16s16sc32sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc* pSrc2,
    int len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16s32s32s_Sfs(const Ipp16s* pSrc1, const Ipp32s* pSrc2,
    int len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_16s16sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc* pSrc2, int
    len, Ipp16sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16sc32sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
    int len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc* pSrc2, int
    len, Ipp32sc* pDp, int scaleFactor);
```

Parameters

pSrc1	Pointer to the first vector to compute the dot product value.
pSrc2	Pointer to the second vector to compute the dot product value.
pDp	Pointer to the output result.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDotProd` is declared in the `ipps.h` file. This function computes the dot product (scalar value) of two vectors, *pSrc1* and *pSrc2*, and stores the result in *pDp*.

The computation is performed as follows:

$$dp = \sum_{n=0}^{len-1} pSrc1[n] \cdot pSrc2[n]$$

To compute the dot product of complex data, use the function `ippsConj` to conjugate one of the operands. The vectors *pSrc1* and *pSrc2* must be of equal length.

Example 5-39 shows how to use the function `ippsDotProd_64f` to verify orthogonality of the sine and cosine functions. Two vectors are orthogonal to each other when the dot product of the two vectors is zero.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDp*, *pSrc1*, or *pSrc*2 pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 5-39 Using the Function ippsDotProd to Verify Orthogonality of Sin and Cos

```
void dotprod(void) {
      Ipp64f x[10], dp;
      int n;
      for (n = 0; n<10; ++n) x[n] = sin(IPP_2PI * n / 8);
      ippsDotProd_64f(x, x+2, 8, &dp);
      printf_64f("dp =", &dp, 1, ippStsNoErr);
}
Output:
    dp = 0.000000
Matlab* Analog:
    >> n = 0:9; x = sin(2*pi*n/8); a = x(1:8); b = x(3:10); a*b'
```

MaxEvery, MinEvery

Computes maximum or minimum value for each pair of elements of two vectors.

Syntax

```
IppStatus ippsMaxEvery_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsMaxEvery_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsMaxEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMaxEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMaxEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);

IppStatus ippsMinEvery_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsMinEvery_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsMinEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMinEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMinEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the first input vector.
pSrcDst	Pointer to the second input vector which stores the result.

len	Number of elements in the vector.

Description

The function `ippsMaxEvery` is declared in the `ipps.h` file. This function computes the maximum between each pair of corresponding elements of two input vectors and stores the result in *pSrcDst*.
The function `ippsMinEvery` is declared in the `ipps.h` file. This function computes minimum values likewise.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

CountInRange

Computes the number of elements of the vector whose values are in the specified range.

Syntax

```
IppStatus ippsCountInRange_32s(const Ipp32s* pSrc, int len, int* pCounts,
    Ipp32s lowerBound, Ipp32s upperBound);
```

Parameters

pSrc	Pointer to the first input vector.
pCounts	Pointer to the second input vector which stores the result.
len	Number of elements in the vector.
lowerBound	Lower boundary of the range.
upperBound	Upper boundary of the range.

Description

The function `ippsCountInRange` is declared in the `ipps.h` file. This function computes the number of elements of the vector *pSrc* whose values are in the range *lowerBound* < *pSrc*[i] < *upperBound*. The number of such elements are stored in the *pCounts*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSCounts* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Sampling Functions

The functions described in this section manipulate signal samples. Sampling functions are used to change the sampling rate of the input signal and thus to obtain the signal vector of a required length. The functions perform the following operations:

- Insert zero-valued samples between neighboring samples of a signal (up-sample).
- Remove samples from between neighboring samples of a signal (down-sample).

The upsampling and downsampling functions are used by some filtering functions described in Chapter 6.

SampleUp

Up-samples a signal, conceptually increasing its sampling rate by an integer factor.

Syntax

```
IppStatus ippsSampleUp_16s (const Ipp16s* pSrc, int srcLen, Ipp16s* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32f (const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64f (const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_16sc (const Ipp16sc* pSrc, int srcLen, Ipp16sc* pDst,
    int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32fc (const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst,
    int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64fc (const Ipp64fc* pSrc, int srcLen, Ipp64fc* pDst,
    int* pDstLen, int factor, int* pPhase);
```

Parameters

pSrc	Pointer to the input array (the signal to be up-sampled).
srcLen	Number of samples in the input array *pSrc*.
pDst	Pointer to the output array.
pDstLen	Pointer to the length value of the output array *pDst*.
factor	Factor by which the signal is up-sampled. That is, *factor* -1 zeros are inserted after each sample of the input array *pSrc*.
pPhase	Pointer to the **input** phase value which determines where each sample from *pSrc* lies within each output block of *factor* samples in *pDst*. The value of *pPhase* is required to be in the range [0; *factor*-1]. The **output** value of *pPhase* can be used for the next up-sampling with the same *factor* and next *pSrc*.

Description

The function ippsSampleUp is declared in the ipps.h file. This function up-samples the *srcLen*-length input array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Up-sampling inserts *factor*-1 zeros between each sample of *pSrc*. The *pPhase* argument determines where each sample from the input array lies within each output block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1].

For example, if the input phase is 0, then every *factor* samples of the output array begin with the corresponding input array sample, the other *factor*-1 samples are equal to 0. The output array length is stored by the *pDstLen* address.

The *pPhase* value is the phase of an input array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

The ippsSampleUp functionality can be described as follows:

$$pDst[factor * n + phase] = pSrc[n],\ 0 \le n < srcLen$$

$$pDst[factor * n + m] = 0,\ 0 \le n < srcLen,\ 0 \le m < factor,\ m \ne phase$$

$$pDstLen = factor * srcLen$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pDst*, *pSrc*, *pDstLen*, or *pPhase* pointer is NULL.

ippStsSizeErr	Indicates an error if *srcLen* is less than or equal to 0.
ippStsSampleFactorErr	Indicates an error if *factor* is less than or equal to 0.
ippStsSamplePhaseErr	Indicates an error when *pPhase* is negative, or bigger than or equal to *factor*.

SampleDown

Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

Syntax

```
IppStatus ippsSampleDown_16s(const Ipp16s* pSrc, int srcLen, Ipp16s* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_16sc(const Ipp16sc* pSrc, int srcLen, Ipp16sc* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst, int*
    pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc* Dst, int*
    pDstLen, int factor, int* pPhase);
```

Parameters

pSrc	Pointer to the input array holding the samples to be down-sampled.
srcLen	Number of samples in the input array *pSrc*.
pDst	Pointer to the array that holds the output of the function ippsSampleDown.
pDstLen	Pointer to the length of the output array *pDst*.
factor	Factor by which the signal is down-sampled. That is, *factor* - 1 samples are discarded from every block of *factor* samples in *pSrc*.

pPhase	Pointer to the **input** phase value that determines which of the samples within each block of *factor* samples from *pSrc* is not discarded and copied to *pDst*. The value of *pPhase* is required to be in the range [0; *factor*-1]. The **output** value of *pPhase* can be used for the next down-sampling (sub-sampling) with the same *factor* and next *pSrc*.

Description

The function ippsSampleDown is declared in the ipps.h file. This function down-samples the *srcLen* -length array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Down-sampling discards *factor* - 1 samples from *pSrc*, copying one sample from each block of *factor* samples from *pSrc* to *pDst*. The *pPhase* argument determines which of the samples in each block is not discarded and where it lies within each input block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1]. The output array length is stored by the *pDstLen* address.

The *pPhase* value is the phase of an input array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

You can use the FIR multi-rate filter to combine filtering and resampling, for example, for antialiasing filtering before the sub-sampling procedure.

The ippsSampleDown functionality can be described as follows:

$$pDstLen = (srcLen + factor - 1 - phase)/factor$$

$$pDst[n] = pSrc[factor * n + phase],\ 0 \le n < pDstLen$$

$$phase = (factor + phase - srcLen \% factor) \% factor$$

Example 5-40 shows how to use the function ippsSampleDown.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst*, *pSrc*, *pDstLen*, or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcLen* is less than or equal to 0.
ippStsSampleFactorErr	Indicates an error when *factor* is less than or equal to 0.

ippStsSamplePhaseErr	Indicates an error when *pPhase* is negative, or bigger than or equal to *factor*.

Example 5-40 Using the ippsSampleDown Function

```
void sampling( void ) {
      Ipp16s x[8] = { 1,2,3,4,5,6,7,8 };
      Ipp16s y[8] = {9,10,11,12,13,14,15,16}, z[8];
      int dstLen1, dstLen2, phase = 2;
      IppStatus st = ippsSampleDown_16s(x, 8, z, &dstLen1, 3, &phase);
      st = ippsSampleDown_16s(y, 8, z+dstLen1, &dstLen2, 3, &phase);
      printf_16s("down-sumpling =", z, dstLen1+dstLen2, st);
}
Output:
    down-sampling = 3 6 9 12 15
```

Filtering Functions

6

This chapter describes Intel® IPP functions that perform convolution and correlation operations, as well as linear and non-linear filtering. These functions are grouped into the following sections:

Convolution and Correlation Functions

Filtering Functions

Each section starts with a table that lists functions described in more detail later in this section, together with the brief description of operations that these functions perform.

Convolution and Correlation Functions

Convolution is an operation used to define an output signal from any linear time-invariant (LTI) processor in response to any input signal.

The correlation functions described later in this section estimate either the auto-correlation of a source vector or the cross-correlation of two vectors

The full list of functions in this group is given in Table 6-1.

Table 6-1 Intel IPP Convolution and Correlation Functions

Function Base Name	Operation
`Conv`	Performs finite, linear convolution of two sequences.
`ConvBiased`	Computes the specified number of elements of the full finite linear convolution of two vectors.
`ConvCyclic`	Performs cyclic convolution of two sequences of the fixed size.

Table 6-1 Intel IPP Convolution and Correlation Functions (continued)

Function Base Name	Operation
AutoCorr	Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.
CrossCorr	Estimates the cross-correlation of two vectors.
UpdateLinear	Integrates an input vector with specified integration weight.
UpdatePower	Integrates the square of an input vector with specified integration weight

Conv

Performs finite, linear convolution of two vectors.

Syntax

```
IppStatus ippsConv_32f(const Ipp32f* pSrc1, int src1Len, const Ipp32f* pSrc2,
    int src2Len, Ipp32f* pDst);
IppStatus ippsConv_64f(const Ipp64f* pSrc1, int src1Len, const Ipp64f* pSrc2,
    int src2Len, Ipp64f* pDst);
IppStatus ippsConv_16s_Sfs(const Ipp16s* pSrc1, int src1Len, const Ipp16s*
    pSrc2, int src2Len, Ipp16s* pDst, int scaleFactor);
```

Parameters

pSrc1, pSrc2	Pointers to the source vectors.
src1Len	Number of elements in the vector *pSrc1*.
src2Len	Number of elements in the vector *pSrc2*.
pDst	Pointer to the destination vector.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsConv` is declared in the `ipps.h` file. This function performs finite linear convolution of two sequences. The *src1Len*- length vector *pSrc1* is convolved with the *src2Len*-length vector *pSrc2* to produce an (*src1Len* + *src2Len* - 1) -length vector *pDst*. The result of the convolution is defined as follows:

$$pDst[n] = \sum_{k=0}^{n} pSrc1[k] \cdot pSrc2[n-k] \qquad 0 \le n \le src1Len + src2Len - 1$$

Here $pSrc1[i] = 0$, if $i \ge src1Len$, and $pSrc2[j] = 0$, if $j \ge src2Len$.

Example 6-1 shows the code for the convolution of two vectors using `ippsConv_16s_Sfs` function.

Example 6-1 Using the Function `ippsConv` to Convolve Two Vectors

```
IppStatus convolution(void) {
      Ipp16s x[5] = {-2,0,1,-1,3}, h[2] = {0,1}, y[6];
      IppStatus st = ippsConv_16s_Sfs(x, 5, h, 2, y, 0);
      printf_16s("conv =", y, 6, st);
      return st;
}
Output:
    conv = 0 -2 0 1 -1 3
Matlab* Analog:
    >> x = [-2,0,1,-1,3]; h = [0,1]; y = conv(x,h)
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *src1Len* or *src2Len* is less than or equal to 0.
`ippStsMemAllocErr`	Indicates an error when there is not enough memory for internal buffers.

ConvBiased

Computes the specified number of elements of the full finite linear convolution of two vectors.

Syntax

```
IppStatus ippsConvBiased_32f(const Ipp32f* pSrc1, int src1Len, const Ipp32f* pSrc2,
    int src2Len, Ipp32f* pDst, int dstLen, int bias);
```

Parameters

pSrc1, pSrc2	Pointers to the two vectors to be convolved.
src1Len	Number of elements in the vector *pSrc1*.
src2Len	Number of elements in the vector *pSrc2*.
pDst	Pointer to the vector *pDst*. This vector stores the result of the convolution.
dstLen	Number of elements in the vector *pDst*.
bias	Parameter that specifes the starting element of the convolution.

Description

The function `ippsConvBiased` is declared in the `ipps.h` file. This function computes *dstLen* elements of finite linear convolution of two specified vectors *pSrc1* and *pSrc2* starting with an element that is specified by the *bias*. The result is stored in the vector *pDst*.

Example 6-2 shows how to call the function `ippsConvBiased`.

Example 6-2 Using the Function `ippsConvBiased`

```
void func_convbiased()
{
    Ipp32f pSrc1[5] = {1.1, -2.0, 3.5, 2.2, 0.0};
    Ipp32f pSrc2[4] = {0.0, 0.2, 2.5, -1.0};
    const int len = 10;
    Ipp32f pDst[len];
    int bias = 1;

    ippsZero_32f(pDst, len);
    ippsConvBiased_32f(pSrc1, 5, &pSrc2[1], 3, pDst, len, bias);
}
Result:
        pDst -> 0.2  2.3  -4.3  9.2  5.5  0.0  0.0  0.0  0.0  0.0
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *src1Len* or *src2Len* is less than or equal to 0.

ConvCyclic

Performs cyclic convolution of two sequences of the fixed size.

Syntax

```
IppStatus ippsConvCyclic8x8_32f(const Ipp32f* x, const Ipp32f* h, Ipp32f* y);
IppStatus ippsConvCyclic4x4_32f32fc(const Ipp32f* x, const Ipp32fc* h, Ipp32fc* y);
IppStatus ippsConvCyclic8x8_16s_Sfs(const Ipp16s* x, const Ipp16s* h, Ipp16s* y,
    int scaleFactor);
```

Parameters

x, *h*	Pointers to the source vectors.
y	Pointer to the destination vector that stores the result of convolution.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Discussion

The function ippsConvCyclic is declared in the ipps.h file. This function performs cyclic convolution of two sequences of the fixed size.

Return Values

ippStsNoErr	Indicates no error.

AutoCorr

Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.

Syntax

```
IppStatus ippsAutoCorr_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst,
    int dstLen);
IppStatus ippsAutoCorr_NormB_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst,
    int dstLen);
IppStatus ippsAutoCorr_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int
    dstLen);
IppStatus ippsAutoCorr_NormA_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst,
    int dstLen);
IppStatus ippsAutoCorr_NormB_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst,
    int dstLen );
IppStatus ippsAutoCorr_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst, int
    dstLen);
IppStatus ippsAutoCorr_NormA_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc*
    pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc*
    pDst, int dstLen);
IppStatus ippsAutoCorr_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc* pDst, int
    dstLen);
IppStatus ippsAutoCorr_NormA_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc*
    pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc*
    pDst, int dstLen);
IppStatus ippsAutoCorr_16s_Sfs(const Ipp16s* pSrc, int srcLen, Ipp16s* pDst,
    int dstLen, int scaleFactor );
IppStatus ippsAutoCorr_NormA_16s_Sfs( const Ipp16s* pSrc, int srcLen, Ipp16s*
    pDst, int dstLen, int scaleFactor );
IppStatus ippsAutoCorr_NormB_16s_Sfs(const Ipp16s* pSrc, int srcLen, Ipp16s*
    pDst, int dstLen, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
srcLen	The number of elements in the source vector.
pDst	Pointer to the destination vector, which stores the estimated auto-correlation results of the source vector.
dstLen	The number of elements in the destination vector (length of auto-correlation).
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The `ippsAutoCorr` function is declared in the `ipps.h` file. This function estimates normal auto-correlation of the *srcLen*-length source vector *pSrc* and stores the results in the *dstLen*-length vector *pDst*. Function flavors `ippsAutoCorr_NormA` and `ippsAutoCorr_NormB` compute biased and unbiased auto-correlation of the source vector, respectively. The resulting vector *pDst* is defined by the following equations:

$$pDst[n] = \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(normal)}$$

$$pDst[n] = \frac{1}{srcLen} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(biased)}$$

$$pDst[n] = \frac{1}{srcLen - n} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(unbiased)}$$

where

$$pSrc[i] = \begin{cases} pSrc[i], & 0 \le i < srcLen \\ 0, & otherwise \end{cases}$$

Example 6-3 shows how to call the function `ippsAutoCorr_NormB_32f`.

Example 6-3 Using the Function ippsAutoCorr

```
void func_autocorr()
{
    Ipp32f pSrc[5] = {0.2, 3.1, 2.0, 1.2, -1.1};
    const int dstLen = 10;
    Ipp32f pDst[dstLen];

    ippsAutoCorr_NormB_32f(pSrc, 5, pDst, dstLen);
}
Result:
pDst -> 3.3 2.0 0.6 -1.6 -0.2 0.0 0.0 0.0 0.0 0.0
```

Application Note

The auto-correlation estimates are computed only for positive lags, since the auto-correlation for a negative lag value is the complex conjugate of the auto-correlation for the equivalent positive lag.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcLen* or *dstLen* is less than or equal to 0.

See Also

ippsCrossCorr	Estimates the cross-correlation of two vectors.

CrossCorr

Estimates the cross-correlation of two vectors.

Syntax

```
IppStatus ippsCrossCorr_32f(const Ipp32f* pSrc1, int src1Len, const Ipp32f* pSrc2,
    int src2Len, Ipp32f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64f(const Ipp64f* pSrc1, int src1Len, const Ipp64f* pSrc2,
    int src2Len, Ipp64f* pDst, int dstLen, int lowLag);
```

```
IppStatus ippsCrossCorr_32fc(const Ipp32fc* pSrc1, int src1Len, const Ipp32fc* pSrc2,
    int src2Len, Ipp32fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64fc(const Ipp64fc* pSrc1, int src1Len, const Ipp64fc* pSrc2,
    int src2Len, Ipp64fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_16s_Sfs(const Ipp16s* pSrc1, int src1Len, const Ipp16s*
    pSrc2, int src2Len, Ipp16s* pDst, int dstLen, int lowLag, int scaleFactor);
IppStatus ippsCrossCorr_16s64s(const Ipp16s* pSrc1, int src1Len, const Ipp16s* pSrc2,
    int src2Len, Ipp64s* pDst, int dstLen, int lowLag);
```

Parameters

pSrc1	Pointer to the first source vector.
src1Len	Number of elements in the vector *pSrc1*.
pSrc2	Pointer to the second source vector.
src2Len	Number of elements in the vector *pSrc2*.
pDst	Pointer to the vector which stores the results of the estimated cross-correlation of the vectors *pSrc1* and *pSrc2*.
dstLen	Number of elements in the vector *pDst*, which determines the range of lags at which the correlation estimates are computed.
lowLag	Lower value of the range of lags at which the correlation estimates are computed.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsCrossCorr` is declared in the `ipps.h` file. This function estimates cross-correlation of the *src1Len*-length vector *pSrc1* and the *src2Len*-length vector *pSrc2*, and stores the results in the vector *pDst*.

The resulting vector *pDst* is defined by the equation:

$$pDst[n] = \sum_{i=0}^{src1Len-1} conj(pSrc1[i]) \cdot pSrc2[n+i+lowLag] \quad ,$$

where $0 \le n < dstLen$, and

$$pSrc2[j] = \begin{cases} pSrc2[j], & 0 \le j < src2Len \\ 0, & otherwise \end{cases}$$

Example 6-4 shows how to use the function ippsCrossCorr.

Example 6-4 Using the Function ippsCrossCorr

```
void crossCorr(void) {
      #undef LEN
      #define LEN 11
      Ipp32f win[LEN], y[LEN];
      IppStatus st;
      ippsSet_32f (1, win, LEN);
      ippsWinHamming_32f_I (win, LEN);
      st = ippsCrossCorr_32f (win, LEN, win, LEN, y, LEN, -(LEN-1));
      printf_32f("cross corr =", y,7,st);
}
Output:
    cross corr = 0.006400 0.026856 0.091831 0.242704 0.533230
    1.009000 1.672774
Matlab* analog:
    >> x = hamming(11)'; y = xcorr(x,x); y(1:7)
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1* or *pSrc2* pointer is NULL.
ippStsSizeErr	Indicates an error when *src1Len* or *src2Len* is less than or equal to 0.

UpdateLinear

Integrates an input vector with specified integration weight.

Syntax

```
IppStatus ippsUpdateLinear_16s32s_I(const Ipp16s* pSrc, int len, Ipp32s* pSrcDst,
    int srcShiftRight, Ipp16s alpha, IppHintAlgorithm hint);
```

Parameters

pSrc	Pointer to the source vector.
len	Number of elements of the vector.
pSrcDst	Pointer to the input value and output result.
srcShiftRight	Shift value; must be non-negative.

alpha	Integration weight.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."

Description

The function ippsUpdateLinear is declared in the ipps.h file. This function performs *len* iterations in which the sum

$alpha*pSrcDst + (1-alpha)*pSrc[i]_{shift}$

is calculated and stored in *pSrcDst*.
Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and $pSrc[i]_{shift}$ is a source vector element right-shifted by the non-negative value *srcShiftRight*.

Example 6-5 shows how to use the function ippsUpdateLinear.

Example 6-5 Using the Function ippsUpdateLinear

```
void func_updatelinear()
{
    Ipp16s pSrc[2] = {16, 32};
    Ipp32s pSrcDst =1;
    Ipp16s alpha = 2;
    int shift = 2;
    int srcLen = 2;

    ippsUpdateLinear_16s32s_I(pSrc, 2, &pSrcDst, shift, alpha,
ippAlgHintFast);

}

Result: -12
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

UpdatePower

Integrates the square of an input vector with specified integration weight.

Syntax

```
IppStatus ippsUpdatePower_16s32s_I(const Ipp16s* pSrc, int len, Ipp32s* pSrcDst,
    int srcShiftRight,Ipp16s alpha, IppHintAlgorithm hint);
```

Parameters

pSrc	Pointer to the source vector.
len	Number of elements of the vector.
pSrcDst	Pointer to input and output
srcShiftRight	Shift value.
alpha	Integration weight.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."

Description

The function `ippsUpdatePower` is declared in the `ipps.h` file. This function performs *len* iterations in which the sum

$alpha*pSrcDst + (1-alpha)*(pSrc[i]*pSrc[i])_{shift}$

is calculated and stored in *pSrcDst*.

Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and $(pSrc[i]*pSrc[i])_{shift}$ is the squared *i*-th element right-shifted by the non-negative value *srcShiftRight*.

Example 6-6 shows how to use the function `ippsUpdatePower`

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

.

Example 6-6 Using the Function ippsUpdatePower

```
void func_updatepower()
{
    Ipp16s pSrc[2] = {16, 32};
    Ipp32s pSrcDst = 1;
    Ipp16s alpha = 3;
    int shift = 1;
    int srcLen = 2;

    ippsUpdatePower_16s32s_I(pSrc, 2, &pSrcDst, shift, alpha,
ippAlgHintFast);
}
Result:  -1783
```

Filtering Functions

The Intel IPP functions described in this section implement the following types of filters:

- finite impulse response (FIR) filter
- adaptive finite impulse response using least mean squares (LMS) filter
- infinite impulse response (IIR) filter
- median filter.

A special set of functions is designed to generate filter coefficients for different types of FIR filters.

The full list of filtering functions is given in Table 6-2

Table 6-2 Intel IPP Filtering Functions

Function Base Name	Operation
SumWindow	Initializes a single-rate FIR filter state structure.
FIR Filter Functions	
FIRInitAlloc	Allocates memory and initializes a single-rate FIR filter state structure.
FIRMRInitAlloc	Allocates memory and initializes a multi-rate FIR filter state structure.
FIRFree	Frees memory allocated for the FIR filter state structure.
FIRInit	Initializes a single-rate FIR filter state structure.
FIRMRInit	Initializes a multi-rate FIR filter state structure.
FIRGetStateSize, FIRMRGetStateSize	Computes the size of an external buffer for FIR filter structure.
ippsFIRGetTaps	Retrieves the tap values from the FIR filter state structure.
ippsFIRSetTaps	Sets the taps values in the FIR filter state structure.
FIRGetDlyLine	Retrieves the delay line contents from the FIR filter state structure.
FIRSetDlyLine	Sets the delay line contents in the FIR filter state structure.
FIROne	Filters a single sample through a FIR filter.
FIR	Filters a block of samples through a FIR filter.
Direct versions	
FIROne_Direct	Directly filters a single sample through a FIR filter.

Table 6-2 Intel IPP Filtering Functions (continued)

Function Base Name	Operation
FIR_Direct	Directly filters a block of samples through a single-rate FIR filter.
FIRMR_Direct	Directly filters a block of samples through a multi-rate FIR filter.
Sparse FIR Filter	
FIRSparseInit	Initializes a sparse FIR filter structure.
FIRSparseGetStateSize	Computes the size of the external buffer for the sparse FIR filter structure.
FIRSparse	Filters a block of samples through a sparse FIR filter.
FIR Filter Coefficient Generating Functions	
FIRGenLowpass	Computes the lowpass FIR filter coefficients.
FIRGenHighpass	Computes the highpass FIR filter coefficients.
FIRGenBandpass	Computes the bandpass FIR filter coefficients.
FIRGenBandstop	Computes the bandstop FIR filter coefficients.
Single-Rate FIR LMS Filter Functions	
FIRLMSInitAlloc	Allocates memory and initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.
FIRLMSFree	Closes an adaptive FIR filter that uses the LMS algorithm.
FIRLMSGetTaps	Gets the taps of a FIR LMS filter.
FIRLMSGetDlyLine	Retrieves the delay line contents from the FIR LMS filter.
FIRLMSSetDlyLine	Sets the delay line contents in the FIR LMS filter.
FIRLMS	Filters an array through a FIR LMS filter.
Direct versions	
FIRLMSOne_Direct	Filters a single sample through a FIR LMS filter.
Multi-Rate FIR LMS Filter Functions	
FIRLMSMRInitAlloc	Allocates memory and initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.
FIRLMSMRFree	Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.
FIRLMSMRSetMu	Sets the adaptation step.

Table 6-2 Intel IPP Filtering Functions (continued)

Function Base Name	Operation
FIRLMSMRUpdateTaps	Updates the filter coefficients using the adaptation error value.
FIRLMSMRGetTaps	Retrieves tap values in the multi-rate FIR LMS filter.
FIRLMSMRSetTaps	Sets tap values in the multi-rate FIR LMS filter.
FIRLMSMRGetTapsPointer	Returns the pointer to the filter coefficients.
FIRLMSMRGetDlyLine	Retrieves the delay line contents from the multi-rate FIR LMS filter state.
FIRLMSMRSetDlyLine	Sets the delay line contents in the multi-rate FIR LMS filter state.
FIRLMSMRGetDlyVal	Gets one delay line values from the specified position.
FIRLMSMRPutVal	Places the input value in the delay line.
FIRLMSMROne	Filters data placed in the delay line.
FIRLMSMROneVal	Filters one input value.
IIR Filter Functions	
IIRGenLowpass IIRGenHighpass	Computes lowpass and highpass IIR filter coefficients.
IIRInitAlloc	Allocates memory and initializes an arbitrary IIR filter state.
IIRInitAlloc_BiQuad	Allocates memory and initializes a biquad IIR filter state.
IIRFree	Closes an IIR filter state.
IIRInit	Initializes an arbitrary IIR filter state.
IIRInit_BiQuad	Initializes a biquad IIR filter state
IIRGetStateSize	Computes the length of the external buffer for the arbitrary IIR filter state structure.
IIRGetStateSize_BiQuad	Computes the length of the external buffer for thebiquad IIR filter state structure.
IIRSetTaps	Sets the taps in an IIR filter state
IIRGetDlyLine	Retrieves the delay line values from the IIR filter state.
IIRSetDlyLine	Sets the delay line values in the IIR filter state.
IIROne	Filters a single sample through an IIR filter.
IIR	Filters a block of samples through an IIR filter.
Direct versions	

Table 6-2 Intel IPP Filtering Functions (continued)

Function Base Name	Operation
IIROne_Direct	Directly filters a single sample through the arbitrary IIR filter.
IIROne_BiQuadDirect	Directly filters a single sample through the biquad IIR filter.
IIR_Direct	Directly filters a block of samples through the arbitrary IIR filter.
IIR_BiQuadDirect	Directly filters a block of samples through the biquad IIR filter
Sparse IIR Filter	
IIRSparseInit	Initializes a sparse IIR filter structure.
IIRSparseGetStateSize	Computes the size of the external buffer for the sparse IIR filetr structure.
IIRSparse	Filters a block of samples through a sparse IIR filter.
Median Filter Functions	
FilterMedian	Computes median values for each input array element.

SumWindow

Sums elements in the mask applied to each element of a vector.

Syntax

```
IppStatus ippsSumWindow_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len, int
    maskSize);
IppStatus ippsSumWindow_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len, int
    maskSize);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements of the vector.
maskSize	Size of the mask.

Description

The function ippsSumWindow is declared in the ipps.h file. This function sets each element in the destination vector *pDst* as the sum of *maskSize* elements of the source vector *pSrc*. The computation is performed as follows:

$$pDst[n] = \sum_{k=n}^{maskSize} pSrc[k] \ , 0 \le n < len$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsMaskSizeErr	Indicates an error when *maskSize* is less than or equal to 0.

FIR Filter Functions

The functions described in this section perform a finite impulse response filtering of input data. The functions initialize a finite impulse response filter, get and set the delay lines and filter coefficients (taps), and perform filtering. To use the FIR filter functions, follow this general scheme:

1. Call either ippsFIRInitAlloc or ippsFIRMRInitAlloc to allocate memory and initialize the taps and the delay line in the filter state structure of a single-rate or multi-rate filter, respectively.
   Or call either ippsFIRInit or ippsFIRMRInit to initialize the taps and the delay line in the corresponding filter state structure in the previously created external buffer. The size of this buffer should be computed beforehand by calling the functions ippsFIRGetStateSize or ippsFIRMRGetStateSize, respectively.
2. Call ippsFIROne to filter a single sample through a single-rate filter and/or call ippsFIR to filter a block of consecutive samples through a single-rate or multi-rate filter.
3. To set new taps and delay line values in the previously initialized filter state, call the functions ippsFIRSetTaps and ippsFIRSetDlyLine. To get taps and delay line values of the initialized filter state, call the functions ippsFIRGetTaps and ippsFIRSetDlyLine.
4. Call ippsFIRFree to free dynamic memory associated with the FIR filter state structure created by ippsFIRInitAlloc or ippsFIRMRInitAlloc.

Alternatively, you may use the direct version of the functions ippsFIROne_Direct, ippsFIR_Direct, ippsFIRMR_Direct. These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

Special set of functions allows to compute the filter coefficients for different filters.

FIRInitAlloc

Allocates memory and initializes a single-rate FIR filter state.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsFIRInitAlloc32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
    pTaps, int tapsLen, int tapsFactor, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32s_16s32f(IppsFIRState32s_16s** ppState, const
    Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc_32s(IppsFIRState_32s** ppState, const Ipp32s* pTaps,
    int tapsLen, const Ipp32s* pDlyLine);
IppStatus ippsFIRInitAlloc64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp32s* pDlyLine);

IppStatus ippsFIRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, const Ipp32sc* pDlyLine);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsFIRInitAlloc_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps,
    int tapsLen, const Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRInitAlloc64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp32f* pDlyLine);
IppStatus ippsFIRInitAlloc_64f(IppsFIRState_64f** ppState,const Ipp64f* pTaps,
    int tapsLen, const Ipp64f* pDlyLine);

IppStatus ippsFIRInitAlloc_32fc(IppsFIRState_32fc** ppState, const Ipp32fc* pTaps,
    int tapsLen, const Ipp32fc* pDlyLine);
IppStatusippsFIRInitAlloc64fc_32fc(IppsFIRState64fc_32fc**ppState,constIpp64fc*
    pTaps, int tapsLen, const Ipp32fc* pDlyLine);
IppStatus ippsFIRInitAlloc_64fc(IppsFIRState_64fc** ppState, const Ipp64fc* pTaps,
    int tapsLen, const Ipp64fc* pDlyLine);
```

Parameters

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *tapsLen*.
ppState	Pointer to the pointer to the FIR state structure to be created.

Description

The function `ippsFIRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a single-rate FIR filter state. The initialization function copies the tap values from the *tapsLen* -length array *pTaps* into the state structure *ppState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the *tapsLen* -length array *pDlyLine* is not `NULL`, the array content is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

Note that the delay line length is different than that for direct FIR filters (where this length is doubled).

If the state is not created, the initialization function returns an error status.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsNullPtrErr	Indicates an error when *pTaps* or *ppState* is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRMRInitAlloc

Allocates memory and initializes a multi-rate FIR filter state.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsFIRMRInitAlloc32s_16s(IppsFIRState32s_16s** ppState, const
    Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32s_16s32f(IppsFIRState32s_16s** ppState, const
    Ipp32f *pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32f_16s(IppsFIRState32f_16s** ppState, const
    Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_16s(IppsFIRState64f_16s** ppState, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_32s(IppsFIRState64f_32s** ppState, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32s* pDlyLine);

IppStatus ippsFIRMRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** ppState,
    const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** ppState,const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32sc* pDlyLine);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsFIRMRInitAlloc_32f(IppsFIRState_32f** ppState, const Ipp32f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32f* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_32f(IppsFIRState64f_32f** ppState, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32f* pDlyLine);
IppStatus ippsFIRMRInitAlloc_64f(IppsFIRState_64f** ppState, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp64f* pDlyLine);

IppStatus ippsFIRMRInitAlloc_32fc(IppsFIRState_32fc** ppState, const Ipp32fc*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc_64fc(IppsFIRState_64fc** ppState, const Ipp64fc*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp64fc* pDlyLine);
```

Parameters

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).
downFactor	Downsampling factor.
downPhase	Phase for downsampled signal.

upFactor	Upsampling factor.
upPhase	Phase for upsampled signal.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array (*tapsLen* + *upFactor* - 1) / *upFactor*.
ppState	Pointer to the pointer to the FIR state structure to be created.

Description

The function `ippsFIRMRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a multi-rate FIR filter state. The initialization functions copy the taps from the *tapsLen* -length array *pTaps* into the state structure *pState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

If the state is not created, the initialization function returns an error status.

The function initializes the state structure *pState* of a multi-rate filter; that is, a filter that internally upsamples and/or downsamples using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the `ippsSampleUp` function for more details). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The argument *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see description of the `ippsSampleDown` function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The argument *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array *pDlyLine* is non-`NULL`, its length is defined as (*tapsLen* + *upFactor* - 1) / *upFactor*.

Example 6-8 shows how to use the function `ippsFIRMRInitAlloc_32f`.

Return Values

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsNullPtrErr	Indicates an error when *pTaps* or *ppState* is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsFIRMRFactorErr	Indicates an error when *upFactor* (*downFactor*) is less than or equal to 0.
ippStsFIRMRPhaseErr	Indicates an error when *upPhase* (*downPhase*) is negative, or greater than or equal to *upFactor* (*downFactor*).
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRFree

Closes a FIR filter state.

Syntax

```
IppStatus ippsFIRFree32s_16s(IppsFIRState32s_16s* pState);
IppStatus ippsFIRFree32f_16s(IppsFIRState32f_16s* pState);
IppStatus ippsFIRFree64f_16s(IppsFIRState64f_16s* pState);
IppStatus ippsFIRFree_32s(IppsFIRState_32s* pState);
IppStatus ippsFIRFree64f_32s(IppsFIRState64f_32s* pState);
IppStatus ippsFIRFree32sc_16sc(IppsFIRState32sc_16sc* pState);
IppStatus ippsFIRFree32fc_16sc(IppsFIRState32fc_16sc* pState);
IppStatus ippsFIRFree64fc_16sc(IppsFIRState64fc_16sc* pState);
IppStatus ippsFIRFree64fc_32sc(IppsFIRState64fc_32sc* pState);

IppStatus ippsFIRFree_32f(IppsFIRState_32f* pState);
IppStatus ippsFIRFree64f_32f(IppsFIRState64f_32f* pState);
IppStatus ippsFIRFree_64f(IppsFIRState_64f* pState);
IppStatus ippsFIRFree_32fc(IppsFIRState_32fc* pState);
IppStatus ippsFIRFree64fc_32fc(IppsFIRState64fc_32fc* pState);
```

```
IppStatus ippsFIRFree_64fc(IppsFIRState_64fc* pState);
```

Parameters

pState	Pointer to the FIR filter state structure to be closed.

Description

The function `ippsFIRFree` is declared in the `ipps.h` file. This function closes the FIR filter state by freeing all memory associated with the filter state structure created by `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`. Call `ippsFIRFree` after filtering is completed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRInit

Initializes a single-rate FIR filter state.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsFIRInit32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
    pTaps, int tapsLen, int tapsFactor, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit32s_16s32f(IppsFIRState32s_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_32s(IppsFIRState_32s** ppState, const Ipp32s* pTaps, int
    tapsLen, const Ipp32s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp32s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRInit32sc_16sc(IppsFIRState32sc_16sc** ppState, const Ipp32sc*
    pTaps, int tapsLen, int tapsFactor, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit32sc_16sc32fc(IppsFIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit32fc_16sc(IppsFIRState32fc_16sc** ppState, const Ipp32fc*
    pTaps, int tapsLen, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64fc_16sc(IppsFIRState64fc_16sc** ppState, const Ipp64fc*
    pTaps, int tapsLen, const Ipp16sc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRInit64fc_32sc(IppsFIRState64fc_32sc** ppState, const Ipp64fc*
    pTaps, int tapsLen, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsFIRInit_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps, int
    tapsLen, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
    pTaps, int tapsLen, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_64f(IppsFIRState_64f** ppState, const Ipp64f* pTaps, int
    tapsLen, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_32fc(IppsFIRState_32fc** ppState, const Ipp32fc* pTaps,
    int tapsLen, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit64fc_32fc(IppsFIRState64fc_32fc** ppState, const Ipp64fc*
    pTaps, int tapsLen, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_64fc(IppsFIRState_64fc** ppState, const Ipp64fc* pTaps,
    int tapsLen, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

Parameters

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *tapsLen*.
ppState	Pointer to the pointer to the FIR state structure to be created.
pBuffer	Pointer to the external buffer for FIR state structure.

Description

The function `ippsFIRInit` is declared in the `ipps.h` file. This function initializes a single-rate FIR filter state structure in the external buffer. The size of this buffer should be computed previously by calling the functions `ippsFIRGetStateSize`. The initialization function copies the taps from the *tapsLen* -length array *pTaps* into the state structure *ppState* of a single-rate filter. To scale integer taps, use the *tapsFactor* value.
The *tapsLen* -length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array contents is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

Note that the delay line length is different than that for direct FIR filters (where this length is doubled).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pTaps*, *ppState* or *pBuffer* is `NULL`.
`ippStsFIRLenErr`	Indicates an error when *tapsLen* is less than or equal to 0.

FIRMRInit

Initializes a multi-rate FIR filter state.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsFIRMRInit32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
    pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32s_16s32f(IppsFIRState32s_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32s* pDlyLine, Ipp8u* pBuffer);


IppStatus ippsFIRMRInit32sc_16sc(IppsFIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32sc_16sc32fc(IppsFIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32fc_16sc(IppsFIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64fc_16sc(IppsFIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64fc_32sc(IppsFIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsFIRMRInit_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor,
    int downPhase, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit_64f(IppsFIRState_64f** ppState, const Ipp64f* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase, const
    Ipp64f* pDlyLine, Ipp8u* pBuffer);


IppStatus ippsFIRMRInit_32fc(IppsFIRState_32fc** ppState, const Ipp32fc* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase, const
    Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64fc_32fc(IppsFIRState64fc_32fc** ppState, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit_64fc(IppsFIRState_64fc** ppState, const Ipp64fc* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor,
    int downPhase, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

Parameters

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).
downFactor	Downsampling factor.
downPhase	Phase for downsampled signal.
upFactor	Upsampling factor.
upPhase	Phase for upsampled signal.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is (*tapsLen* + *upFactor* - 1) / *upFactor*.
ppState	Pointer to the pointer to the FIR state structure.
pBuffer	Pointer to the external buffer for FIR state structure.

Description

The function `ippsFIRMRInit` is declared in the `ipps.h` file. This function initializes a multi-rate FIR filter state structure in the external buffer. The size of this buffer should be computed previously by calling the functions `ippsFIRMRGetStateSize`. The initialization functions copy the taps from the *tapsLen* -length array *pTaps* into the state structure *ppState*. To scale integer taps, use the *tapsFactor* value.
The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array contents is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

The function initializes the state structure *ppState* of a multi-rate filter, that is, a filter that internally upsamples and/or downsamples signals using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the `ippsSampleUp` function for more details). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The argument *upPhase* is the parameter, which determines where a non-zero sample lies within the *upFactor*-length block of the upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the ippsSampleDown function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

The argument *downPhase* is the parameter, which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array *pDlyLine* is not NULL, its length is defined as (*tapsLen* + *upFactor* - 1) / *upFactor*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pTaps*, *ppState* or *pBuffer* is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsFIRMRFactorErr	Indicates an error when *upFactor* (*downFactor*) is less than or equal to 0.
ippStsFIRMRPhaseErr	Indicates an error when *upPhase* (*downPhase*) is negative, or greater than or equal to *upFactor* (*downFactor*).

FIRGetStateSize, FIRMRGetStateSize

Returns the length of the FIR filter state structure.

Syntax

Case 1: Single-rate filter.

```
IppStatus ippsFIRGetStateSize_32s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64f(int tapsLen, int* pBufferSize);
```

```
IppStatus ippsFIRGetStateSize_64fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32s_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32s_16s32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32sc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32sc_16sc32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32fc(int tapsLen, int* pBufferSize);
```

Case 2: Multi-rate filter.

```
IppStatus ippsFIRMRGetStateSize_32f(int tapsLen, int upFactor, int downFactor,
    int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_32fc(int tapsLen, int upFactor, int downFactor,
    int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_64f(int tapsLen, int upFactor, int downFactor,
    int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_64fc(int tapsLen, int upFactor, int downFactor,
    int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32s_16s(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32s_16s32f(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32sc_16sc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32sc_16sc32fc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32f_16s(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32fc_16sc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
```

```
IppStatus ippsFIRMRGetStateSize64f_16s(int tapsLen, int upFactor,  int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_32s(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_32f(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_16sc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_32sc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_32fc(int tapsLen, int upFactor, int
    downFactor, int* pBufferSize);
```

Parameters

tapsLen	Number of elements in the array containing the tap values.
downFactor	Downsampling factor for the multi-rate filter.
upFactor	Upsampling factor for the multi-rate filter.
pBufferSize	Pointer to the computed buffer size value.

Description

The functions `ippsFIRGetStateSize` and `ippsFIRMRGetStateSize` are declared in the `ipps.h` file. These functions compute the size of the external buffer for a single-rate or multi-rate FIR filter state, respectively, and store the result in *pBufferSize*.

To compute the size of a buffer for the single-rate FIR filter state, the number of taps *tapsLen* only need be specified.
To compute the size of a buffer for the multi-rate FIR filter state, the number of taps *tapsLen* and upsampling/downsampling parameters *upFactor* and *downFactor* should be specified.
The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the `ippsSampleUp` function for more details). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the `ippsSampleDown` function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pBufferSize* pointer is NULL.
ippStsFIRLenErr	IIndicates an error if *nxTapsLen* is less than or equal to 0.
ippStsFIRMRFactorErr	Indicates an error if *upFactor* (*downFactor*) is less than or equal to 0.

FIRGetTaps

Retrieves the tap values from the FIR filter state structure.

Syntax

```
IppStatus ippsFIRGetTaps_32s(const IppsFIRState_32s* pState, Ipp32f* pTaps);
IppStatus ippsFIRGetTaps_32f(const IppsFIRState_32f* pState, Ipp32f* pTaps);
IppStatus ippsFIRGetTaps_64f(const IppsFIRState_64f* pState, Ipp64f* pTaps);
IppStatus ippsFIRGetTaps32f_16s(const IppsFIRState32f_16s* pState, Ipp32f*
    pTaps);
IppStatus ippsFIRGetTaps64f_16s(const IppsFIRState64f_16s* pState, Ipp64f*
    pTaps);
IppStatus ippsFIRGetTaps64f_32s(const IppsFIRState64f_32s* pState, Ipp64f*
    pTaps);
IppStatus ippsFIRGetTaps64f_32f(const IppsFIRState64f_32f* pState, Ipp64f*
    pTaps);
IppStatus ippsFIRGetTaps_32fc(const IppsFIRState_32fc* pState, Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps_64fc(const IppsFIRState_64fc* pState, Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps32fc_16sc(const IppsFIRState32fc_16sc* pState, Ipp32fc*
    pTaps);
IppStatus ippsFIRGetTaps64fc_16sc(const IppsFIRState64fc_16sc* pState, Ipp64fc*
    pTaps);
IppStatus ippsFIRGetTaps64fc_32sc(const IppsFIRState64fc_32sc* pState, Ipp64fc*
    pTaps);
IppStatus ippsFIRGetTaps64fc_32fc(const IppsFIRState64fc_32fc* pState, Ipp64fc*
    pTaps);
```

```
IppStatus ippsFIRGetTaps32s_16s32f(const IppsFIRState32s_16s* pState, Ipp32f*
    pTaps);
IppStatus ippsFIRGetTaps32sc_16sc32fc(const IppsFIRState32sc_16sc*  pState,
    Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps32s_16s(const IppsFIRState32s_16s* pState, Ipp32s*
    pTaps, int* pTapsFactor);
IppStatus ippsFIRGetTaps32sc_16sc(const IppsFIRState32sc_16sc* pState, Ipp32sc*
    pTaps, int* pTapsFactor);
```

Parameters

pState	Pointer to the FIR filter state structure.
pTaps	Pointer to the array containing the tap values.
pTapsFactor	Pointer to the scale factor for the taps of `Ipp32s` data type (for integer versions only).

Description

The function `ippsFIRGetTaps` is declared in the `ipps.h` file. This function copies the tap values from the initialized FIR state structure *pState* to the *tapsLen* -length array *pTaps*. To scale integer taps, use the *tapsFactor* value.

Before calling `ippsFIRGetTaps` function, the corresponding filter state structure must be initialized, for example by calling the function `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicatesanerrorwhenoneofthespecifiedpointersis`NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRSetTaps

Sets the tap values in the FIR filter state structure.

Syntax

```
IppStatus ippsFIRSetTaps_32s(const Ipp32s* pTaps, IppsFIRState_32s* pState);
IppStatus ippsFIRSetTaps_32f(const Ipp32f* pTaps, IppsFIRState_32f* pState);
```

```
IppStatus ippsFIRSetTaps_64f(const Ipp64f* pTaps, IppsFIRState_64f* pState);
IppStatus ippsFIRSetTaps32f_16s(const Ipp32f* pTaps, IppsFIRState32f_16s*
    pState);
IppStatus ippsFIRSetTaps64f_16s(const Ipp64f* pTaps, IppsFIRState64f_16s*
    pState);
IppStatus ippsFIRSetTaps64f_32s(const Ipp64f* pTaps, IppsFIRState64f_32s*
    pState);
IppStatus ippsFIRSetTaps64f_32f(const Ipp64f* pTaps, IppsFIRState64f_32f*
    pState);
IppStatus ippsFIRSetTaps_32fc(const Ipp32fc* pTaps, IppsFIRState_32fc* pState);
IppStatus ippsFIRSetTaps_64fc(const Ipp64fc* pTaps, IppsFIRState_64fc* pState);
IppStatus ippsFIRSetTaps32fc_16sc(const Ipp32fc* pTaps, IppsFIRState32fc_16sc*
    pState);
IppStatus ippsFIRSetTaps64fc_16sc(const Ipp64fc* pTaps, IppsFIRState64fc_16sc*
    pState);
IppStatus ippsFIRSetTaps64fc_32sc(const Ipp64fc* pTaps, IppsFIRState64fc_32sc*
    pState);
IppStatus ippsFIRSetTaps64fc_32fc(const Ipp64fc* pTaps, IppsFIRState64fc_32fc*
    pState);
IppStatus ippsFIRSetTaps32s_16s32f(const Ipp32f* pTaps, IppsFIRState32s_16s*
    pState);
IppStatus ippsFIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps, IppsFIRState32sc_16sc*
    pState);
IppStatus ippsFIRSetTaps32s_16s(const Ipp32s* pTaps, IppsFIRState32s_16s* pState,
    int tapsFactor);
IppStatus ippsFIRSetTaps32sc_16sc(const Ipp32sc* pTaps, IppsFIRState32sc_16sc*
    pState, int tapsFactor);
```

Parameters

pState	Pointer to the FIR filter state structure.
pTaps	Pointer to the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).

Description

The function `ippsFIRSetTaps` is declared in the `ipps.h` file. This function sets new tap values in the previously initialized FIR filter state structure *pState*. New tap values must be specified in the array *pTaps*. The length of the array *pTaps* must be equal to the *tapsLen* parameter value of the initialized filter state.

To scale integer taps, use the *tapsFactor* value.

Before calling `ippsFIRSetTaps` ffunction, the corresponding filter state structure must be initialized, for example by calling the function `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicatesanerrorwhenoneofthespecifiedpointersis`NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRGetDlyLine

Retrieves the delay line contents from the FIR filter state structuure.

Syntax

```
IppStatus ippsFIRGetDlyLine_32f(const IppsFIRState_32f* pState, Ipp32f*
    pDlyLine);
IppStatus ippsFIRGetDlyLine_64f(const IppsFIRState_64f* pState, Ipp64f*
    pDlyLine);
IppStatus ippsFIRGetDlyLine32s_16s(const IppsFIRState32s_16s* pState, Ipp16s*
    pDlyLine);
IppStatus ippsFIRGetDlyLine32f_16s(const IppsFIRState32f_16s* pState, Ipp16s*
    pDlyLine);
IppStatus ippsFIRGetDlyLine64f_16s(const IppsFIRState64f_16s* pState, Ipp16s*
    pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32s(const IppsFIRState64f_32s* pState, Ipp32s*
    pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32f(const IppsFIRState64f_32f* pState, Ipp32f*
    pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine_32fc(const IppsFIRState_32fc* pState, Ipp32fc*
    pDlyLine);
IppStatus ippsFIRGetDlyLine_64fc(const IppsFIRState_64fc* pState, Ipp64fc*
    pDlyLine);
IppStatus ippsFIRGetDlyLine32sc_16sc(const IppsFIRState32sc_16sc* pState,
    Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine32fc_16sc(const IppsFIRState32fc_16sc* pState,
    Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_16sc(const IppsFIRState64fc_16sc* pState,
    Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_32sc(const IppsFIRState64fc_32sc* pState,
    Ipp32sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_32fc(const IppsFIRState64fc_32fc* pState,
    Ipp32fc* pDlyLine);
```

Parameters

pState	Pointer to the FIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values.

Description

The function `ippsFIRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from the state structure *pState* and stores them into *pDlyLine*. The destination array *pDlyLine* contains samples in the reverse order as compared to the order of samples in the source vector.

Before calling `ippsFIRGetDlyLine`, the corresponding filter state structure must be initialized, for example by calling the function `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRSetDlyLine

Sets the delay line contents in the FIR filter state structure.

Syntax

```
IppStatus ippsFIRSetDlyLine_32f(IppsFIRState_32f* pState, const Ipp32f*
    pDlyLine);
IppStatus ippsFIRSetDlyLine_64f(IppsFIRState_64f* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsFIRSetDlyLine32s_16s(IppsFIRState32s_16s* pState, const Ipp16s*
    pDlyLine);
IppStatus ippsFIRSetDlyLine32f_16s(IppsFIRState32f_16s* pState, const Ipp16s*
    pDlyLine);
IppStatus ippsFIRSetDlyLine64f_16s(IppsFIRState64f_16s* pState, const Ipp16s*
    pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32s(IppsFIRState64f_32s* pState, const Ipp32s*
    pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32f(IppsFIRState64f_32f* pState, const Ipp32f*
    pDlyLine);

IppStatus ippsFIRSetDlyLine_32fc(IppsFIRState_32fc* pState, const Ipp32fc*
    pDlyLine);
IppStatus ippsFIRSetDlyLine_64fc(IppsFIRState_64fc* pState, const Ipp64fc*
    pDlyLine);
IppStatus ippsFIRSetDlyLine32sc_16sc(IppsFIRState32sc_16sc* pState, const
    Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine32fc_16sc(IppsFIRState32fc_16sc* pState, const
    Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_16sc(IppsFIRState64fc_16sc* pState, const
    Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32sc(IppsFIRState64fc_32sc* pState, const
    Ipp32sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32fc(IppsFIRState64fc_32fc* pState, const
    Ipp32fc* pDlyLine);
```

Parameters

pState	Pointer to the FIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values.

Description

The function `ippsFIRSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. The source array *pDlyLine* must contain samples in the reverse order as compared to the order of samples in the source vector.

Before calling `ippsFIRGetDlyLine` the corresponding filter state structure must be initialized, for example by calling the function `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIROne

Filters a single sample through a FIR filter.

Syntax

Case 1: Integer sample.

```
IppStatus ippsFIROne_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, IppsFIRStates_32s*
    pState, int scaleFactor);
IppStatus ippsFIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsFIRState32s_16s*
    pState, int scaleFactor);
IppStatus ippsFIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsFIRState32f_16s*
    pState, int scaleFactor);
IppStatus ippsFIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsFIRState64f_16s*
    pState, int scaleFactor);
IppStatus ippsFIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, IppsFIRState64f_32s*
    pState, int scaleFactor);
```

```
IppStatus ippsFIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
    IppsFIRState64fc_32sc* pState, int scaleFactor);
```

Case 2: Floating point sample.

```
IppStatus ippsFIROne_32f(Ipp32f src, Ipp32f* pDstVal, IppsFIRState_32f*
    pState);
IppStatus ippsFIROne_64f(Ipp64f src, Ipp64f* pDstVal, IppsFIRState_64f*
    pState);
IppStatus ippsFIROne64f_32f(Ipp32f src, Ipp32f* pDstVal, IppsFIRState64f_32f*
    pState);
IppStatus ippsFIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsFIRState_32fc*
    pState);
IppStatus ippsFIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal, IppsFIRState_64fc*
    pState);
IppStatus ippsFIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsFIRState64fc_32fc*
    pState);
```

Parameters

pState	Pointer to the FIR filter state structure.
src	Input sample to be filtered by the function `ippsFIROne`.
pDstVal	Pointer to the output sample filtered by the function `ippsFIROne`.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsFIROne` is declared in the `ipps.h` file. This function filters a single sample *src* through a single-rate filter and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted $x(n)$ and the taps are denoted $h(i)$.

The return value $y(n)$ is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Before calling `ippsFIROne`, initialize the filter state by calling `ippsFIRInitAlloc`. Specify the number of taps *tapsLen*, the tap values in *pTaps*, and the delay line values in *pDlyLine* beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIR

Filters a block of samples through a single-rate or multi-rate FIR filter.

Syntax

Case 1: Not-in-place operation on integer samples.

```
IppStatus ippsFIR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int numIters,
    IppsFIRState_32s* pState, int scaleFactor);
IppStatus ippsFIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
    IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
    IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
    IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int numIters,
    IppsFIRState64f_32s* pState, int scaleFactor);
IppStatus ippsFIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, IppsFIRState32fc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsFIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int
    numIters, IppsFIRState64fc_32sc* pState, int scaleFactor);
```

Case 2 Not-in-place operation on floating point samples.

```
IppStatus ippsFIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    IppsFIRState_32f* pState);
IppStatus ippsFIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
    IppsFIRState_64f* pState);
IppStatus ippsFIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
    IppsFIRState_32fc* pState);
IppStatus ippsFIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int numIters,
    IppsFIRState_64fc* pState);
IppStatus ippsFIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    IppsFIRState64f_32f* pState);
IppStatus ippsFIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
    IppsFIRState64fc_32fc* pState);
```

Case 3 In-place operation on integer samples.

```
IppStatus ippsFIR_32s_ISfs(Ipp32s* pSrcDst, int numIters, IppsFIRState_32s*
    pState, int scaleFactor);
IppStatus ippsFIR32s_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIR32f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_32s_ISfs(Ipp32s* pSrcDst, int numIters,
    IppsFIRState64f_32s* pState, int scaleFactor);
IppStatus ippsFIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
    IppsFIRState64fc_32sc* pState, int scaleFactor);
```

Case 4 In-place operation on floating point samples.

```
IppStatus ippsFIR_32f_I(Ipp32f* pSrcDst, int numIters, IppsFIRState_32f*
    pState);
IppStatus ippsFIR_64f_I(Ipp64f* pSrcDst, int numIters, IppsFIRState_64f*
    pState);
IppStatus ippsFIR64f_32f_I(Ipp32f* pSrcDst, int numIters, IppsFIRState64f_32f*
    pState);
IppStatus ippsFIR_32fc_I(Ipp32fc* pSrcDst, int numIters, IppsFIRState_32fc*
    pState);
IppStatus ippsFIR_64fc_I(Ipp64fc* pSrcDst, int numIters, IppsFIRState_64fc*
    pState);
IppStatus ippsFIR64fc_32fc_I(Ipp32fc* pSrcDst, int numIters, IppsFIRState64fc_32fc*
    pState);
```

Parameters

pState	Pointer to the FIR filter state structure.
pSrc	Pointer to the input array to be filtered by the function `ippsFIR`.
pDst	Pointer to the output array filtered by the function `ippsFIR`.
pSrcDst	Pointer to the input and output array (for the in-place operation) to be filtered by the function `ippsFIR`.
numIters	Parameter associated with the number of samples to be filtered by the function `ippsFIR`. For single-rate filters, the *numIters* samples in the input array are filtered and the resulting *numIters* samples are stored in the output array. For multi-rate filters, the (*numIters* * *downFactor*) samples in the input array are filtered and the resulting (*numIters* * *upFactor*) samples are stored in the output array.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsFIR` is declared in the `ipps.h` file. This function filters an input array *pSrc* or *pSrcDst* through a single-rate or multi-rate filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.

For single-rate filters, the *numIters* samples in the array *pSrc* or *pSrcDst* are filtered, and the resulting *numIters* samples are stored in the array *pDst* or *pSrcDst*. The results are identical to *numIters* consecutive calls to `ippsFIROne`.

In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters$$

After the function has performed calculations, it updates the delay line values stored in the state. For a single-rate filter, the *numIters* parameter defines the length of the source and destination arrays.

For multi-rate filters, `ippsFIR` filters (*numIters**downFactor*) input samples and stores the resulting (*numIters**upFactor*) samples in the output array. The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps. Thus, the function does not create an internal (*upFactor**srcLen*)-size buffer to store the upsampling result.

Before calling `ippsFIR`, initialize the filter state by calling either `ippsFIRInitAlloc` for single-rate filter, or `ippsFIRMRInitAlloc` for multi-rate filter. Specify the number of taps *tapsLen*, the tap values in *pTaps*, and the delay line values in *pDlyLine* beforehand.

Example 6-7 illustrates single-rate filtering with the function `ippsFIR_32f`.

Example 6-8 illustrates the using of multi-rate filter functions for vector interpolation (resizing).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicatesanerrorwhenoneofthespecifiedpointersis`NULL`.
`ippStsSizeErr`	Indicates an error when *numIters* is less or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect..

Example 6-7 Single-Rate Filtering with the `ippsFIR` Function

```
IppStatus fir(void) {
#undef NUMITERS
#define NUMITERS 150
     int n;
     IppStatus status;
     IppsIIRState_32f *ictx;
     IppsFIRState_32f *fctx;
     Ipp32f *x = ippsMalloc_32f(NUMITERS),
```

Example 6-7 Single-Rate Filtering with the ippsFIR Function

```
                *y = ippsMalloc_32f(NUMITERS),
                *z = ippsMalloc_32f(NUMITERS);
        const float taps[] = {
            0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
           0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        };
        for (n =0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *0.2);
        ippsIIRInitAlloc_32f( &ictx, taps, 10, NULL );
        ippsFIRInitAlloc_32f( &fctx, taps, 11, NULL );
        status = ippsIIR_32f( x, y, NUMITERS, ictx);
        printf_32f("IIR 32f output + 120 =", y+120, 5, status);
        ippsIIRFree_32f(ictx);
        status = ippsFIR_32f( x, z, NUMITERS, fctx );
        printf_32f("FIR 32f output + 120 =", z+120, 5, status);
        ippsFIRFree_32f(fctx);
        ippsFree(z);
        ippsFree(y);
        ippsFree(x);
        return status;
}
Output:
    IIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
    FIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
Matlab* Analog:
    >> F = 0.2; N = 150; n = 0:N-1; x = sin(2*pi*n*F);
    y = filter(fir1(10,0.15),1,x); y(121:125)
```

Example 6-8 Using Multi-Rate FIR Functions

Input vector (src) is:

```
x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,….
```

```
Taps are (tapsLen=8 for example):
t0,t1,t2,t3,t4,t5,t6,t7
```

We want to resize the input vector in 2/3 proportion, so upsample factor for MR FIR is equal to 2 and downsample factor is equal to 3:

```
Upsample operation:
x0,0,x1,0,x2,0,x3,0,x4,0,x5,0,x6,0,x7,0,x8,0,x9,0,….

Interpolation (filtering):
y0 = x0*t0+0*t1+x1*t2+0*t3+x2*t4+0*t5+x3*t6+0*t7 = (optimized) =
x0*t0+x1*t2+x2*t4+x3*t6
y1 = 0*t0+x1*t1+0*t2+x2*t3+0*t4+x3*t5+0*t6+x4*t7 = (optimized) =
x1*t1+x2*t3+x3*t5+x4*t7
y2 = x1*t0+0*t1+x2*t2+0*t3+x3*t4+0*t5+x4*t6+0*t7 = (optimized) =
x1*t0+x2*t2+x3*t4+x4*t6
```

Example 6-8 Using Multi-Rate FIR Functions

```
y3 = 0*t0+x2*t1+0*t2+x3*t3+0*t4+x4*t5+0*t6+x5*t7 = (optimized) =
x2*t1+x3*t3+x4*t5+x5*t7
y4 = x2*t0+0*t1+x3*t2+0*t3+x4*t4+0*t5+x5*t6+0*t7 = (optimized) =
x2*t0+x3*t2+x4*t4+x5*t6
y5 = 0*t0+x3*t1+0*t2+x4*t3+0*t4+x5*t5+0*t6+x6*t7 = (optimized) =
x2*t1+x3*t3+x4*t5+x5*t7
as a result after this operation y vector is 2x times longer than x

Downsample operation:
dst = y0, y3, y6,... (optimized version doesn't calculate y1,y2,etc.) -
                     so dst will have 2/3 length in comparison with src with interpolation
                     that is defined by taps coefficients (so it can be cubic, supersampling,
                      lanczos or any other spline polynomial)

"C" analogue:

#define n 99

IppsFIRState_32f *pState;
Ipp32f pTaps[8] = { t0,t1,t2,t3,t4,t5,t6,t7};
int tapsLen = 8;
int upFactor = 2;
int upPhase = 0;
int downFactor = 3;
int downPhase = 0;
Ipp32f *pDlyLine = NULL;
Ipp32f *pSrc[n] = { x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,….};
Ipp32f *pDst[2*n/3] ;
int numIters = n;
IppStatus status;

status = ippsFIRMRInitAlloc_32f ( &pState, pTaps, tapsLen, upFactor, upPhase,
                                  downFactor, downPhase, pDlyLine);
if( ippStsNoErr != status) return error;
status = ippsFIR_32f( pSrc, pDst, numIters, pState);
// pDst will contain y0, y3, y6,… according to the above formulas;
if( ippStsNoErr != status) return error;
status = ippsFIRFree_32f( pState );
if( ippStsNoErr != status) return error;
.......
```

FIROne_Direct

Directly filters a single sample through a FIR filter.

Syntax

Case 1: Not-in-place operation on integer sample.

```
IppStatus ippsFIROne_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const Ipp16s*
    pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIROne32f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
    Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIROne64f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
    Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIROne64f_Direct_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, const
    Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIROne32fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
    Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIROne64fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
    Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIROne64fc_Direct_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal, const
    Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIROne32s_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
    Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine,  int*
    pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32sc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
```

Case 2: Not-in-place operation on floating point sample.

```
IppStatus ippsFIROne_Direct_32f(Ipp32f src, Ipp32f* pDstVal, const Ipp32f*
    pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIROne_Direct_64f(Ipp64f src, Ipp64f* pDstVal, const Ipp64f*
    pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal, const Ipp32fc*
    pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64fc(Ipp64fc src, Ipp64fc* pDstVal, const Ipp64fc*
    pTaps, int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne64f_Direct_32f(Ipp32f src, Ipp32f* pDstVal, const Ipp64f*
    pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne64fc_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal, const
    Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
```

Case 3: In-place operation on integer sample.

```
IppStatus ippsFIROne_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp16s* pTaps,
    int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32f_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp32f*
    pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp64f*
    pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_32s_ISfs(Ipp32s* pSrcDstVal, const Ipp64f*
    pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp32fc*
    pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp64fc*
    pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDstVal, const Ipp64fc*
    pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32s_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp32s*
    pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine, int* pDlyLineIndex,
    int scaleFactor);
IppStatus ippsFIROne32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp32sc*
    pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int* pDlyLineIndex,
    int scaleFactor);
```

Case 4: In-place operation on floating point sample.

```
IppStatus ippsFIROne_Direct_32f_I(Ipp32f* pSrcDstVal, const Ipp32f* pTaps, int
    tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64f_I(Ipp64f* pSrcDstVal, const Ipp64f* pTaps, int
    tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIROne_Direct_32fc_I(Ipp32fc* pSrcDstVal, const Ipp32fc* pTaps,
    int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64fc_I(Ipp64fc* pSrcDstVal, const Ipp64fc* pTaps,
    int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne64f_Direct_32f_I(Ipp32f* pSrcDstVal, const Ipp64f* pTaps,
    int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne64fc_Direct_32fc_I(Ipp32fc* pSrcDstVal, const Ipp64fc*
    pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
```

Parameters

src	Input sample to be filtered by the function.
pDstVal	Pointer to the output sample filtered by the function.
pSrcDstVal	Pointer to the input and output sample for in-place operation.
pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
pTapsQ15	Pointer to the array containing the tap values, represented in Q0.15 format. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *2*tapsLen*. Note that the delay line length is different than that for FIR filters using state structure.
pDlyLineIndex	Pointer to the current delay line index.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsFIROne_Direct` is declared in the `ipps.h` file. This function directly filters a single sample *src* or *pSrcDstVal* through a single-rate filter and stores the result in *pDstVal* or *pSrcDstVal*.

The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps, the *tapsFactor* value is used.

The set of *tapsLen* input samples is copied twice to the *2*tapsLen*-length array *pDlyLine*. Double length of the delay line in direct FIR filters is used to improve filter performance by decreasing the number of sample copyings. The current delay line index is specified in the *pDlyLineIndex*.

The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)* and the taps are denoted *h(i)*.

The return value $y(n)$ is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.

FIR_Direct

Directly filters a block of samples through a single-rate FIR filter.

Syntax

Case 1: Not-in-place operation on integer samples.

```
IppStatus ippsFIR_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp16s* pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s*
    pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc*
    pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

Case 2: Not-in-place operation on floating point samples.

```
IppStatus ippsFIR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
    const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
    const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int numIters,
    const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int*
    pDlyLineIndex);
```

Case 3: In-place operation on integer samples.

```
IppStatus ippsFIR_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const Ipp16s*
    pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
```

```
IppStatus ippsFIR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);
IppStatus ippsFIR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int
    scaleFactor);


IppStatus ippsFIR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int*
    pDlyLineIndex, int scaleFactor);
```

Case 4: In-place operation on floating point samples.

```
IppStatus ippsFIR_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp32f*
    pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64f_I(Ipp64f* pSrcDst, int numIters, const Ipp64f*
    pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const Ipp32fc*
    pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters, const Ipp64fc*
    pTaps, int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp64f*
    pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
```

Parameters

pSrc	Pointer to the input array to be filtered.
pDst	Pointer to the output array.
pSrcDst	Pointer to the input and output array for the in-place operation.
numIters	Number of samples in the input array to be filtered.

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
pTapsQ15	Pointer to the array containing the tap values, represented in Q0.15 format. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *2*tapsLen*. Note that the delay line length is different than that for FIR filters using state structure.
pDlyLineIndex	Pointer to the current delay line index.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsFIR_Direct` is declared in the `ipps.h` file. This function filters an input array *pSrc* or *pSrcDst* containing *numIters* samples through a single-rate filter, and stores the resulting *numIters* samples in *pDst* or *pSrcDst*, respectively. The results are identical to *numIters* consecutive calls to `ippsFIROne_Direct`.
The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used.
The set of *tapsLen* input samples is copied to the *2*tapsLen*-length array *pDlyLine*. Double length of the delay line in direct FIR filters is used to improve filter performance by decreasing the number of sample copyings. The current delay line index is specified in the *pDlyLineIndex*.
The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsFIRLenErr`	Indicates an error when *tapsLen* is less than or equal to 0.

ippStsSizeErr	Indicates an error when *numIters* is less than or equal to 0.

FIRMR_Direct

Directly filters a block of samples through a multi-rate FIR filter.

Syntax

Case 1: Not-in-place operation on integer samples.

```
IppStatus ippsFIRMR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp16s* pDlyLine, int
    scaleFactor);
IppStatus ippsFIRMR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp16sc* pDlyLine, int
    scaleFactor)
```

Case 2: Not-in-place operation on floating point samples.

```
IppStatus ippsFIRMR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
    const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
    int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
    const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
    int downPhase, Ipp64f* pDlyLine);
IppStatus ippsFIRMR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
    numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32fc* pDlyLine);
IppStatus ippsFIRMR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp32fc* pDlyLine);
IppStatus ippsFIRMR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
    numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp64fc* pDlyLine);
```

Case 3: In-place operation on integer samples.

```
IppStatus ippsFIRMR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32sc* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
    Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
    Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
    downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
```

Case 4: In-place operation on floating point samples.

```
IppStatus ippsFIRMR_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp32f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const
    Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR_Direct_64f_I(Ipp64f* pSrcDst, int numIters, const Ipp64f*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp64f* pDlyLine);;)
IppStatus ippsFIRMR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
    Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32fc* pDlyLine);
IppStatus ippsFIRMR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp32fc* pDlyLine);
IppStatus ippsFIRMR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters, const
    Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
    downPhase, Ipp64fc* pDlyLine);
```

Parameters

pSrc	Pointer to the input array to be filtered.
pDst	Pointer to the output array.
pSrcDst	Pointer to the input and output array for the in-place operation.
numIters	Parameter associated with the number of samples to be filtered by the function. The (*numIters* * *downFactor*) samples of the input array are filtered and the resulting (*numIters* * *upFactor*) samples are stored in the output array
pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of `Ipp32s` data type.

pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is (*tapsLen* + *upFactor* - 1) / *upFactor*.
upFactor	Factor for upsampling the multi-rate signals.
downFactor	Factor for downsampling the multi-rate signals.
upPhase	Phase for upsampling the multi-rate signals.
downPhase	Phase for downsampling the multi-rate signals.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsFIRMR_Direct` is declared in the `ipps.h` file. This function filters an input array *pSrc* or *pSrcDst* through a multi-rate filter, and stores the resulting samples in *pDst* or *pSrcDst*, respectively. The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used. The array *pDlyLine* specifies the delay line values. The input array contains (*numIters***downFactor*) samples, and the output array stores the resulting (*numIters***upFactor*) samples.
The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the `ippsSampleUp` function for more details). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The argument *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see description of the `ippsSampleDown` function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The argument *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response. The length of the delay line array *pDlyLine* is defined as (*tapsLen* + *upFactor* - 1) / *upFactor*.
The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsSizeErr	Indicates an error when *numIters* is less than or equal to 0.
ippStsFIRMRFactorErr	Indicates an error when *upFactor* (*downFactor*) is less than or equal to 0.
ippStsFIRMRPhaseErr	Indicates an error when *upPhase* (*downPhase*) is negative, or greater than or equal to *upFactor* (*downFactor*).

FIRSparseInit

Initializes a sparse FIR filter structure.

Syntax

```
IppStatus ippsFIRSparseInit_32f(IppsFIRSparseState_32f** ppState,
    const Ipp32f* pNZTaps, const Ipp32s* pNZTapPos, int nzTapsLen, const
    Ipp32f* pDlyLine, Ipp8u* pBuffer);
```

Parameters

pNZTaps	Pointer to the array containing the non-zero tap values. The number of elements in the array is *nzTapsLen*.
pNZTapPos	Pointer to the array containing positions of the non-zero tap values. The number of elements in the array is *nzTapsLen*.
nzTapsLen	Number of elements in the array with non-zero tap values.
pDlyLine	Pointer to the array containing the delay line values.
ppState	Double pointer to the sparse FIR state structure.
pBuffer	Pointer to the external buffer for the sparse FIR state structure.

Description

The functions `ippsFIRSparseInit` is declared in the `ipps.h` file. This function initializes a sparse FIR filter state structure *pState* in the external buffer *pBuffer*. The size of this buffer should be computed previously by calling the function `ippsFIRSparseGetStateSize`. The initialization function copies the values of filter coefficients from the array *pNZTaps* containing *nzTapsLen* non-zero taps and their

positions from the array *pNZTapPos* into the state structure *pState*.
The array *pDlyLine* specifies the delay line values. The number of elements in this array is *pNZTapPos*[*nzTapsLen* -1]. If the pointer to the array *pDlyLine* is not NULL, the array contents are copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

CAUTION. The values of *nzTapsLen* and *pNZTapPos*[*nzTapsLen* -1] should be equal to those specified in the ippsFIRSparseGetStateSize function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers *ppState*, *pNZTaps*, *pNZTapPos*, or *pBuffer* is NULL.
ippStsFIRLenErr	Indicates an error if *nxTapsLen* is less than or equal to 0.
ippStsSparseErr	Indicates an error if positions of the non-zero taps are not in ascending order, or are negative or repetitive.

FIRSparseGetStateSize

Computes the size of the external buffer for the sparse FIR filter structure.

Syntax

```
IppStatus ippsFIRSparseGetStateSize_32f(int nzTapsLen, int order, int*
    pStateSize);
```

Parameters

nzTapsLen	Number of elements in the array containing the non-zero tap values.
order	Order of the sparse FIR filter.
pStateSize	Pointer to the computed value of the external buffer.

Description

The function ippsFIRSparseGetStateSize is declared in the ipps.h file. This function computes the size of the external buffer for a sparse FIR filter structure that is required for the function ippsFIRSparseInit. Computation is based on the specified number of non-zero filter coefficients *nzTapsLen* and filter order *order* that is equal to the number of elements in the delay line *pNZTapPos*[*nzTapsLen* -1] (see description of the function ippsFIRSparseInit). The result value is stored in the *pStateSize*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pStateSize* pointer is NULL.
ippStsFIRLenErr	IIndicates an error if *nzTapsLen* or *order* is less than or equal to 0; or *nzTapsLen* is more than *order*.

FIRSparse

Filters a block of samples through a sparse FIR filter.

Syntax

```
IppStatus ippsFIRSparse_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    IppsFIRSparseState_32f* pState);
```

Parameters

pState	Pointer to the sparse FIR filter state structure.
pSrc	Pointer to the input array.
pDst	Pointer to the output array.
len	Number of elements that will be filtered.

Description

The function ippsFIRSparse is declared in the ipps.h file. This function applies the sparse FIR filter to the *len* elements of the input array *pSrc*, and stores the results in *pDst*. The filter parameters - the number of non-zero taps *nzTapsLen*, their values

pNZTaps and their positions *pNZTapPos*, and the delay line values *pDlyLine* - are specified in the sparse FIR filter structure *pState* that should be previously initialized by calling the function ippsFIRSparseInit.

In the following definition of the sparse FIR filter, the sample to be filtered is denoted *x(n)*, the non-zero taps are denoted *pNZTaps(i)*, their positions are denoted *pNZTapPos(i)* and the return value is *y(n)*.

The return value *y(n)* is defined by the formula for a sparse FIR filter:

$$y(n) = \sum_{i=0}^{nzTapsLen-1} pNZTaps(i) \cdot x(n - pNZTapPos(i)), \quad 0 \le n < len$$

After the function has performed calculations, it updates the delay line values stored in the state.

Example 6-9 shows how to use the sparse FIR filter functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *len* is less or equal to 0.

Example 6-9 Using the Sparse FIR Filter Functions

```
int buflen;
Ipp8u *buf;
int nzTapsLen = 5; //number of non-zero taps
Ipp32f nzTaps [] = {0.5, 0.4, 0.3, 0.2, 0.1};  //non-zero taps values
Ipp32s nzTapsPos[] = {0, 10, 20, 30, 40};  //non-zero tap positions
IppsFIRSparseState_32f* firState;
Ipp32f *src, *dst;

/* ........................... */

ippsFIRSparseGetStateSize_32f(nzTapsLen, nzTapsPos [nzTapsLen - 1],
&buflen);
buf = ippsMalloc_8u(buflen);
ippsFIRSparseInit_32f(&firState, nzTaps, nzTapsPos, nzTapsLen, NULL,
buf);

/* .... initializing src somehow .... */

ippsFIRSparse_32f(src, dst, len, firState);
```

Example 6-9 Using the Sparse FIR Filter Functions

```
/*dst[i]=src[i]*0.5 + src[i-10]*0.4 + src[i-20]*0.3 + src[i-30]*0.2 +
src[i-40]*0.1 */

/* ........................... */

ippsFree(buf);
```

FIR Filter Coefficient Generating Functions

The functions described in this section compute coefficients (tap values) for different FIR filters by windowing the ideal infinite filter coefficients.

FIRGenLowpass

Computes lowpass FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenLowpass_64f(Ipp64f rFreq, Ipp64f* taps, int tapsLen,
    IppWinType winType, IppBool doNormal);
```

Parameters

rFreq	Normalized cut-off frequency, should be in the range (0, 0.5).
pTaps	Pointer to the array where computed tap values are stored. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values; should be equal or greater than 5.
winType	Values of this argument specify what type of window will be used in computations. The *winType* must have one of the following values: `ippWinBartlett` Bartlett window; `ippWinBlackman` Blackman window; `ippWinHamming` Hamming window; `ippWinHann` Hann window.
doNormal	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The *doNormal* must have one of the following values: `ippTrue` The function computes normalized sequence of coefficients. `ippFalse` The function computes non-normalized sequence of coefficients.

Description

The function `ippsFIRGenLowpass` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for lowpass FIR filter with the cut-off frequency *rFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Example 6-10 shows how the function `ippsFIRGenLowpass` can be used.

Return Values

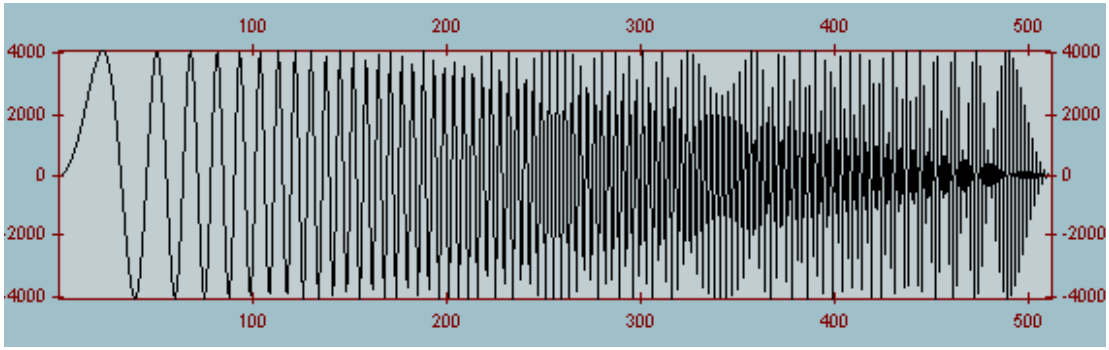
`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTaps* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when the *tapsLen* is less than 5, or *rFreq* is out of range.

Example 6-10 Using the Function `ippsFIRGenLowpass`

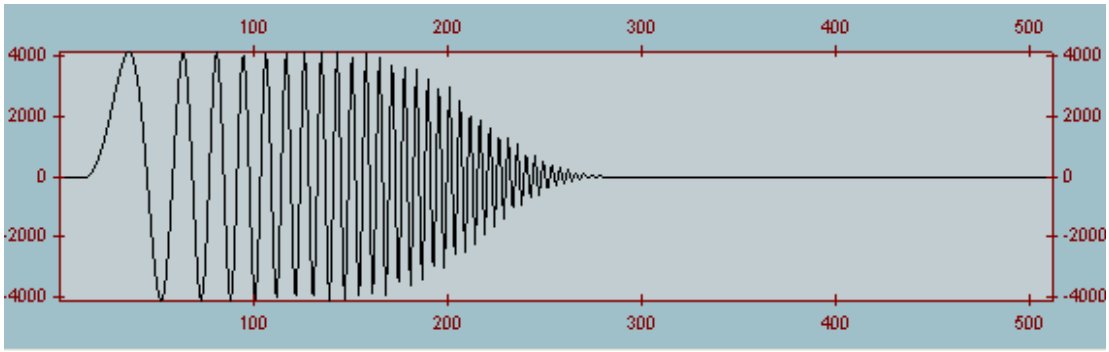
```
void func_firgenlowpass()
{
    int len = 512;
    Ipp64f pDst[512];
    Ipp64f magn = 4095;
    Ipp64f rFreq = 0.2;
    int tapslen = 27;
    int numIters = 512;
    IppsFIRState_64f* pState;
    IppStatus st;


    Ipp64f* FIRDst = ippsMalloc_64f(512*sizeof(Ipp64f));
    Ipp64f* taps = ippsMalloc_64f(tapslen*sizeof(Ipp64f));
    Ipp64f* pDL = ippsMalloc_64f(tapslen*sizeof(Ipp64f));

    ippsZero_64f(pDL,tapslen);
    //generate source vector
    ippsVectorJaehne_64f(pDst, len, magn);// create a Jaehne vector
    //computes tapsLen coefficients for lowpass FIR filter
    ippsFIRGenLowpass_64f(rFreq, taps, tapslen, ippWinBartlett, ippTrue);
    ippsFIRInitAlloc_64f(&pState, taps,tapslen, pDL);
    //filter an input vector
    ippsFIR_64f(pDst, FIRDst, numIters, pState);
}
Result:
```

Example 6-10 Using the Function `ippsFIRGenLowpass`

initial signal



filtered signal



FIRGenHighpass

Computes highpass FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenHighpass_64f(Ipp64f rFreq, Ipp64f* taps, int tapsLen,
    IppWinType winType, IppBool doNormal);
```

Parameters

rFreq	Normalized cut-off frequency, should be in the range (0, 0.5).
pTaps	Pointer to the array where computed tap values are stored. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values; should be equal or greater than 5.
winType	Values of this argument specify what type of window will be used in computations. The *winType* must have one of the following values: `ippWinBartlett` Bartlett window; `ippWinBlackman` Blackman window; `ippWinHamming` Hamming window; `ippWinHann` Hann window.
doNormal	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The *doNormal* must have one of the following values: `ippTrue` The function computes normalized sequence of coefficients. `ippFalse` The function computes non-normalized sequence of coefficients.

Description

The function `ippsFIRGenHighpass` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for highpass FIR filter the cut-off frequency *rFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTaps* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when the *tapsLen* is less than 5, or *rFreq* is out of the range.

FIRGenBandpass

Computes bandpass FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenBandpass_64f(Ipp64f rLowFreq, Ipp64f rHighFreq, Ipp64f*
    pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

Parameters

rLowFreq	Normalized low cut-off frequency, should be in the range (0, 0.5) and less than *rHighFreq*.
rHighFreq	Normalized high cut-off frequency, should be in the range (0, 0.5) and greater than *rLowFreq*.
pTaps	Pointer to the array where computed tap values are stored. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values; should be equal or greater than 5.
winType	Values of this argument specify what type of window will be used in computations. The *winType* must have one of the following values: `ippWinBartlett` Bartlett window; `ippWinBlackman` Blackman window; `ippWinHamming` Hamming window; `ippWinHann` Hann window.
doNormal	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The *doNormal* must have one of the following values: `ippTrue` The function computes normalized sequence of coefficients. `ippFalse` The function computes non-normalized sequence of coefficients.

Description

The function `ippsFIRGenBandpass` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for bandpass FIR filter with the cut-off frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType*

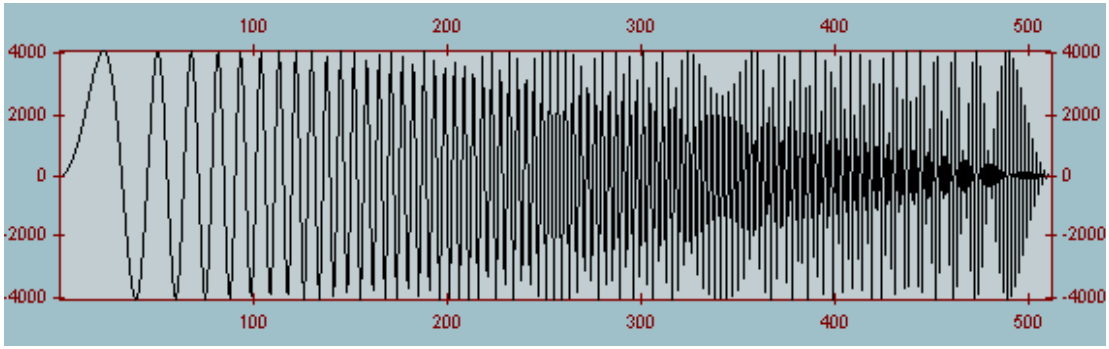
specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Example 6-11 shows how the function `ippsFIRGenBandpass` can be used.

Example 6-11 Using the Function `ippsFIRGenBandpass`

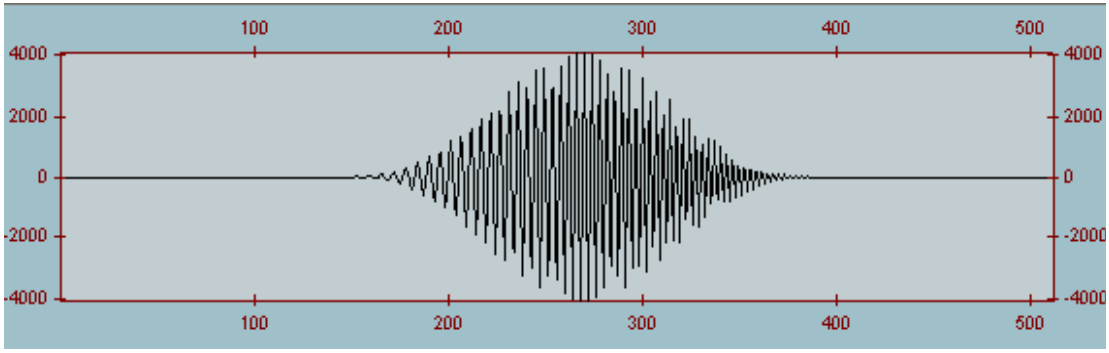
```
void func_BandPass()
{
    int len = 512;
    Ipp64f pDst[512];
    Ipp64f magn = 4095;
    Ipp64f rLowFreq = 0.2;
    Ipp64f rHighFreq = 0.3;

    int tapslen = 27;
    int numIters = 512;
    IppsFIRState_64f* pState;
    IppStatus st;

    Ipp64f* FIRDst = ippsMalloc_64f(512*sizeof(Ipp64f));
    Ipp64f* taps = ippsMalloc_64f(tapslen*sizeof(Ipp64f));
    Ipp64f* pDL = ippsMalloc_64f(tapslen*sizeof(Ipp64f));

    ippsZero_64f(pDL,tapslen);
    ippsVectorJaehne_64f(pDst, len, magn);// //generate source vector
    //computes tapsLen coefficients for bandstop  FIR filter
    ippsFIRGenBandpass_64f( rLowFreq, rHighFreq, taps, tapslen,
ippWinHamming, ippTrue);
    ////filter an input vector
    ippsFIRInitAlloc_64f(&pState, taps,tapslen, pDL);
    ippsFIR_64f(pDst, FIRDst, numIters, pState);
}
Result:
```

initial signal



Example 6-11 Using the Function `ippsFIRGenBandpass`

filtered signal



Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTaps* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when the *tapsLen* is less than 5, or *rLowFreq* is greater than or equal to *rHighFreq*, or one of the frequency parameters *rLowFreq* and *rHighFreq* is out of the range.

FIRGenBandstop

Computes bandstop FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenBandstop_64f(Ipp64f rLowFreq, Ipp64f rHighFreq, Ipp64f*
    pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

Parameters

rLowFreq	Normalized low cut-off frequency, should be in the range (0, 0.5) and less than *rHighFreq*.
rHighFreq	Normalized high cut-off frequency, should be in the range (0, 0.5) and greater than *rLowFreq*.
pTaps	Pointer to the array where computed tap values are stored. The number of elements in the array is *tapsLen*.

tapsLen	Number of elements in the array containing the tap values, should be equal or greater than 5.
winType	Values of this argument specify what type of window will be used in computations. The *winType* must have one of the following values: ippWinBartlett Bartlett window; ippWinBlackman Blackman window; ippWinHamming Hamming window; ippWinHann Hann window.
doNormal	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The *doNormal* must have one of the following values: ippTrue The function computes normalized sequence of coefficients. ippFalse The function computes non-normalized sequence of coefficients.

Description

The function ippsFIRGenBandstop is declared in the ipps.h file. This function computes *tapsLen* coefficients for bandstop FIR filter with the cut-off frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pTaps* pointer is NULL.
ippStsSizeErr	Indicates an error when the *tapsLen* is less than 5, or *rLowFreq* is greater than or equal to *rHighFreq*, or one of the frequency parameters *rLowFreq* and *rHighFreq* is out of the range.

Single-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a single-rate FIR LMS filter

- get and set the delay line values
- get the filter coefficients (taps) values
- perform filtering
- free dynamic memory allocated for the functions state

To use the single-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call ippsFIRLMSInitAlloc to allocate memory and initialize a single-rate FIR LMS filter.
2. Call ippsFIRLMSOne_Direct to make one iteration of FIR filter taps fitting with one input sample and/or call ippsFIRLMS to fit taps with a block of consecutive input samples.
3. Call ippsFIRLMSGetTaps to get the filter coefficients (taps). Call ippsFIRLMSGetDlyLine and ippsFIRLMSSetDlyLine to get and set the values in the delay line.
4. Call ippsFIRLMSFree to release dynamic memory associated with the FIR LMS filter.

FIRLMSInitAlloc

Allocates memory and initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.

Syntax

```
IppStatus ippsFIRLMSInitAlloc_32f(IppsFIRLMSState_32f** ppState, const Ipp32f*
    pTaps, int tapsLen, const Ipp32f* pDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSInitAlloc32f_16s(IppsFIRLMSState32f_16s** ppState, const
    Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine, int dlyLineIndex);
```

Parameters

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is 2**tapsLen*.
dlyLineIndex	Current index of the delay line.

ppState	Pointer to the pointer to the state structure.

Description

The function `ippsFIRLMSInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a single-rate FIR LMS filter state. The function `ippsFIRLMSInitAlloc` copies the taps from the *tapsLen* -length array *pTaps* into the state structure *ppState*. The 2**tapsLen* -length array *pDlyLine* specifies the delay line values. The current index of the delay line *pDlyLine* is defined by *dlyLineIndex*. If the pointer to the array *pDlyLine* or *pTaps* is `NULL`, then the corresponding values of the state structure are initialized to 0.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicatesanerrorwhenoneofthespecifiedpointersis`NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSFree

Closes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.

Syntax

```
IppStatus ippsFIRLMSFree_32f(IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMSFree32f_16s(IppsFIRLMSState32f_16s* pState);
```

Parameters

pState	Pointer to the FIR LMS filter state structure to be closed.

Description

The function `ippsFIRLMSFree` is declared in the `ipps.h` file. This function closes the FIR LMS filter state by freeing all memory associated with a filter state created by `ippsFIRLMSInitAlloc`. Call `ippsFIRLMSFree` after filtering is completed.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSGetTaps

Retrieves the tap values from the FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSGetTaps_32f(const IppsFIRLMSState_32f* pState, Ipp32f*
    pOutTaps);
IppStatus ippsFIRLMSGetTaps32f_16s(const IppsFIRLMSState32f_16s* pState,
    Ipp32f* pOutTaps);
```

Parameters

pState	Pointer to the FIR LMS filter state structure.
pOutTaps	Pointer to the array holding copies of the taps.

Description

The function `ippsFIRLMSGetTaps` is declared in the `ipps.h` file. This function copies the taps from the state structure *pState* to the *tapsLen* -length array *pOutTaps*. To set new taps in the state structure, create a new state using the function `ippsFIRLMSInitAlloc`.

Before calling the function `ippsFIRLMSGetTaps` the filter state structure must be initialized by the function `ippsFIRLMSInitAlloc`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSGetDlyLine

Retrieves the delay line contents from the FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState, Ipp32f*
    pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s* pState,
    Ipp16s* pDlyLine, int* pDlyLineIndex);
```

Parameters

pState	Pointer to the FIR LMS filter state structure.
pDlyLine	Pointer to the *tapsLen* -length array holding the delay line values.
pDlyLineIndex	Pointer to the array to store the current delay line index copied from the filter state structure.

Description

The function `ippsFIRLMSGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pDlyLine* and *pDlyLineIndex*, respectively.

Before calling the function `ippsFIRLMSGetDlyLine` the filter state structure must be initialized by the function `ippsFIRLMSInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicatesanerrorwhenoneofthespecifiedpointersis`NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSSetDlyLine

Sets the delay line contents in the FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState, Ipp32f*
    pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s* pState,
    Ipp16s* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine_32f(IppsFIRLMSState_32f* pState, const Ipp32f*
    pDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine32f_16s(IppsFIRLMSState32f_16s* pState, const
    Ipp16s* pDlyLine, int dlyLineIndex);
```

Parameters

pState	Pointer to the FIR LMS filter state structure.
pDlyLine	Pointer to the *tapsLen* -length array holding the delay line values.
dlyLineIndex	Initial index of the delay line to be stored in the filter state structure *pState*.

Description

The function `ippsFIRLMSSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from *pDlyLine*, and the current delay line index from *dlyLineIndex*, and stores them into the state structure *pState*.

Before calling the function `ippsFIRLMSSetDlyLine` the filter state structure must be initialized by the function `ippsFIRLMSInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMS

Filters an array through a FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMS_32f(const Ipp32f* pSrc, const Ipp32f* pRef, Ipp32f* pDst,
    int len, float mu, IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMS32f_16s(const Ipp16s* pSrc, const Ipp16s* pRef, Ipp16s*
    pDst, int len, float mu, IppsFIRLMSState32f_16s* pState);
```

Parameters

pState	Pointer to the FIR LMS filter state structure.
pSrc	Pointer to the input sample to be filtered.
pRef	Pointer to the reference signal
pDst	Pointer to the output signal
len	Number of elements in the array.
mu	Adaptation step.

Description

The function `ippsFIRLMS` is declared in the `ipps.h` file. This function filters an input array *pSrc* using an adaptive FIR LMS filter.

Each of *len* iterations performed by the function consists of two main procedures. First, `ippsLMS` filters the current sample of the input signal *pSrc* and stores the result in *pDst*. Next, the function updates the current taps using the reference signal *pRef*, the computed result signal *pDst*, and the adaptation step *mu*.

The filtering procedure can be described as a FIR filter operation:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Here the input sample to be filtered is denoted by $x(n)$, the taps are denoted by $h(i)$, and $y(n)$ is the return value.

The function updates the filter coefficients that are stored in the filter state structure *pState*. Updated filter coefficients are defined as

$$h_{n+1}(i) = h_n(i) + 2 \cdot mu \cdot errVal \cdot x(n - i)$$

where $h_{n+1}(i)$ denotes new taps, $h_n(i)$ denotes initial taps, *mu* and *errVal* are the adaptation step and adaptation error value, respectively. An adaptation error value *errVal* is computed inside the function as the difference between the output and reference signals.

Before using `ippsFIRLMS`, initialize the *pState* structure by calling the `ippsFIRLMSInitAlloc` function.

Example 6-12 demonstrates how the use of the functions `ippsFIRLMS_32f` to filter a signal sample.

Example 6-12 Filtering with the Function `ippsFIRLMS`

```
IppStatus firlms(void) {
     IppStatus st;
     Ipp32f taps = 0, x[LEN], y[LEN], mu = 0.03f;
     IppsFIRLMSState_32f* ctx;
     int i;
     /// no taps and no delay line from outside
     ippsFIRLMSInitAlloc_32f( &ctx, 0, 1, 0, 0 );
     /// make a const signal of amplitude 1 and noise it
     for(i=0; i<LEN; ++i) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
     st = ippsFIRLMS_32f( x, x+1, y, LEN-1, mu, ctx);
     /// get FIR LMS tap, it must be near to 1
     ippsFIRLMSGetTaps_32f(ctx, &taps);
     ippsFIRLMSFree_32f(ctx);
     printf_32f("FIR LMS tap fitted =", &taps, 1, st);
     return st;
}
Output:
    FIR LMS tap adapted = 0.986842
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSOne_Direct

Filters a single sample through a FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSOne_Direct_32f(Ipp32f src, Ipp32f refVal, Ipp32f* pDstVal,
    Ipp32f* pTapsInv, int tapsLen, float mu, Ipp32f* pDlyLine, int*
    pDlyLineIndex);
IppStatus ippsFIRLMSOne_Direct32f_16s(Ipp16s src, Ipp16s refVal, Ipp16s*
    pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu, Ipp16s* pDlyLine, int*
    pDlyLineIndex);
IppStatus ippsFIRLMSOne_DirectQ15_16s(Ipp16s src, Ipp16s refVal, Ipp16s*
    pDstVal, Ipp32s* pTapsInv, int tapsLen, int muQ15, Ipp16s* pDlyLine, int*
    pDlyLineIndex);
```

Parameters

src	Input sample to be filtered.
pDstVal	Pointer to the output sample.
refVal	Reference signal sample.
pTapsInv	Pointer to the array containing the FIR filter taps to be adapted. The tap values are stored in the array in the inverse order.
tapsLen	Number of elements in the array containing the tap values.
pDlyLine	Pointer to the array holding the delay line values.
pDlyLineIndex	Pointer to the current index of the delay line.
mu	Adaptation step.
muQ15	Integer version adaptation step.

Description

The function `ippsFIRLMSOne_Direct` is declared in the `ipps.h` file. This function performs directly a single iteration of FIR filter taps adaptation. The *tapsLen* -length array *pTapsInv* contains the FIR filter taps in the inverse order. The 2**tapsLen* -length array *pDlyLine* specifies the delay line values. The *pDlyLineIndex* array specifies the current index of the delay line. The output signal is stored in *pDstVal*.

NOTE. The adaptation error value can be computed as follows: *err*[*n*] = *refVal*[*n*] - **pDstVal*.

The function `ippsFIRLMSOne_Direct` performs a single iteration of FIR filter taps adaptation with the *mu* step value. The taps are floating-point numbers.

Set the taps to zero or to values close to calculated to speed up the process.

The function `ippsLMSOne_Direct` is to be called within cycle with the number of iterations equal to the number of input samples. You can decide what data is to be saved as a result of adaptation: either the filtered output signal or the adaptation error.

The function `ippsFIRLMSOne_DirectQ15` performs a single iteration of FIR filter taps adaptation with the *muQ15* step value. The taps are integer numbers. The adaptation step *muQ15* can be computed as follows:

muQ15 = (int)(*mu* *(1<<15)+0.5f)

Example 6-13 illustrates the use of the function `ippsFIRLMSOne_Direct` to adapt the FIR filter taps. After adaptation the taps are close to 1.0.

Example 6-13 Using the Function `ippsLMSOne_Direct`

```
IppStatus firlmsone(void) {
#define LEN 200
      IppStatus st = ippStsNoErr;
      Ipp32f taps = 0, dly[2] = {0};
      Ipp32f x[LEN], y[LEN], mu = 0.05f;
      int i, indx = 0;
      /// make a const signal of amplitude 1 and noise it
      for( i=0; i<LEN; ++i ) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
      for( i=0; i<LEN-1 && ippStsNoErr==st; ++i )
      st=ippsFIRLMSOne_Direct_32f( x[i], x[i+1], y+1+i, &taps, 1, mu,
          dly, &indx );
      printf_32f("FIRLMSOne tap adapted =", &taps, 1, st );
      return st;
}
Output:
```

Example 6-13 Using the Function ippsLMSOne_Direct

```
FIRLMSOne tap adapted = 0.993872
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsSizeErr	Indicates an error when *tapsLen* is less or equal to 0.

Multi-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a multi-rate FIR LMS filter
- get and set the delay line values
- get and set the filter coefficients (taps) values
- set the adaptation step value
- perform filtering
- update the filter coefficients using the result of the filter operation
- free dynamic memory allocated for the functions state

To use the multi-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call ippsFIRLMSMRInitAlloc to initialize a multi-rate FIR LMS filter.
2. Call ippsFIRLMSMRPutVal a required number of times to place the input values in the delay line.
3. Call ippsFIRLMSMROne to filter the samples in the delay line.
4. Call ippsFIRLMSMRUpdateTaps to update the taps using the value of adaptation error that is based on comparison between filtered and reference signals.
5. Call ippsFIRLMSMRGetTaps and ippsFIRLMSMRSetTaps to get and set the filter coefficients (taps). Call ippsFIRLMSMRGetDlyLine and ippsFIRLMSMRSetDlyLine to get and set the values in the delay line.
6. Call ippsFIRLMSMRFree to release dynamic memory associated with the FIR LMS filter.

FIRLMSMRInitAlloc

Allocates memory and initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.

Syntax

```
IppStatus ippsFIRLMSMRInitAlloc32s_16s(IppsFIRLMSMRState32s_16s** ppState,
    const Ipp32s* pTaps, int tapsLen, const Ipp16s* pDlyLine, int dlyLineIndex,
    int dlyStep, int updateDly, int mu);
IppStatus ippsFIRLMSMRInitAlloc32sc_16sc(IppsFIRLMSMRState32sc_16sc** ppState,
    const Ipp32sc* pTaps, int tapsLen, const Ipp16sc* pDlyLine, int
    dlyLineIndex, int dlyStep, int updateDly, int mu);
```

Parameters

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is *tapsLen*dlyStep + updateDly*.
dlyLineIndex	Current index of the delay line.
dlyStep	Multi-rate down factor applied to delay line values.
updateDly	Value of adaptation delay in samples.
mu	Adaptation step.
ppState	Pointer to the pointer to the filter state structure.

Description

The function `ippsFIRLMSMRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a multi-rate FIR LMS filter state. The function copies the filter coefficients from *tapsLen*-length array *pTaps* into the state structure *pState*. The array *pDlyLine* specifies the delay line values. The structure is initialized by the downsampling factor over the delay line *dlyStep*, by the adaptation delay value *updateDly*, and by the adaptation step value *mu*. The function returns the pointer in the output parameter *ppState* and also the operation status value.

The function uses copies of the tap values during the filtering procedure. The initial values passed to the function may be obtained from the previous filtering procedure.If the pointer *pTaps* is NULL, then the filter coefficient values are set to zero.

Copies of the input samples are used in the filtering procedure. Values in the delay line represent data in the same format as the input data to be filtered. If the pointer *pDlyLine* is NULL, then the filter delay line values are set to zero.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRFree

Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.

Syntax

```
IppStatus ippsFIRLMSMRFree32s_16s(IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRFree32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState);
```

Parameters

pState	Pointer to the multi-rate FIR LMS filter state structure to be closed.

Description

The function ippsFIRLMSMRFree is declared in the ipps.h file. This function closes the multi-rate FIR LMS filter state by freeing all memory associated with a filter state created by ippsFIRLMSMRInitAlloc. Call ippsFIRLMSMRFree after filtering is completed.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRSetMu

Sets the adaptation step.

Syntax

```
IppStatus ippsFIRLMSMRSetMu32s_16s(IppsFIRLMSMRState32s_16s* pState, const int
    mu);
IppStatus ippsFIRLMSMRSetMu32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState, const
    int mu);
```

Parameters

mu	New adaptation step
pState	Pointer to the filter state structure.

Description

The function `ippsFIRLMSMRSetMu` is declared in the `ipps.h` file. This function updates the adaptation step stored in *pState* with the new value *mu*.

Before calling the function `ippsFIRLMSMRSetMu` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMRUpdateTaps

Updates the filter coefficients using the adaptation error value.

Syntax

```
IppStatus ippsFIRLMSMRUpdateTaps32s_16s(Ipp32s errVal,
    IppsFIRLMSMRState32s_16s* pState);
```

```
IppStatus ippsFIRLMSMRUpdateTaps32sc_16sc(Ipp32sc errVal,
    IppsFIRLMSMRState32sc_16sc* pState);
```

Parameters

pState	Pointer to the filter state structure.
errVal	Adaptation error value.

Description

The function `ippsFIRLMSMRUpdateTaps` is declared in the `ipps.h` file. This function updates the filter coefficients that are stored in the filter state structure *pState*. It is assumed that the filter operation was performed and the adaptation error value *errVal* was computed before calling this function. The adaptation error value is computed outside the function as the difference between output and reference signals. Updated filter coefficients are defined as

$$h_{n+1}(i) = h_n(i) + mu \cdot errVal \cdot x(n - (i \cdot dlyStep) - updateDly)$$

where $h_{n+1}(i)$ denotes new taps, $h_n(i)$ denotes initial taps, and *mu*, *errVal* and *updateDly* are the adaptation step, adaptation error value and adaptation delay, respectively.
Before calling the function `ippsFIRLMSMRUpdateTaps` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetTaps

Retrieves tap values from the multi-rate FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSMRGetTaps32s_16s(IppsFIRLMSMRState32s_16s* pState, Ipp32s*
    pOutTaps);
```

```
IppStatus ippsFIRLMSMRGetTaps32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    Ipp32sc* pOutTaps);
```

Parameters

pState	Pointer to the filter state structure.
pOutTaps	Pointer to the array holding copies of the taps.

Description

The function `ippsFIRLMSMRGetTaps` is declared in the `ipps.h` file. This function copies the values of the filter coefficients stored in the filter state structure *pState* to the array pointed by the *pOutTaps* pointer.

Before calling the function `ippsFIRLMSMRGetTaps` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMRSetTaps

Sets tap values in the multi-rate FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSMRSetTaps32s_16s(IppsFIRLMSMRState32s_16s* pState, const
    Ipp32s* pInTaps);
IppStatus ippsFIRLMSMRSetTaps32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    const Ipp32sc* pInTaps);
```

Parameters

pState	Pointer to the filter state structure.
pInTaps	Pointer to the array holding new tap values.

Description

The function `ippsFIRLMSMRSetTaps` is declared in the `ipps.h` file. This function sets the the filter coefficients stored in the filter state structure *pState* to the new values stored in an array pointed by the *pInTaps* pointer. If the pointer is `NULL`, then the filter coefficients values are set to zero.

Before calling the function `ippsFIRLMSMRSetTaps` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicatesanerrorwhenoneofthespecifiedpointersis`NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetTapsPointer

Returns the pointer to the filter coefficients.

Syntax

```
IppStatus ippsFIRLMSMRGetTapsPointer32s_16s(IppsFIRLMSMRState32s_16s* pState,
    Ipp32s** ppTaps);
IppStatusippsFIRLMSMRGetTapsPointer32sc_16sc(IppsFIRLMSMRState32sc_16sc*pState,
    Ipp32sc** ppTaps);
```

Parameters

pState	Pointer to the filter state structure.
pTaps	Pointer to the variable that contains the pointer to the tap values.

Description

The function `ippsFIRLMSMRGetTapsPointer` is declared in the `ipps.h` file. This function writes the pointer to the filter coefficients stored in the filter state structure *pState* to the variable pointed by *pTaps*.

CAUTION. To get the pointer to tap values directly is faster than to copy them using the `ippsFIRLMSMRGetTaps` function, but this operation may be error-prone.

Before calling the function `ippsFIRLMSMRGetTapsPointer` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetDlyLine

Retrieves the delay line contents from the multi-rate FIR LMS filter state.

Syntax

```
IppStatus ippsFIRLMSMRGetDlyLine32s_16s(IppsFIRLMSMRState32s_16s* pState,
    Ipp16s* pOutDlyLine, int* pOutDlyLineIndex);
IppStatus ippsFIRLMSMRGetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    Ipp16sc* pOutDlyLine, int* pOutDlyLineIndex);
```

Parameters

pState	Pointer to the filter state structure.
pOutDlyLine	Pointer to the array containing the copies of delay line values. The number of elements in the array is *tapsLen*dlyStep+updateDly*.
pOutDlyLineIndex	Pointer to the array to store the current delay line index copied from *pState*.

Description

The function `ippsFIRLMSMRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pOutDlyLine* and *pOutDlyLineIndex*, respectively.

Before calling the function `ippsFIRLMSMRGetDlyLine` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMRSetDlyLine

Gets and sets the delay line contents of a multi-rate FIR LMS filter state.

Syntax

```
IppStatus ippsFIRLMSMRSetDlyLine32s_16s(IppsFIRLMSMRState32s_16s* pState, const
    Ipp16s* pInDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSMRSetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    const Ipp16sc* pInDlyLine, int dlyLineIndex);
```

Parameters

pState	Pointer to the filter state structure.
pInDlyLine	Pointer to the array containing the new delay line values.The number of elements in the array is *tapsLen*dlyStep+updateDly*.
dlyLineIndex	Initial index of the delay line to be stored in *pState* .

Description

The function `ippsFIRLMSMSetDlyLine` is declared in the `ipps.h` file. This function copies the new values stored in the *pInDlyLine* and corresponding delay line index from *dlyLineIndex* and stores them into the state structure *pState*. If the pointer *pInDlyLine* is `NULL`, the delay line values are set to zero.

Before calling the function `ippsFIRLMSMRSetDlyLine` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetDlyVal

Gets one delay line values from the specified position.

Syntax

```
IppStatus ippsFIRLMSMRGetDlyVal32s_16s(IppsFIRLMSMRState32s_16s* pState,
    Ipp16s* pOutVal, int index);
IppStatus ippsFIRLMSMRGetDlyVal32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    Ipp16sc* pOutVal, int index);
```

Parameters

pState	Pointer to the filter state structure.
pOutVal	Pointer to the copied delay line value.
index	Index of the required delay line value.

Description

The function `ippsFIRLMSMRGetDlyVal` is declared in the `ipps.h` file. This function copies from the filter state structure *pState* one value to the *pOurVal*. The position of this sample in the delay line is specified by *index* (which means that *index* iterations ago the sample was placed into the delay line).

Before calling the function `ippsFIRLMSMRGetDlyVal` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMRPutVal

Places the input value in the delay line.

Syntax

```
IppStatus ippsFIRLMSMRPutVal32s_16s(Ipp16s val, IppsFIRLMSMRState32s_16s*
    pState);
IppStatus ippsFIRLMSMRPutVal32sc_16sc(Ipp16sc val, IppsFIRLMSMRState32sc_16sc*
    pState);
```

Parameters

pState	Pointer to the filter state structure.
val	Value of the input sample.

Description

The function `ippsFIRLMSMRPutVal` is declared in the `ipps.h` file. This function places the value of the input sample *val* into the delay line, thus preparing the filter with the state structure *pState* for the filtering procedure.

Before calling the function `ippsFIRLMSMRPutVal` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMROne

Filters data placed in the delay line.

Syntax

```
IppStatus ippsFIRLMSMROne32s_16s(Ipp32s* pDstVal, IppsFIRLMSMRState32s_16s*
    pState);
```

```
IppStatus ippsFIRLMSMROne32sc_16sc(Ipp32sc* pDstVal, IppsFIRLMSMRState32sc_16sc*
    pState);
```

Parameters

pState	Pointer to the filter state structure.
pDstVal	Pointer to the output signal value.

Description

The function `FIRLSMMROne` is declared in the `ipps.h` file. This function filters the samples placed in the delay line using the filter coefficients stored in the filter state structure *pState*. The resulting value is placed in the *pDstVal*. The downsampling factor *dlyStep* defines the number of samples that are filtered. The filter coefficients are not updated.

The filtering procedure can be described as a FIR filter operation (here the input sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*):

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n - (i \cdot dlyStep))$$

Note that the function operates with values stored in the delay line that are copies of the input samples.
Before calling the function `ippsFIRLMSMROne`, initialize the filter state by calling `ippsFIRLMSMRInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMROneVal

Filters one input value.

Syntax

```
IppStatus ippsFIRLMSMROneVal32s_16s(Ipp16s val, Ipp32s* pDstVal,
    IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMROneVal32sc_16sc(Ipp16sc val, Ipp32sc* pDstVal,
    IppsFIRLMSMRState32sc_16sc* pState);
```

Parameters

pState	Pointer to the filter state structure.
pDstVal	Pointer to the output signal value.
val	Value of the input signal sample.

Description

The function `ippsFIRLMSMROneVal` is declared in the `ipps.h` file. This function places one input sample *val* into the delay line and filters it using the filter coefficients specified in the filter state structure *pState*. The result value is stored in the *pDstVal*.

Before calling the function `ippsFIRLMSMROneVal`, initialize the filter state by calling `ippsFIRLMSMRInitAlloc`.

Return Values

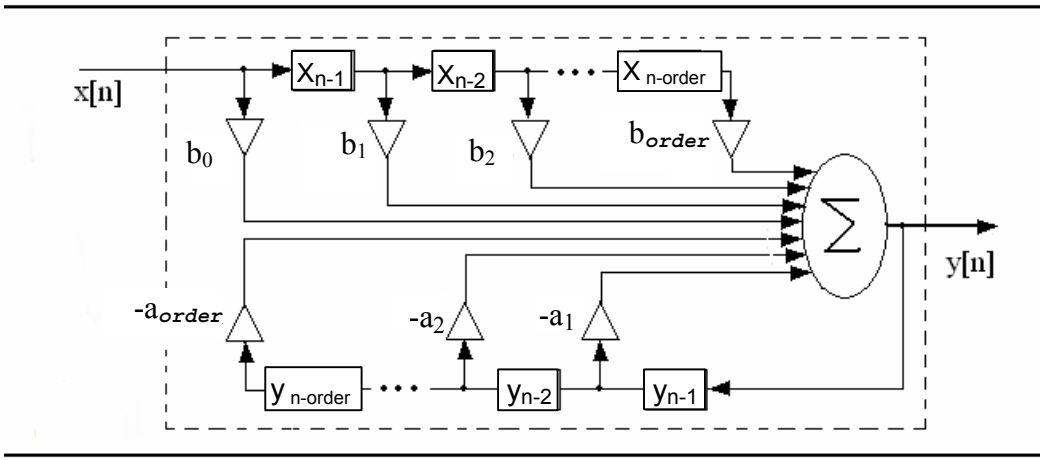
`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

IIR Filter Functions

The functions described in this section initialize an infinite impulse response (IIR) filter and perform filtering. Intel IPP supports two types of filters: arbitrary order filter and biquad filter.

Figure 6-1 shows the structure of an arbitrary order IIR filter.

Figure 6-1 Structure of an Arbitrary Order Filter



Here `x[n]` is a sample of the input signal, `y[n]` is a sample of the output signal, *order* is the filter order, and b_0, b_1, . . ., b_{order}, a_1, . . ., a_{order} are the reduced filter coefficients:

The output signal is computed by the following formula:

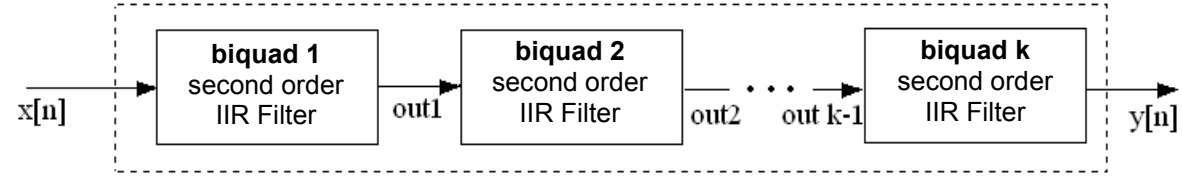
$$y[n] = \sum_{k=0}^{order} b_k \cdot x(n-k) - \sum_{k=1}^{order} a_k \cdot y(n-k)$$

Reduced coefficients are calculated as $a_k = \frac{A_k}{A_0}$ and $b_k = \frac{B_k}{A_0}$

where A_0, A_1,...A_{order}, B_0, B_1,...B_{order} are initial filter coefficients (taps).

A biquad IIR filter is a cascade of second-order filters. The Figure 6-2 illustrates the structure of the biquad filter with **k** cascades of second-order filters.

Figure 6-2 Structure of a BiQuad IIR Filter



To initialize and use an IIR filter, follow this general scheme:

1. Call `ippsIIRInitAlloc` to allocate memory and initialize the filter as an arbitrary order IIR filter, or call `ippsIIRInitAlloc_BiQuad` to allocate memory and initialize the filter as a cascade of biquads.
   Or alternatively call either `ippsIIRInit` to initialize the filter as an arbitrary order IIR filter in the external buffer, or `ippsIIRInit_BiQuad` to initialize the filter as a cascade of biquads in the external buffer. Size of the buffer can be computed by calling the functions `ippsIIRGetStateSize` or `ippsIIRGetStateSize_BiQuad`, respectively.
2. Call `ippsIIROne` repeatedly to filter a single sample through an IIR filter or call `ippsIIR` to filter consecutive samples at once.
3. Call `ippsIIRGetDlyLine` and `ippsIIRSetDlyLine` to get and set the delay line values in the IIR state structure.
4. Call `ippsIIRSetTaps` to set new tap values in the previously initialized filter state structure.
5. After all filtering is complete, call `ippsIIRFree` to release dynamic memory associated with the filter state structure created by `ippsIIRInitAlloc` or `ippsIIRInitAlloc_BiQuad`.

Alternatively, you may use the direct version of the functions. These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

IIRInitAlloc

Allocates memory and initializes the state structure for an arbitraryIIR filter.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsIIRInitAlloc32s_16s(IppsIIRState32s_16s** ppState, const Ipp32s*
    pTaps, int order, int tapsFactor, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32s_16s32f(IppsIIRState32s_16s** ppState, const
    Ipp32f* pTaps, int order, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32f_16s(IppsIIRState32f_16s** ppState, const Ipp32f*
    pTaps, int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_16s(IppsIIRState64f_16s** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32s(IppsIIRState64f_32s** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_16sc(IppsIIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int order, int tapsFactor, const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_16sc32fc(IppsIIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32fc_16sc(IppsIIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_16sc(IppsIIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32sc(IppsIIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsIIRInitAlloc_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps,
    int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32f(IppsIIRState64f_32f** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc_64f(IppsIIRState_64f** ppState, const Ipp64f* pTaps,
    int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_32fc(IppsIIRState_32fc** ppState, const Ipp32fc*
    pTaps, int order, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32fc(IppsIIRState64fc_32fc** ppState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc_64fc(IppsIIRState_64fc** ppState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine);
```

Parameters

pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1).
tapsFactor	Scale factor for the taps of integer data type.
order	Order of the IIR filter.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order*.
ppState	Pointer to the pointer to the IIR state structure.

Description

The function `ippsIIRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes an arbitrary IIR filter state. The initialization functions copy the taps from the array *pTaps* into the state structure *ppState*. The *tapsFactor* is used to scale integer tap values. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The 2*(*order* + 1) -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

$A_0 \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. Example 6-15 shows how to convert floating-point taps into integer data type.

Return Values

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory allocated.
ippStsNullPtrErr	Indicates an error when *pTaps* or *ppState* is NULL.
ippStsIIROrderErr	Indicates an error when *order* is less than or equal to 0.
ippStsDivByZeroErr	Indicates an error when A_0 is equal to 0.

IIRInitAlloc_BiQuad

Allocates memory and initializes the state structure for the biquad IIR filter.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsIIRInitAlloc32s_BiQuad_16s(IppsIIRState32s_16s** ppState, const
    Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32s_BiQuad_16s32f(IppsIIRState32s_16s** ppState,
    const Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32f_BiQuad_16s(IppsIIRState32f_16s** ppState, const
    Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_BiQuad_16s(IppsIIRState64f_16s** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_BiQuad_32s(IppsIIRState64f_32s** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc(IppsIIRState32sc_16sc** ppState,
    const Ipp32sc* pTaps, int numBq, int tapsFactor, const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc** ppState,
    const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32fc_BiQuad_16sc(IppsIIRState32fc_16sc** ppState,
    const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_16sc(IppsIIRState64fc_16sc** ppState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc64fc_BiQuad_32sc(IppsIIRState64fc_32sc** ppState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsIIRInitAlloc_BiQuad_32f(IppsIIRState_32f** ppState, const Ipp32f*
    pTaps, int numBq, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_BiQuad_32f(IppsIIRState64f_32f** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_64f(IppsIIRState_64f** ppState, const Ipp64f*
    pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc_BiQuad_32fc(IppsIIRState_32fc** ppState, const
    Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_32fc(IppsIIRState64fc_32fc** ppState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_64fc(IppsIIRState_64fc** ppState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

Parameters

pTaps	Pointer to the array containing the taps. The number of elements in the array is 6**numBq*.
tapsFactor	Scale factor for the taps of integer data type.
numBq	Number of cascades of biquads.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is 2**numBq*.
ppState	Pointer to the pointer to the biquad IIR state structure.

Description

The function `ippsIIRInitAlloc_BiQuad` is declared in the `ipps.h` file. This function allocates memory and initializes a biquad (BQ) IIR filter state. The initialization functions copy the taps from the array *pTaps* into the state structure *ppState*. The *tapsFactor* is used to scale integer tap values. The 2**numBq* -length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

A biquad IIR filter is defined by a cascade of biquads. The number of cascades of biquads is specified by the *numBq* value. The 6**numBq* -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}$, $B_{0,1}$, $B_{0,2}$, $A_{0,0}$, $A_{0,1}$, $A_{0,2}$; $B_{1,0}$, $B_{1,1}$, $B_{1,2}$, $A_{1,0}$, $A_{1,1}$, $A_{1,2}$; . . . $A_{numBq-1,2}$

$A_{n,0} \neq 0$, $B_{n,0} \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. Example 6-15 shows how to convert floating-point taps into integer data type.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsMemAllocErr`	Indicates an error when no memory allocated.
`ippStsNullPtrErr`	Indicates an error when *pTaps* or *ppState* is `NULL`.
`ippStsIIROrderErr`	Indicates an error when *numBq* is less than or equal to 0.
`ippStsDivByZeroErr`	Indicates an error when A_0, $A_{n,0}$ or $B_{n,0}$ is equal to 0.

IIRFree

Closes an IIR filter state.

Syntax

```
IppStatus ippsIIRFree_32f(IppsIIRState_32f* pState);
IppStatus ippsIIRFree_64f(IppsIIRState_64f* pState);
IppStatus ippsIIRFree_32fc(IppsIIRState_32fc* pState);
IppStatus ippsIIRFree_64fc(IppsIIRState_64fc* pState);
IppStatus ippsIIRFree32s_16s(IppsIIRState32s_16s* pState);
IppStatus ippsIIRFree32sc_16sc(IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRFree32f_16s(IppsIIRState32f_16s* pState);
IppStatus ippsIIRFree32fc_16sc(IppsIIRState32fc_16sc* pState);
IppStatus ippsIIRFree64f_16s(IppsIIRState64f_16s* pState);
IppStatus ippsIIRFree64f_32s(IppsIIRState64f_32s* pState);
IppStatus ippsIIRFree64f_32f(IppsIIRState64f_32f* pState);
```

```
IppStatus ippsIIRFree64fc_16sc(IppsIIRState64fc_16sc* pState);
IppStatus ippsIIRFree64fc_32sc(IppsIIRState64fc_32sc* pState);
IppStatus ippsIIRFree64fc_32fc(IppsIIRState64fc_32fc* pState);
```

Parameters

pState	Pointer to an IIR filter state structure to be closed.

Description

The function `ippsIIRFree` is declared in the `ipps.h` file. This function closes the IIR filter state by freeing all memory associated with a filter state created by `ippsIIRInitAlloc` or `ippsIIRInitAlloc_BiQuad`. Call `ippsIIRFree` after filtering is completed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

IIRInit

Initializes an arbitrary IIR filter state.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsIIRInit32s_16s(IppsIIRState32s_16s** ppState, const Ipp32s*
    pTaps, int order, int tapsFactor, const Ipp32s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32s_16s32f(IppsIIRState32s_16s** ppState, const Ipp32f*
    pTaps, int order, const Ipp32s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32f_16s(IppsIIRState32f_16s** ppState, const Ipp32f*
    pTaps, int order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_16s(IppsIIRState64f_16s** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_32s(IppsIIRState64f_32s** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsIIRInit32sc_16sc(IppsIIRState32sc_16sc** ppState, const Ipp32sc*
    pTaps, int order, int tapsFactor, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32sc_16sc32fc(IppsIIRState32sc_16sc** ppState, const
    Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32fc_16sc(IppsIIRState32fc_16sc** ppState, const Ipp32fc*
    pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_16sc(IppsIIRState64fc_16sc** ppState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_32sc(IppsIIRState64fc_32sc** ppState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsIIRInit_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps, int
    order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_32f(IppsIIRState64f_32f** ppState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_64f(IppsIIRState_64f** ppState, const Ipp64f* pTaps, int
    order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_32fc(IppsIIRState_32fc** ppState, const Ipp32fc* pTaps,
    int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_32fc(IppsIIRState64fc_32fc** ppState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_64fc(IppsIIRState_64fc** ppState, const Ipp64fc* pTaps,
    int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

Parameters

pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1).
tapsFactor	Scale factor for the taps of integer data type.
order	Order of the IIR filter.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order*.
ppState	Pointer to the pointer to the arbitrary IIR state structure to be created.
pBuffer	Pointer to the external buffer.

Description

The function `ippsIIRInit` is declared in the `ipps.h` file. This function initializes an arbitrary IIR filter state in the external buffer. The size of this buffer should be computed previously by calling the function `IIRGetStateSize`. The initialization functions copy the taps from the array *pTaps* into the state structure *pState*. The *tapsFactor* is used to scale integer tap values. The *order* -length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The 2*(*order* + 1) -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

$A_0 \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. Example 6-15 shows how to convert floating-point taps into integer data type.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the *pTaps*, *pBuffer*, or *ppState* is `NULL`.
`ippStsDivByZeroErr`	Indicates an error when A_0 is equal to 0.
`ippStsIIROrderErr`	Indicates an error when *order* is less than or equal to 0.

IIRInit_BiQuad

Initializes an IIR filter state.

Syntax

Case 1: Operation on integer samples.

```
IppStatus ippsIIRInit32s_BiQuad_16s(IppsIIRState32s_16s** ppState, const
    Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine, Ipp8u*
    pBuffer);
IppStatus ippsIIRInit32s_BiQuad_16s32f(IppsIIRState32s_16s** ppState, const
    Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32f_BiQuad_16s(IppsIIRState32f_16s** ppState, const
    Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_BiQuad_16s(IppsIIRState64f_16s** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_BiQuad_32s(IppsIIRState64f_32s** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32sc_BiQuad_16sc(IppsIIRState32sc_16sc** ppState, const
    Ipp32sc* pTaps, int numBq, int tapsFactor, const Ipp32sc* pDlyLine, Ipp8u*
    pBuffer);
IppStatus ippsIIRInit32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc** ppState,
    const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32fc_BiQuad_16sc(IppsIIRState32fc_16sc** ppState, const
    Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_BiQuad_16sc(IppsIIRState64fc_16sc** ppState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_BiQuad_32sc(IppsIIRState64fc_32sc** ppState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

Case 2: Operation on floating point samples.

```
IppStatus ippsIIRInit_BiQuad_32f(IppsIIRState_32f** ppState, const Ipp32f*
    pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_BiQuad_32f(IppsIIRState64f_32f** ppState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_BiQuad_64f(IppsIIRState_64f** ppState, const Ipp64f*
    pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_32fc(IppsIIRState_32fc** ppState, const Ipp32fc*
    pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_BiQuad_32fc(IppsIIRState64fc_32fc** ppState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_BiQuad_64fc(IppsIIRState_64fc** ppState, const Ipp64fc*
    pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

Parameters

pTaps	Pointer to the array containing the taps. The number of elements in the array is 6**numBq*.
tapsFactor	Scale factor for the taps of integer data type.
numBq	Number of cascades of biquads..
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is 2**numBq*.
ppState	Pointer to the pointer to the biquad IIR state structure.
pBuffer	Pointer to the external buffer .

Description

The function `ippsIIRInit_BiQuad` are declared in the `ipps.h` file. This function initializes a biquad (BQ) IIR filter state in the external buffer. The size of this buffer should be computed previously by calling the function `IIRGetStateSize_BiQuad`. The initialization function copies the taps from the array *pTaps* into the state structure *ppState*. The *tapsFactor* is used to scale integer tap values. The 2**numBq* -length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

A biquad IIR filter is defined by a cascade of biquads. The number of cascades of biquads is specified by the *numBq* value. The 6**numBq* -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \ldots A_{numBq\text{-}1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. Example 6-15 shows how to convert floating-point taps into integer data type.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the *pTaps*, *pBuffer*, or *ppState* is `NULL`.

`ippStsDivByZeroErr`	Indicates an error when A_0, $A_{n,0}$ or $B_{n,0}$ is equal to 0.
`ippStsIIROrderErr`	Indicates an error when *numBq* is less than or equal to 0.

IIRGetStateSize

Computes the length of the external buffer for the arbitrary IIR filter state structure.

Syntax

```
IppStatus ippsIIRGetStateSize32s_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32s_16s32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32s(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize32sc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_16sc32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32fc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32sc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64f(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize_32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64fc(int order, int* pBufferSize);
```

Parameters

order	Order of the IIR filter.
pBufferSize	Pointer to the computed buffer size value.

Description

The function `ippsIIRGetStateSize` is declared in the `ipps.h` file. This function computes the size of the external buffer for an arbitrary IIR filter state, and stores the result in *pBufferSize*.

To compute a size of the buffer, the filter order argument *order* should be specified.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pBufferSize* pointer is `NULL`.
`ippStsIIROrderErr`	Indicates an error when *order* is less than or equal to 0.

IIRGetStateSize_BiQuad

Computes the length of the external buffer for the biquad IIR filter state structure.

Syntax

```
IppStatus ippsIIRGetStateSize32s_BiQuad_16s(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize32s_BiQuad_16s32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize32f_BiQuad_16s(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_16s(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32s(int numBq, int* pBufferSize);

IppStatus ippsIIRGetStateSize32sc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_BiQuad_16sc32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize32fc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32sc(int numBq, int* pBufferSize);

IppStatus ippsIIRGetStateSize_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64f(int numBq, int* pBufferSize);
```

```
IppStatus ippsIIRGetStateSize_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64fc(int order, int* pBufferSize);
```

Parameters

numBq	Number of cascades of biquads.
pBufferSize	Pointer to the computed buffer size value.

Description

The function `ippsIIRGetStateSize_BiQuad` is declared in the `ipps.h` file. This function computes the size of the external buffer for a biquad IIR filter state, and store the result in *pBufferSize*.

To compute a size of the buffer, the number of cascades of biquads *numBq* should be specified.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pBufferSize* pointer is `NULL`.
`ippStsIIROrderErr`	Indicates an error when *numBq* is less than or equal to 0.

IIRSetTaps

Sets the taps in an IIR filter state.

Syntax

```
IppStatus ippsIIRSetTaps_32f(const Ipp32f* pTaps, IppsIIRState_32f* pState);
IppStatus ippsIIRSetTaps_32fc(const Ipp32fc* pTaps, IppsIIRState_32fc* pState);
IppStatus ippsIIRSetTaps_64f(const Ipp64f* pTaps, IppsIIRState_64f* pState);
IppStatus ippsIIRSetTaps_64fc(const Ipp64fc* pTaps, IppsIIRState_64fc* pState);
IppStatus ippsIIRSetTaps32s_16s(const Ipp32s* pTaps, IppsIIRState32s_16s*
    pState, int tapsFactor);
IppStatus ippsIIRSetTaps32s_16s32f(const Ipp32f* pTaps, IppsIIRState32s_16s*
    pState);
```

```
IppStatus ippsIIRSetTaps32sc_16sc(const Ipp32sc* pTaps, IppsIIRState32sc_16sc*
    pState, int tapsFactor);
IppStatus ippsIIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps,
    IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRSetTaps32f_16s(const Ipp32f* pTaps, IppsIIRState32f_16s*
    pState);
IppStatus ippsIIRSetTaps32fc_16sc(const Ipp32fc* pTaps, IppsIIRState32fc_16sc*
    pState);
IppStatus ippsIIRSetTaps64f_16s(const Ipp64f* pTaps,  IppsIIRState64f_16s*
    pState);
IppStatus ippsIIRSetTaps64f_32s(const Ipp64f* pTaps, IppsIIRState64f_32s*
    pState);
IppStatus ippsIIRSetTaps64f_32f(const Ipp64f* pTaps, IppsIIRState64f_32f*
    pState);
IppStatus ippsIIRSetTaps64fc_16sc(const Ipp64fc* pTaps, IppsIIRState64fc_16sc*
    pState);
IppStatus ippsIIRSetTaps64fc_32sc(const Ipp64fc* pTaps, IppsIIRState64fc_32sc*
    pState);
IppStatus ippsIIRSetTaps64fc_32fc(const Ipp64fc* pTaps, IppsIIRState64fc_32fc*
    pState);
```

Parameters

pTaps	Pointer to the array containing the tap values.
pState	Pointer to the IIR filter state structure.
tapsFactor	Scale factor for the taps of `Ipp32s` data type (for integer versions only).

Description

The function `ippsIIRSetTaps` is declared in the `ipps.h` file. This function sets new tap values in the previously initialized IIR filter state structure *pState*. New tap values must be specified in the array *pTaps*. To scale integer taps use the *tapsFactor* value.

The filter state must be initialized before calling the function `ippsIIRSetTaps`. The length of the array *pTaps* should be equal to the *tapsLen* parameter of the initialized filter state.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsDivByZeroErr	Indicates an error when when A_0 is equal to 0 for an arbitrary IIR filter state, or A_0, $A_{n,0}$ or $B_{n,0}$ is equal to 0 for a biquad IIR filter state.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

IIRGetDlyLine

Retrieves the delay line contents from the IIR filter state.

Syntax

```
IppStatus ippsIIRGetDlyLine32s_16s(IppsIIRState32s_16s* pState, const Ipp32s*
    pDlyLine);
IppStatus ippsIIRGetDlyLine32f_16s(IppsIIRState32f_16s* pState, const Ipp32f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine64f_16s(IppsIIRState64f_16s* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32s(IppsIIRState64f_32s* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState, const
    Ipp32sc* pDlyLine);
IppStatus ippsIIRGetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState, const
    Ipp32fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState, const
    Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState, const
    Ipp64fc* pDlyLine);

IppStatus ippsIIRGetDlyLine_32f(IppsIIRState_32f* pState, const Ipp32f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32f(IppsIIRState64f_32f* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRGetDlyLine_64f(IppsIIRState_64f* pState, const Ipp64f*
    pDlyLine);
```

```
IppStatus ippsIIRGetDlyLine_32fc(IppsIIRState_32fc* pState, const Ipp32fc*
    pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState, const
    Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine_64fc(IppsIIRState_64fc* pState, const Ipp64fc*
    pDlyLine);
```

Parameters

pState	Pointer to the IIR filter state structure.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters.

Description

The function `ippsIIRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from the state structure *pState* and stores them into the arrays *pDlyLine*. If the pointer is `NULL`, then the delay line values in the state structure are initialized to zero.

The filter state must be previously initialized by the one of the initialization functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

IIRSetDlyLine

Sets the delay line contents in an IIR filter state.

Syntax

```
IppStatus ippsIIRSetDlyLine32s_16s(IppsIIRState32s_16s* pState, const Ipp32s*
    pDlyLine);
IppStatus ippsIIRSetDlyLine32f_16s(IppsIIRState32f_16s* pState, const Ipp32f*
    pDlyLine);
```

```
IppStatus ippsIIRSetDlyLine64f_16s(IppsIIRState64f_16s* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32s(IppsIIRState64f_32s* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRSetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState, const
    Ipp32sc* pDlyLine);
IppStatus ippsIIRSetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState, const
    Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState, const
    Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState, const
    Ipp64fc* pDlyLine);

IppStatus ippsIIRSetDlyLine_32f(IppsIIRState_32f* pState, const Ipp32f*
    pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32f(IppsIIRState64f_32f* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRSetDlyLine_64f(IppsIIRState_64f* pState, const Ipp64f*
    pDlyLine);
IppStatus ippsIIRSetDlyLine_32fc(IppsIIRState_32fc* pState, const Ipp32fc*
    pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState, const
    Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine_64fc(IppsIIRState_64fc* pState, const Ipp64fc*
    pDlyLine);
```

Parameters

pState	Pointer to the IIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters. If the pointer is NULL, then the delay line values in the state structure are initialized to zero.

Description

The function ippsIIRSetDlyLine is declared in the ipps.h file. This function copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. If the pointer is NULL, then the delay line values in the state structure are initialized to zero.

The filter state must be previously initialized by the one of the initialization functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* pointers is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

IIROne

Filters a single sample through an IIR filter.

Syntax

```
IppStatus ippsIIROne_32f(Ipp32f src, Ipp32f* pDstVal, IppsIIRState_32f*
    pState);
IppStatus ippsIIROne_64f(Ipp64f src, Ipp64f* pDstVal, IppsIIRState_64f*
    pState);
IppStatus ippsIIROne64f_32f(Ipp32f src, Ipp32f* pDstVal, IppsIIRState64f_32f*
    pState);
IppStatus ippsIIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsIIRState_32fc*
    pState);
IppStatus ippsIIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal, IppsIIRState_64fc*
    pState);
IppStatus ippsIIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsIIRState64fc_32fc*
    pState);

IppStatus ippsIIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsIIRState32s_16s*
    pState, int scaleFactor);
IppStatus ippsIIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsIIRState32f_16s*
    pState, int scaleFactor);
IppStatus ippsIIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsIIRState64f_16s*
    pState, int scaleFactor);
IppStatus ippsIIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, IppsIIRState64f_32s*
    pState, int scaleFactor);

IppStatus ippsIIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsIIRState32sc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsIIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
    IppsIIRState64fc_32sc* pState, int scaleFactor);
```

Parameters

pState	Pointer to the IIR filter state structure.
src	Input sample.
pDstVal	Pointer to the output filtered sample.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsIIROne` is declared in the `ipps.h` file. This function filters a single sample *src* through an IIR filter with real taps, and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

The filter state must be initialized before calling either the `ippsIIR` or `ippsFIROne` function. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

IIR

Filters a block of samples through an IIR filter.

Syntax

Case 1: Not-in-place operation on integer samples.

```
IppStatus ippsIIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    IppsIIRState64f_32s* pState, int scaleFactor);

IppStatus ippsIIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int len,
    IppsIIRState64fc_32sc* pState, int scaleFactor);
```

Case 2: Not-in-place operation on floating point samples.

```
IppStatus ippsIIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, IppsIIRState_32f*
    pState);
IppStatus ippsIIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, IppsIIRState_64f*
    pState);
IppStatus ippsIIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    IppsIIRState64f_32f* pState);
IppStatus ippsIIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, IppsIIRState_32fc*
    pState);
IppStatus ippsIIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, IppsIIRState_64fc*
    pState);
IppStatus ippsIIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    IppsIIRState64fc_32fc* pState);
```

Case 3: In-place operation on integer samples.

```
IppStatus ippsIIR32s_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState32s_16s*
    pState, int scaleFactor);
IppStatus ippsIIR32f_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState32f_16s*
    pState, int scaleFactor);
```

```
IppStatus ippsIIR64f_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState64f_16s*
    pState, int scaleFactor);
IppStatus ippsIIR64f_32s_ISfs(Ipp32s* pSrcDst, int len, IppsIIRState64f_32s*
    pState, int scaleFactor);

IppStatus ippsIIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int len, IppsIIRState32fc_16sc*
    pState, int scaleFactor);
IppStatus ippsIIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int len, IppsIIRState32sc_16sc*
    pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int len, IppsIIRState64fc_16sc*
    pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int len, IppsIIRState64fc_32sc*
    pState, int scaleFactor);
```

Case 4: In-place operation on floating point samples.

```
IppStatus ippsIIR_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState_32f* pState);
IppStatus ippsIIR_64f_I(Ipp64f* pSrcDst, int len, IppsIIRState_64f* pState);
IppStatus ippsIIR64f_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState64f_32f*
    pState);
IppStatus ippsIIR_32fc_I(Ipp32fc* pSrcDst, int len, IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc_I(Ipp64fc* pSrcDst, int len, IppsIIRState_64fc* pState);
IppStatus ippsIIR64fc_32fc_I(Ipp32fc* pSrcDst, int len, IppsIIRState64fc_32fc*
    pState);
```

Parameters

pState	Pointer to the IIR filter state structure.
pSrc	Pointer to the input array of samples.
pDst	Pointer to the output array of filtered samples.
pSrcDst	Pointer to the input and output array of samples for the in-place operations.
len	Number of samples to be filtered.
scaleFactor	Refer to "Integer Scaling in Chapter 2.

Description

The function `ippsIIR` is declared in the `ipps.h` file. This function filters *len* samples in the input array *pSrc* or *pSrcDst* through an IIR filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

The filter state must be initialized before calling the `ippsIIR` function. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

Example 6-14 - Example 6-16 show how the Intel IPP functions for IIR filtering can be used.

Example 6-14 illustrates using `ippsIIR_32f` to suppress a 60 Hz signal.

Example 6-15 illustrates using `ippsIIR` to filter a sample. The function `ippsCnvrt_64f32s_Sfs` converts floating-point taps into integer data type before calling `ippsIIRInitAlloc_32s`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less or equal to 0.

Example 6-14 Using the `ippsIIR_32f` Function to Suppress a 60 Hz Signal

```
IppStatus iir( void ) {
#undef NUMITERS
#define NUMITERS 150
     int n;
     IppStatus status;
     IppsIIRState_32f *ctx;
     Ipp32f *x = ippsMalloc_32f( NUMITERS ), *y = ippsMalloc_32f( NUMITERS );
     /// A second-order notch filter having notch freq at 60 Hz
     const float taps[] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
     };
     /// generate a signal having 60 Hz freq sampled with 400 Hz freq
     for(n=0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *60 /400);
     ippsIIRInitAlloc_32f( &ctx, taps, 2, NULL );
     status = ippsIIR_32f( x, y, NUMITERS, ctx );
     printf_32f( " IIR 32f output+120 =", y+120, 5, status );
     ippsIIRFree_32f( ctx );
     ippsFree( y );
     ippsFree( x );
     return status;
}
Output:
    IIR 32f output + 120 =  -0.000094 0.000339 0.000458 0.000208 -0.000173
Matlab* Analog:
    >> B = [0.940809,-1.105987,0.940809]; A = [1,-1.105987,0.881618];
    n = 0:150; x = sin(2*pi*n*60/400); y = filter(B,A,x); y(121:125)
```

Example 6-15 Using the ippsIIR Function to Filter a Sample

```
IppStatus iir16s( void ) {
#undef NUMITERS
#define NUMITERS 150
     int n, tapsfactor = 30;
     IppStatus status;
     IppsIIRState32s_16s *ctx;
     Ipp16s *x = ippsMalloc_16s( NUMITERS ), *y = ippsMalloc_16s( NUMITERS );
     /// A second-order notch filter having notch freq at 60 Hz
     Ipp64f taps[6] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
     };
     Ipp32s taps32s[6];
     Ipp64f tmax, tmp[6];
     ippsAbs_64f( taps, tmp, 6 );
     ippsMax_64f( tmp, 6, &tmax );
     tapsfactor = 0;
     if( tmax > IPP_MAX_32S )
        while( (tmax/=2) > IPP_MAX_32S ) ++tapsfactor;
     else
        while( (tmax*=2) < IPP_MAX_32S ) --tapsfactor;
     if(tapsfactor >0 )
        ippsDivC_64f_I( (float)(1<<(++tapsfactor)), taps, 6 );
     else if( tapsfactor < 0 )
        ippsMulC_64f_I( (float)(1<<(-(tapsfactor))), taps, 6 );
     ippsConvert_64f32s_Sfs( taps, taps32s, 6, ippRndNear, 0 );
     /// generate a signal of 60 Hz freq that is sampled with 400 Hz freq
     for(n=0; n<NUMITERS; ++n) x[n] = (Ipp16s)(1000*sin(IPP_2PI*n*60/400));
     ippsIIRInitAlloc32s_16s( &ctx, taps32s, 2, tapsfactor, NULL );
     status = ippsIIR32s_16s_Sfs( x, y, NUMITERS, ctx, 0 );
     printf_16s( " IIR 32s output+120 =", y+120, 5, status );
     ippsIIRFree32s_16s( ctx );
     ippsFree( y );
     ippsFree( x );
     return status;
}
Output:
    IIR 32s output + 120 =  0 0 0 0 0
```

Example 6-16 Using the function ippsIIR for Low Pass Filtering

```
#include <ipp.h>
/*
y[n] = ( 1 * x[n- 2])
+ ( 2 * x[n- 1])
+ ( 1 * x[n- 0])
+ ( -0.1958157127 * y[n- 2])
+ ( -0.3695273774 * y[n- 1])
*/
#define NZEROS 2
#define NPOLES 2
#define GAIN 2.555350342e+00f
static float xv[NZEROS+1], yv[NPOLES+1];
static void filterloop( const float* x, float* y, int len )
{ for (int i=0; i<len; i++ )
{ xv[0] = xv[1]; xv[1] = xv[2];
xv[2] = x[i] / GAIN;
yv[0] = yv[1]; yv[1] = yv[2];
yv[2] = (xv[0] + xv[2]) + 2 * xv[1]
+ ( -0.1958157127f * yv[0]) + ( -0.3695273774f * yv[1]);
y[i] = yv[2];
}
}
const int LEN = 100;
int main() {
float x[LEN], yu[LEN], yi[LEN];
for(int i=0; i<LEN; i++) x[i] = i-3;
/// customer will compute
/// -1.1740073 -2.6968584 -1.9042342 -0.33358926
filterloop(x,yu,LEN);
/// matlab gives
///>> b=[1 2 1];a=[1 0.3695273774 0.1958157127
];x=[-3:1:15]/2.555350342;filter(b,a,x)
/// -1.1740 -2.6969 -1.9042 -0.3336
/// IPP will compute
/// -1.1740074 -2.6968584 -1.9042342 -0.33358920
const int order = 2;
float taps[6] = {1.0f, 2.0f, 1.0f, 1.0f, 0.3695273774f, 0.1958157127f};
IppsIIRState_32f *ctx;
ippsIIRInitAlloc_32f( &ctx, taps, order, (Ipp32f*)0 );
ippsDivC_32f_I( GAIN, x, LEN);
ippsIIR_32f( x, yi, LEN, ctx );
ippsIIRFree_32f( ctx );
return 0;
}
```

IIROne_Direct

Directly filters a single sample through the arbitrary IIR filter.

Syntax

```
IppStatus ippsIIROne_Direct_16s(Ipp16s src, Ipp16s* pDtsVal, const Ipp16s*
    pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIROne_Direct_16s_I(Ipp16s* pSrcDtsVal, const Ipp16s* pTaps, int
    order, Ipp32s* pDlyLine);
```

Parameters

src	Input sample.
pDstVal	Pointer to the output filtered sample.
pSrcDstVal	Pointer to the input and output sample for in-place operation.
pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1).
order	Order of the arbitrary IIR filter.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order*.

Description

The function `ippsIIROne_Direct` is declared in the `ipps.h` file. This function filters a single sample *src* (*pSrcDstVal* for in-place flavor) through an arbitrary order IIR filter and stores the result in *pDstVal* (*pSrcDstVal*). The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The *order* -length array *pDlyLine* specifies the delay line values. The 2*(*order* + 1) -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

The value $A_0 \geq 0$ holds the scale factor (and not a divisor) for all the other taps.

Return Values

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.

ippStsIIROrderErr	Indicates an error when *order* is less than or equal to 0.
ippStsScaleRangeErr	Indicates an error when A_0 is less than 0.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect

IIROne_BiQuadDirect

Directly filters a single sample through the biquad IIR filter.

Syntax

```
IppStatus ippsIIROne_BiQuadDirect_16s(Ipp16s src, Ipp16s* pDtsVal, const
    Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
IppStatus ippsIIROne_BiQuadDirect_16s_I(Ipp16s* pSrcDtsVal, const Ipp16s*
    pTaps, int numBq, Ipp32s* pDlyLine);
```

Parameters

src	Input sample.
pDstVal	Pointer to the output filtered sample.
pSrcDstVal	Pointer to the input and output sample for in-place operation.
pTaps	Pointer to the array containing the taps. The number of elements in the array is 6**numBq*.
numBq	Number of cascades of biquads.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is 2**numBq*.

Description

The function `ippsIIROne_BiQuadDirect` is declared in the `ipps.h` file. This function filters a single sample *src* (*pSrcDstVal* for in-place flavors) through a biquad IIR filter and stores the result in *pDstVal* (*pSrcDstVal*). The number of cascades of biquads is defined by the *numBq* value. The 2**numBq* -length array *pDlyLine* specifies the delay line values. The 6**numBq* -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \ldots A_{numBq\text{-}1,2}$

Values $A_{n,0} \geq 0$, $B_{n,0} \geq 0$ hold the scale factors (and not divisors) for all the other taps of each section.

Return Values

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsIIROrderErr	Indicates an error when *numBq* is less than or equal to 0.
ippStsScaleRangeErr	Indicates an error when $A_{n,0}$ is less than 0.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect

IIR_Direct

Directly filters a block of samples through the arbitrary IIR filter.

Syntax

```
IppStatus ippsIIR_Direct_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, const
    Ipp16s* pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIR_Direct_16s_I(Ipp16s* pSrcDst, int len, const Ipp16s* pTaps,
    int order, Ipp32s* pDlyLine);
```

Parameters

pSrc	Pointer to the input array of samples.
pDst	Pointer to the output array of filtered samples.
pSrcDst	Pointer to the input and output array for the in-place operation.
len	Number of samples to be filtered.
pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1).
order	Order of the arbitrary IIR filter.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order*.

Description

The function `ippsIIR_Direct` is declared in the `ipps.h` file. This function filters *len* samples in the input array *pSrc* or *pSrcDst* through an arbitrary IIR filter and stores the results in *pDst* or *pSrcDst*, respectively. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The *order* -length array *pDlyLine* specifies the delay line values. The 2*(*order* + 1) -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

The value $A_0 \geq 0$ holds the scale factor (and not a divisor) for all the other taps.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less or equal to 0.
`ippStsIIROrderErr`	Indicates an error when *order* is less than or equal to 0.
`ippStsScaleRangeErr`	Indicates an error when A_0 is less than 0.

IIR_BiQuadDirect

Directly filters a block of samples through the biquad IIR filter.

Syntax

```
IppStatus ippsIIR_BiQuadDirect_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
IppStatus ippsIIR_BiQuadDirect_16s_I(Ipp16s* pSrcDst, int len, const Ipp16s*
    pTaps, int numBq, Ipp32s* pDlyLine);
```

Parameters

pSrc	Pointer to the input array of samples.
pDst	Pointer to the output array of filtered samples.
pSrcDst	Pointer to the input and output array for the in-place operation.

len	Number of samples to be filtered.
pTaps	Pointer to the array containing the taps. The number of elements in the array is 6**numBq* for BQ filters.
numBq	Number of cascades of biquads.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is 2**numBq* for BQ filters.

Description

The function `ippsIIR_BiQuadDirect` is declared in the `ipps.h` file. This function filters *len* samples in the input array *pSrc* or *pSrcDst* through a biquad IIR filter and stores the result in *pDst* or *pSrcDst*, respectively. The number of cascades of biquads is defined by the *numBq* value. The 2**numBq* -length array *pDlyLine* specifies the delay line values. The 6**numBq* -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \ldots A_{numBq-1,2}$

Values $A_{n,0} \geq 0$, $B_{n,0} \geq 0$ hold the scale factors (and not divisors) for all the other taps of each section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.
`ippStsNullPtrErr`	Indicatesanerrorwhenoneofthespecifiedpointersis`NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsIIROrderErr`	Indicates an error when *numBq* is less than or equal to 0.
`ippStsScaleRangeErr`	Indicates an error when $A_{n,0}$ is less than 0.

IIRSparseInit

Initializes a sparse IIR filter structure.

Syntax

```
IppStatus ippsIIRSparseInit_32f(IppsIIRSparseState_32f** ppState, const
    Ipp32f* pNZTaps, const Ipp32s* pNZTapPos, int nzTapsLen1, int
    nzTapsLen2, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
```

Parameters

pNZTaps	Pointer to the array containing the non-zero tap values.
pNZTapPos	Pointer to the array containing positions of the non-zero tap values. The number of elements in the array is *nzTapsLen*.
nzTapsLen1, *nzTapsLen2*	Number of elements in the array containing the non-zero tap values.
pDlyLine	Pointer to the array containing the delay line values.
ppState	Double pointer to the sparse IIR state structure.
pBuffer	Pointer to the external buffer for the sparse IIR state structure.

Description

The function `ippsIIRSparseInit` is declared in the `ipps.h` file. This function initializes a sparse IIR filter state structure *pState* in the external buffer *pBuffer*. The size of this buffer should be computed previously by calling the function `ippsIIRSparseGetStateSize`.

The (*nzTapsLen1* + *nzTapsLen2*)-length array *pNZTaps* specifies the non-zero taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{nzTapsLen1-1}, A_0, A_1, \ldots, A_{nzTapsLen2-1}$.

The (*nzTapsLen1* + *nzTapsLen2*)-length array *pNZTapPos* specifies the non-zero tap positions arranged in the array as follows:

$BP_0, BP_1, \ldots, BP_{nzTapsLen1-1}, AP_0, AP_1, \ldots, AP_{nzTapsLen2-1}, \quad AP_0 \neq 0$

The initialization function copies the values of filter coefficients from the array *pNZTaps* containing non-zero taps and their positions from the array *pNZTapPos* into the state structure *pState*.
The array *pDlyLine* contains the delay line values.The number of elements in this array is

pNZTapPos[*nzTapsLen1*-1] + *pNZTapPos*[*nzTapsLen1* + *nzTapsLen2* - 1]. If the pointer to the array *pDlyLine* is not NULL, the array contents is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

CAUTION. The values of *nzTapsLen1*, *nzTapsLen2*, *pNZTapPos*[*nzTapsLen* -1], and *pNZTapPos*[*nzTapsLen1* + *nzTapsLen2* - 1] should be equal to those specified in the ippsIIRSparseGetStateSize function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers *ppState*, *pNZTaps*, *pNZTapPos*, or *pBuffer* is NULL.
ippStsIIROrderErr	Indicates an error if *nzTapsLen1* is less than or equal to 0, or *nzTapsLen2* is less than 0.
ippStsSparseErr	Indicates an error if positions of the non-zero taps are not in ascending order, or are negative or repetitive; or *pNZTapPos*[*nzTapsLen1*] is equal to 0.

IIRSparseGetStateSize

Computes the size of the external buffer for the sparse IIR filter structure.

Syntax

```
IppStatus ippsIIRSparseGetStateSize_32f(int nzTapsLen1, int nzTapsLen2,
    int order1, int order2, int* pStateSize);
```

Parameters

nzTapsLen1, *nzTapsLen2*	Number of elements in the array containing the non-zero tap values.
order1, *order2*	Order of the sparse IIR filter.
pStateSize	Pointer to the computed value of the external buffer.

Description

The function `ippsIIRSparseGetStateSize` is declared in the `ipps.h` file. This function computes the size in bytes of the external buffer for a sparse IIR filter state that is required for the function `ippsIIRSparseInit`. The computations are based on the specified number of non-zero filter coefficients *nzTapsLen1*, *nzTapsLen2* and filter orders *order1*, *order2*.
order1 = *pNZTapPos*[*nzTapsLen1*-1],
order1 = *pNZTapPos*[*nzTapsLen1* + *nzTapsLen2* - 1] (see description of the function `ippsIIRSparseInit`). The result value is stored in the *pStateSize*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pStateSize* pointer is `NULL`.
`ippStsIIROrderErr`	Indicates an error if *nzTapsLen1* is less than or equal to 0, or *nzTapsLen2* is less than 0.
`ippStsSparseErr`	Indicates an error if *order1* or *order2* is less than 0.

IIRSparse

Filters a block of samples through a sparse IIR filter.

Syntax

```
IppStatus ippsIIRSparse_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    IppsIIRSparseState_32f* pState);
```

Parameters

pState	Pointer to the sparse IIR filter state structure.
pSrc	Pointer to the input array.
pDst	Pointer to the output array.
len	Number of elements that will be filtered.

Description

The function `ippsIIRSparse` is declared in the `ipps.h` file. This function applies the sparse IIR filter to the *len* elements of the input array *pSrc*, and stores the results in *pDst*. The filter parameters - the number of non-zero taps *nzTapsLen1*, *nzTapsLen2*,

their values *pNZTaps* and their positions *pNZTapPos*, and the delay line values *pDlyLine* - are specified in the sparse IIR filter structure *pState* that should be previously initialized the function ippsIIRSparseInit.

In the following definition of the sparse IIR filter, the sample to be filtered is denoted $x(n)$, the non-zero taps are denoted B_i and A_i, their positions are denoted BP_i and AP_i .

The non-zero taps are arranged in the array as follows:

$B_0, B_1, \ldots, B_{nzTapsLen1-1}, A_0, A_1, \ldots, A_{nzTapsLen2-1}$.

The non-zero tap positions are arranged in the array as follows:

$BP_0, BP_1, \ldots, BP_{nzTapsLen1-1}, AP_0, AP_1, \ldots, AP_{nzTapsLen2-1}, \quad AP_0 \neq 0$

The return value is $y(n)$ is defined by the formula for a sparse IIR filter:

$$y(n) = \sum_{k=0}^{nzTapsLen1-1} B_k \cdot x(n - BP_k) + \sum_{k=1}^{nzTapsLen2-1} A_k \cdot y(n - AP_k) \quad 0 \le n < len$$

After the function has performed calculations, it updates the delay line values stored in the filter state structure.

Example 6-17 shows how to use the sparse IIR filter functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *len* is less than or equal to 0.

Example 6-17 Using the Sparse IIR Filter Functions

```
int buflen;
Ipp8u *buf;
int nzTapsLen1 = 5;//number of non-zero taps in the FIR part of the formula
int nzTapsLen2 = 3;//number of non-zero taps in the IIR part of the formula
Ipp32f nzTaps [] = {0.5, 0.4, 0.3, 0.2, 0.1, 0.8, 0.7, 0.6};
                                                //non-zero taps values (FIR+IIR)
Ipp32s nzTapsPos[] = {0, 10, 20, 30, 40, 1, 5, 15};
                                               //non-zero tap positions (FIR+IIR)
IppsIIRSparseState_32f* iirState;
Ipp32f *src, *dst;

/* ............................................ */

ippsIIRSparseGetStateSize_32f(nzTapsLen1, nzTapsLen2, nzTapsPos [nzTapsLen1 - 1],
                              nzTapsPos [nzTapsLen1 + nzTapsLen2 - 1], &buflen);
buf = ippsMalloc_8u(buflen);
ippsIIRSparseInit_32f(&iirState, nzTaps, nzTapsPos, nzTapsLen1, nzTapsLen2,
                      NULL, buf);

/* . . . . initializing src somehow . . . . */

ippsIIRSparse_32f(src, dst, len, iirState);

/* dst[i] = src[i] * 0.5 + src[i-10] * 0.4 +  src[i-20] * 0.3 +  src[i-30] * 0.2
+  src[i-40] * 0.1 + dst[i-1] * 0.8 + dst[i-5] * 0.7 + dst[i-15] * 0.6  */

/* ................................................ */

ippsFree(buf);
```

IIRGenLowpass
IIRGenHighpass

Computes lowpass and highpass IIR filter coefficients.

Syntax

```
IppStatus ippsIIRGenLowpass_64f(Ipp64f rFreq, Ipp64f ripple, int order, Ipp64f*
    pTaps, IppsIIRFilterType filterType);
IppStatus ippsIIRGenHighpass_64f(Ipp64f rFreq, Ipp64f ripple, int order,
    Ipp64f* pTaps, IppsIIRFilterType filterType);
```

Parameters

rFreq	Cut-off frequency, should be in the range (0, 0.5).
ripple	Posible ripple in pass band for `ippChebyshev1` type of filter.
order	Order of the filter [1, 12]
pTaps	Pointer to the array where computed tap values are stored.
filterType	Type of the IIR filter, possible values: `ippButterworth`, `ippChebyshev1`.

Description

The functions `ippsIIRGenLowpass` and `ippsIIRGenHighpass` are declared in the `ipps.h` file. These functions computes coefficients for lowpass or highpass FIR filters, respectively, with the cut-off frequency *rFreq*. The parameter *filterType* specifies the type of the filter. The computed coefficients are stored in the array *pTaps*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTaps* pointer is `NULL`.
`ippStsIIRGenOrderErr`	Indicates an error when the *order* is less than 1 or greater than 12.
`ippStsFilterFrequencyErr`	Indicates an error when the *rFreq* is out of the range.

Median Filter Functions

Median filters are nonlinear rank-order filters based on replacing each element of the input array with the median value, taken over the fixed neighborhood (mask) of the processed element. These filters are extensively used in image and signal processing applications. Median filtering removes impulsive noise, while keeping the signal blurring to the minimum. Typically mask size (or window width) is set to odd value which ensures simple function implementation and low output signal bias. In the Intel IPP median function implementation, the mask is always centered at the input element for which the median value is computed. You can use an even mask size in function calls as well, but internally it will be changed to odd by subtracting 1. Another specific feature of the median function implementation in the Intel IPP is that elements outside the input array, which are needed to determine the median value for “border” elements, are set to be equal to the corresponding edge element of the input array, i.e. are padded by cloning the edge element.

FilterMedian

Computes median values for each input array element.

Syntax

```
IppStatus ippsFilterMedian_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int
    maskSize);
IppStatus ippsFilterMedian_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
    maskSize);
IppStatus ippsFilterMedian_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, int
    maskSize);
IppStatus ippsFilterMedian_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, int
    maskSize);
IppStatus ippsFilterMedian_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, int
    maskSize);

IppStatus ippsFilterMedian_8u_I(Ipp8u* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_16s_I(Ipp16s* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_32s_I(Ipp32s* pSrcDst, int len, int maskSize);
```

```
IppStatus ippsFilterMedian_32f_I(Ipp32f* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_64f_I(Ipp64f* pSrcDst, int len, int maskSize);
```

Parameters

pSrcDst	Pointer to the input and output array (for the in-place operation).
pSrc	Pointer to the input array to be filtered by the function `ippsFilterMedian`.
pDst	Pointer to the output array filtered by the function `ippsFilterMedian`.
len	Number of elements in the array.
maskSize	Median mask size, must be a positive integer. If an even value is specified, the function subtracts 1 and uses the odd value of the filter mask for median filtering.

Description

The function `ippsFilterMedian` is declared in the `ipps.h` file. This function computes median values for each element of the input array *pSrc* or *pSrcDst*, and stores the result in *pDst* or *pSrcDst*, respectively.

NOTE. The value of a non-existent point is equal to the last point value, for example: $x[-1]=x[0]$ or $x[len]=x[len-1]$.

Example 6-18 illustrates using `ippsFilterMedian_16s_I` for single-rate filtering.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsEvenMedianMaskSize`	Indicates a warning when the median mask length is even.

Example 6-18 Single-Rate Filtering with the `ippsFilterMedian` Function

```
void median(void) {
      Ipp16s x[8] = {1,2,127,4,5,0,7,8};
      IppStatus status = ippsFilterMedian_16s_I(x, 8, 3);
      printf_16s("median =", x,8, status);
}
Output:
    median =  1 2 4 5 4 5 7 8
Matlab* Analog:
    >> x = [1 2 127 4 5 0 7 8]; medfilt1(x)
```

Transform Functions

7

This chapter describes Intel® IPP functions that perform Fourier and discrete cosine transforms (DCT), as well as Hilbert and wavelet transforms of signals.

The full list of functions in this group is given in Table 7-1.

Table 7-1 Intel IPP Transform Functions

Function Base Name	Operation
Support Functions	
ConjPack	Converts the data in Pack format to complex data format.
ConjPerm	Converts the data in Perm format to complex data format.
ConjCcs	Converts the data in CCS format to complex data format.
MulPack	Multiplies the elements of two vectors stored in Pack format.
MulPerm	Multiplies the elements of two vectors stored in Perm format.
MulPackConj	Multiplies elements of a vector by the elements of a complex conjugate vector stored in Pack format.
Fast Fourier Transform Functions	
FFTInitAlloc_R, FFTInitAlloc_C	Allocates memory and initializes a fast Fourier transform specification structure for real and complex signals.
FFTFree_R, FFTFree_C	Closes a fast Fourier transform structure for real and complex signals.
FFTInit_R, FFTInit_C	Initializes a fast Fourier transform specification structure for real and complex signals.

Table 7-1 Intel IPP Transform Functions (continued)

Function Base Name	Operation
FFTGetSize_R, FFTGetSize_C	Computes sizes of the FFT specification structure and required working buffers.
FFTGetBufSize_R, FFTGetBufSize_C	Computes the size of the FFT work buffer.
FFTFwd_CToC	Computes the forward fast Fourier transform (FFT) of a complex signal.
FFTInv_CToC	Computes the inverse fast Fourier transform (FFT) of a complex signal.
FFTFwd_RToPack, FFTFwd_RToPerm, FFTFwd_RToCCS	Computes the forward fast Fourier transform (FFT) of a real signal.
FFTInv_PackToR, FFTInv_PermToR, FFTInv_CCSToR	Computes the inverse fast Fourier transform (FFT) of a real signal.
Discrete Fourier Transform Functions	
DFTInitAlloc_R, DFTInitAlloc_C	Initializes the discrete Fourier transform structure for real and complex signals.
DFTFree_R, DFTFree_C	Closes the discrete Fourier transform structure for real and complex signals.
DFTGetBufSize_R, DFTGetBufSize_C	Computes the size of the discrete Fourier transform work buffer.
DFTFwd_CToC	Computes the forward discrete Fourier transform of a complex signal.
DFTInv_CToC	Computes the inverse discrete Fourier transform of a complex signal.
DFTFwd_RToPack, DFTFwd_RToPerm, DFTFwd_RToCCS	Computes the forward discrete Fourier transform of a real signal.
DFTInv_PackToR, DFTInv_PermToR, DFTInv_CCSToR	Computes the inverse discrete Fourier transform of a real signal.
DFTOutOrdInitAlloc_C	Initializes the out-of-order discrete Fourier transform structure.
DFTOutOrdFree_C	Closes the out-of-order discrete Fourier transform structure.

Table 7-1 Intel IPP Transform Functions (continued)

Function Base Name	Operation
DFTOutOrdGetBufSize_C	Computes the size of the work buffer for the out-of-order discrete Fourier transform.
DFTOutOrdFwd_CToC	Computes the forward out-of-order discrete Fourier transform.
DFTOutOrdInv_CToC	Computes the inverse out-of-order discrete Fourier transform.
Goertz	Computes the discrete Fourier transform for a given frequency for a single complex signal.
GoertzTwo	Computes two discrete Fourier transforms for a given frequency for a single complex signal.
DCT Functions	
DCTFwdInitAlloc	Allocates memory and initializes the forward discrete cosine transform structure.
DCTInvInitAlloc	Allocates memory and initializes the inverse discrete cosine transform structure.
DCTFwdFree	Closes the forward discrete cosine transform structure.
DCTInvFree	Closes the inverse discrete cosine transform structure.
DCTFwdGetBufSize	Computes the size of the forward DCT work buffer.
DCTInvGetBufSize	Computes the size of the inverse DCT work buffer.
DCTFwdInit	Initializes the forward discrete cosine transform structure.
DCTInvInit	Initializes the inverse discrete cosine transform structure.
DCTFwdGetSize	Computes the size of all buffers required for the forward DCT.
DCTInvGetSize	Computes the size of all buffers required for the inverse DCT.
DCTFwd	Computes the forward discrete cosine transform (DCT) of a signal.
DCTInv	Computes the inverse discrete cosine transform (DCT) of a signal.
Hilbert Transform Functions	
HilbertInitAlloc	Initializes the Hilbert transform structure.
HilbertFree	Closes a Hilbert transform structure.

Table 7-1 Intel IPP Transform Functions (continued)

Function Base Name	Operation
Hilbert	Computes an analytic signal using the Hilbert transform.
Wavelet Transform Functions	
WTHaarFwd, WTHaarInv	Performs forward or inverse single-level discrete wavelet Haar transforms.
WTFwdInitAlloc, WTInvInitAlloc	Initializes the wavelet transform structure.
WTFwdFree, WTInvFree	Closes a wavelet transform structure.
WTFwd	Computes the forward wavelet transform.
WTFwdSetDlyLine, WTFwdGetDlyLine	Sets and gets the delay lines of the forward wavelet transform.
WTInv	Computes the inverse wavelet transform.
WTInvSetDlyLine, WTInvGetDlyLine	Sets and gets the delay lines of the inverse wavelet transform.

Fourier Transform Functions

This section describes the Fourier and the discrete cosine transform functions in the Intel IPP. The Intel IPP contains functions which perform the discrete Fourier transform (DFT), the fast Fourier transform (FFT), and the discrete cosine transform (DCT) of signal samples. It also includes variations of the basic functions to support different application requirements.

Special Arguments

This section describes the flag and hint arguments used by the Fourier transform functions.

Flag and Hint Arguments

The Fourier transform functions require you to specify the *flag* and *hint* arguments.

The *flag* argument specifies the result normalization method. Table 7-2 lists the values you can enter for the *flag* argument. Specify one and only one of the represented values in the *flag* argument. The *A* and *B* factors are multipliers used in the DFT computation.

Table 7-2 Flag Arguments for Fourier Transform Functions

Value	A	B	Description
IPP_FFT_DIV_FWD_BY_N	$1/N$	1	Forward transform is done with the $1/N$ normalization.
IPP_FFT_DIV_INV_BY_N	1	$1/N$	Inverse transform is done with the $1/N$ normalization.
IPP_FFT_DIV_BY_SQRTN	$1/N^{1/2}$	$1/N^{1/2}$	Forward and inverse transform is done with the $1/N^{1/2}$ normalization.
IPP_FFT_NODIV_BY_ANY	1	1	Forward or inverse transform is done without the $1/N$ or $1/N^{1/2}$ normalization.

The *hint* argument suggests using special code, faster but less accurate calculation, or more accurate but slower calculation. Table 7-3 lists values you can enter for the *hint* argument.

Note that the *hint* argument is also used for tone generation, see Tone-Generating Functions in Chapter 4, and for logarithmic addition, see Model Evaluation in Chapter 8.

Table 7-3 Hint Arguments for Fourier Transform Functions

Value	Description
ippAlgHintNone	No suggestions. The function selects an algorithm
ippAlgHintFast	Fast algorithm is suggested for computation.
ippAlgHintAccurate	Accurate algorithm is suggested for computation.

Packed Formats

This section describes the main packed formats Perm, Pack, and CCS used by the Fourier transform functions.

Pack Format

The Pack format is a convenient, compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real Fourier transform algorithms (“natural” in the sense that

bit-reversed order is natural for radix-2 complex Fourier transforms). In `Pack` format, the output samples of the Fourier transform are arranged as shown in Table 7-4 and Table 7-5. The output signal can be unpacked to a complex signal using the function `ippsConjPack`.

Perm Format

The `Perm` format stores the values in the order in which the Fourier transform algorithms use them. This is the most natural way of storing values for the Fourier transform algorithms. The `Perm` format is an arbitrary permutation of the `Pack` format. An important characteristic of the `Perm` format is that the real and imaginary parts of a given sample need not be adjacent.

In `Perm` format, the output samples of the Fourier transform are arranged as shown in Table 7-4 and Table 7-5. The output signal can be unpacked to a complex signal using the function `ippsConjPerm`.

NOTE. For input signal of odd length the perm and pack format are identical.

CCS Format

The `CCS` format stores the values of the first half of the output complex signal resulted from the forward Fourier transform.

In `CCS` format, the output samples of the Fourier transform are arranged as shown in Table 7-4 and Table 7-5. Note that the signal stored in `CCS` format is one complex element longer. The output signal can be unpacked to a complex signal using the function `ippsConjCcs`.

Table 7-4 Arrangement of Forward Fourier Transform Results in Packed Formats - Even Length

Index	0	1	2	3	. . .	N-2	N-1	N	N+1
Pack	R_0	R_1	I_1	R_2	. . .	$I_{(N-1)/2}$	$R_{N/2}$		
Perm	R_0	$R_{N/2}$	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	R_0	0	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

Table 7-5 Forward Fourier transform Result Representation in Packed Formats - Odd Length

Index	0	1	2	3	...	N-2	N-1	N
Pack	R_0	R_1	I_1	R_2	. . .	$R_{(N-1)/2}$	$I_{(N-1)/2}$	
Perm	R_0	R_1	I_1	R_2	. . .	$R_{(N-1)/2}$	$I_{(N-1)/2}$	
CCS	R_0	0	R_1	I_1	. . .	$I_{(N-1)/2-1}$	$R_{(N-1)/2}$	$I_{(N-1)/2}$

Format Conversion Functions

The following functions `ConjPack`, `ConjPerm`, and `ConjCcs` convert data from the packed formats to a usual complex data format using the FFT symmetry property for transforming real data. The output data is complex, the output array length is defined by the number of complex elements in the output vector. Note that the output array size is two times as big as the input array size. The data stored in `CCS` format require a bigger array than the other formats. Even and odd length arrays have some specific features discussed for each function separately.

ConjPack

Converts the data in `Pack` *format to complex data format.*

Syntax

```
IppStatus ippsConjPack_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPack_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPack_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPack_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPack_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPack_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
lenDst	Number of elements in the vector.

Description

The function `ippsConjPack` is declared in the `ipps.h` file. This function converts the data in `Pack` format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjPack` converts the data in `Pack` format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-6 shows the examples of unpack from the `Pack` format. The `Data` column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the `Packed` column. The output result is the complex data vector in the `Extended` column. The number of vector elements is in the `Length` column. Example 7-1 shows how to use the function `ippsConjPack`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst*, *pDst*, or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *lenDst* is less than or equal to 0.

Table 7-6 Examples of Unpack from the Pack Format

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -2, 7, -7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-1 Using the `ippsConjPack` Function

```
void ConjPack(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPack_16sc( x, y, 6 );
      printf_16sc("pack 6", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPack_16sc( x, y, 5 );
      printf_16sc("pack 5", y, 5, st );
}
Output:
    Pack 6: {1,0} {2,3} {5,6} {7,0} {5,-6} {2,-3}
    Pack 5: {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjPerm

Converts the data in `Perm` *format to complex data format.*

Syntax

```
IppStatus ippsConjPerm_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPerm_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPerm_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPerm_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Parameters

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
lenDst	Number of elements in the vector.

Description

The function `ippsConjPerm` is declared in the `ipps.h` file. This function converts the data in `Perm` format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjPerm` converts the data in `Perm` format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-7 shows the examples of unpack from the `Perm` format. The `Data` column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the `Packed` column. The output result is the complex data vector in the `Extended` column. The number of vector elements is in the `Length` column. Example 7-2 shows how to use the function `ippsConjPerm`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst*, *pDst*, or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *lenDst* is less than or equal to 0.

Table 7-7 Examples of Packed Data Obtained by FFT

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -7, -2, 7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-2 Using the ippsConjPerm Function

```
void ConjPerm(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPerm_16sc( x, y, 6 );
      printf_16sc("Perm 6:", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPerm_16sc( x, y, 5 );
      printf_16sc("Perm 5:", y, 5, st );
}
Output:
    Perm 6:  {1,0} {3,5} {6,7} {2,0} {6,-7} {3,-5}
    Perm 5:  {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjCcs

Converts the data in CCS *format to complex data format.*

Syntax

```
IppStatus ippsConjCcs_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjCcs_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjCcs_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjCcs_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Parameters

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
lenDst	Number of elements in the vector.

Description

The function `ippsConjCcs` is declared in the `ipps.h` file. This function converts the data in CCS format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjCcs` converts the data in CCS format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-8 shows the examples of unpack from the `CCS` format. The `Data` column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the `Packed` column. The output result is the complex data vector in the `Extended` column. The number of vector elements is in the `Length` column.The data stored in `CCS` format are two real elements longer.Example 7-3 shows how to use the function `ippsConjCcs`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst*, *pDst*, or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *lenDst* is less than or equal to 0.

Table 7-8 Examples of Unpack from the CCS Format

Data	Packed	Extended	Length
FFT([1])	1, 0	{1, 0}	1
FFT([1 2])	3, 0, -1, 0	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, 0, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, 0, -2, 7, -7, 0	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-3 Using the ippsConjCcs Function

```
void ConjCcs(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjCcs_16sc( x, y, 6 );
      printf_16sc("CCS 6:", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjCcs_16sc( x, y, 5 );
      printf_16sc("CCS 5:", y, 5, st );
}
Ouput:
    CCS 6:  {1,2} {3,5} {6,7} {8,9} {6,-7} {3,-5}
    CCS 5:  {1,2} {3,5} {6,7} {6,-7} {3,-5}
```

Functions for Packed Data Multiplication

The functions described in this section perform the element-wise complex multiplication of vectors stored in Pack or Perm formats. These functions are used with the function ippsFFTFwd and ippsFFTInv to perform fast convolution on real signals.

The standard vector multiplication function ippsMul can not be used to multiply Pack or Perm format vectors because:

- Two real samples are stored in Pack format.
- The Perm format might not pair the real parts of a signal with their corresponding imaginary parts.

NOTE. The vectors stored in CCS format can be multiplied using the standard function for complex data multiplication.

MulPack

Multiply the elements of two vectors stored in `Pack` *format.*

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsMulPack_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsMulPack_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
IppStatus ippsMulPack_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
```

Case 2: In-place operation

```
IppStatus ippsMulPack_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMulPack_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsMulPack_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
    int scaleFactor);
```

Parameters

pSrc1, pSrc2	Pointers to the vectors whose elements are to be multiplied together.
pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc1*[*n*] * *pSrc2*[*n*].
pSrc	Pointer to the vector whose elements are to be multiplied by the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMulPack` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2*, and store the result in *pDst*.

The in-place flavors `ippsMulPack` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The functions multiply the packed data according to their packed format. The data in Pack packed format include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

For the functions with the `Sfs` suffixes scaling is performed in accordance with the *scaleFactor* value. When the output value exceeds the data range, the result may become saturated.

Example 7-4 shows how to use the function `ippsMulPack_32f_I`.

Return Values

`ippStsNoErr`	Indicates no error
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst*, *pDst*, *pSrc1*, *pSrc2*, or *pSrc* pointer is `NULL`
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0

Example 7-4 Using the `ippsMulPack` Function

```
void mulpack( void ) {
     Ipp32f x[8], X[8], h[8]={1.0f/3,1.0f/3,1.0f/3,0,0,0,0,0}, H[8];
     IppStatus st;
     IppsFFTSpec_R_32f* spec;
     st = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
         ippAlgHintNone);
     ippsSet_32f( 3, x, 8 );
     x[3] = 5;
     st = ippsFFTFwd_RToPack_32f( x, X, spec, NULL );
     st = ippsFFTFwd_RToPack_32f( h, H, spec, NULL );
     ippsMulPack_32f_I( H, X, 8 );
     st = ippsFFTInv_PackToR_32f( X, x, spec, NULL );
     printf_32f("filtered =", x, 8, st );
     ippsFFTFree_R_32f( spec );
}
Output:
    filtered =  3.0 3.0 3.0 3.666667 3.666667 3.666667 3.0 3.0
Matlab* analog:
    >> x=3*ones(1,8); x(4)=5;h=zeros(1,8); h(1:3)=1/3;
    real(ifft(fft(x).*fft(h)))
```

MulPerm

Multiply the elements of two vectors stored in `Perm` *format.*

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsMulPerm_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
```

```
IppStatus ippsMulPerm_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
IppStatus ippsMulPerm_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
```

Case 2: In-place operation

```
IppStatus ippsMulPerm_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMulPerm_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsMulPerm_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
    int scaleFactor);
```

Parameters

pSrc1, pSrc2	Pointers to the vectors whose elements are to be multiplied together.
pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc1*[*n*] * *pSrc2*[*n*].
pSrc	Pointer to the vector whose elements are to be multiplied by the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsMulPerm` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2*, and store the result in *pDst*.

The in-place flavors of `ippsMulPerm` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The function multiplies the packed data according to their packed format. The data in `Perm` packed formats include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

For the functions with the `Sfs` suffixes scaling is performed in accordance with the *scaleFactor* value. When the output value exceeds the data range, the result may become saturated.

Return Values

ippStsNoErr	Indicates no error
ippStsNullPtrErr	Indicates an error when the *pSrcDst*, *pDst*, *pSrc1*, *pSrc2*, or *pSrc* pointer is NULL
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0

MulPackConj

Multiplies elements of a vector by the elements of a complex conjugate vector stored in `Pack` *format.*

Syntax

```
IppStatus ippsMulPackConj_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMulPackConj_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the first source vector.
pSrcDst	Pointer to the second source and destination vector.
len	Number of elements in the vector.

Description

The function `ippsMulPackConj` is declared in the `ipps.h` file. This function multiplies the elements of a source vector *pSrc* by elements of the vector that is complex conjugate to the source vector *pSrcDst* and stores the results in *pSrcDst*. The function performs only in-place operations on data stored in `Pack` format.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Fast Fourier Transform Functions

The functions described in this section compute the forward and inverse fast Fourier transform of real and complex signals. The FFT is similar to the discrete Fourier transform (DFT) but is significantly faster. The length of the vector transformed by the FFT must be a power of 2.

To use the FFT functions, initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms. The amount of prior calculations is thus reduced and the overall performance increased.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method.

The FFT specification structure can be initialized by the functions ippsFFTInitAlloc_C or ippsFFTInitAlloc_R that allocate memory and initialize the structure. Alternatively the structure can be initialized by the functions ippsFFTInit_C and ippsFFTInit_R. In this case the user must allocate memory for the FFT specification structure whose size should be computed previously using either ippsFFTGetSize_C or ippsFFTGetSize_R functions.

The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in Perm, Pack, or CCS format.

You can speed up the FFT by using an external buffer. The external buffer increases performance because it allows to avoid allocation and deallocation of internal buffers and to store data in cache. If the FFT structure is initialized by the functions ippsFFTInitAlloc_C or ippsFFTInitAlloc_R, then the size of the external buffer should be previously computed by either ippsFFTGetBufSize_C or ippsFFTGetBufSize_R. If the FFT structure is initialized by the functions ippsFFTInit_C or ippsFFTInit_R, then the size of the external buffer is returned by this functions.

If the external buffer is not specified (correspondent parameter is set to NULL), then the FFT function itself allocates the memory needed for operation.

FFTInitAlloc_R, FFTInitAlloc_C

Allocates memory and initializes an FFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsFFTInitAlloc_R_16s(IppsFFTSpec_R_16s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_R_32s(IppsFFTSpec_R_32s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_R_32f(IppsFFTSpec_R_32f** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_R_64f(IppsFFTSpec_R_64f** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_R_16s32s(IppsFFTSpec_R_16s32s** ppFFTSpec, int
    order, int flag, IppHintAlgorithm hint);
```

Case 2: Operation on complex signal

```
IppStatus ippsFFTInitAlloc_C_16s(IppsFFTSpec_C_16s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32s(IppsFFTSpec_C_32s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32f(IppsFFTSpec_C_32f** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64f(IppsFFTSpec_C_64f** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_16sc(IppsFFTSpec_C_16sc** ppFFTSpec, int order,
    int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32sc(IppsFFTSpec_C_32sc** ppFFTSpec, int order,
    int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32fc(IppsFFTSpec_C_32fc** ppFFTSpec, int order,
    int flag, IppHintAlgorithm hint);
```

```
IppStatus ippsFFTInitAlloc_C_64fc(IppsFFTSpec_C_64fc** ppFFTSpec, int order,
    int flag, IppHintAlgorithm hint);
```

Parameters

flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
order	FFT order. The input signal length is $N=2^{order}$.
ppFFTSpec	Double pointer to the FFT specification structure to be created.

Description

The functions `ippsFFTInitAlloc_R` and `ippsFFTInitAlloc_C` are declared in the `ipps.h` file. These functions allocate memory, create, and initialize the FFT specification structure *ppFFTSpec* with the following parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The *order* argument defines the transform length. Thus the input and output signals are 2^{order}-length arrays.

ippsFFTInitAlloc_R. The function `ippsFFTInitAlloc_R` allocates memory and initializes the real FFT specification structure.

ippsFFTInitAlloc_C. The function `ippsFFTInitAlloc_C` allocates memory and initializes the complex FFT specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *ppFFTSpec* pointer is `NULL`.
`ippStsFftOrderErr`	Indicates an error when the *order* value is incorrect.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

FFTFree_R, FFTFree_C

Closes an FFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsFFTFree_R_16s(IppsFFTSpec_R_16s* pFFTSpec);
IppStatus ippsFFTFree_R_32s(IppsFFTSpec_R_32s* pFFTSpec);
IppStatus ippsFFTFree_R_32f(IppsFFTSpec_R_32f* pFFTSpec);
IppStatus ippsFFTFree_R_64f(IppsFFTSpec_R_64f* pFFTSpec);
IppStatus ippsFFTFree_R_16s32s(IppsFFTSpec_R_16s32s* pFFTSpec);
```

Case 2: Operation on complex signal

```
IppStatus ippsFFTFree_C_16s(IppsFFTSpec_C_16s* pFFTSpec);
IppStatus ippsFFTFree_C_32s(IppsFFTSpec_C_32s* pFFTSpec);
IppStatus ippsFFTFree_C_32f(IppsFFTSpec_C_32f* pFFTSpec);
IppStatus ippsFFTFree_C_64f(IppsFFTSpec_C_64f* pFFTSpec);

IppStatus ippsFFTFree_C_16sc(IppsFFTSpec_C_16sc* pFFTSpec);
IppStatus ippsFFTFree_C_32sc(IppsFFTSpec_C_32sc* pFFTSpec);
IppStatus ippsFFTFree_C_32fc(IppsFFTSpec_C_32fc* pFFTSpec);
IppStatus ippsFFTFree_C_64fc(IppsFFTSpec_C_64fc* pFFTSpec);
```

Parameters

pFFTSpec	Pointer to the FFT specification structure.

Description

The function `ippsFFTFree` is declared in the `ipps.h` file. This function closes the FFT specification structure *pFFTSpec* by freeing all memory associated with the specification created by `ippsFFTInitAlloc_C` or `ippsFFTInitAlloc_R`. Call `ippsFFTFree` after the transform is completed.

ippsFFTFree_C. The function ippsFFTFree_C closes the complex FFT specification structure.

ippsFFTFree_R. The function ippsFFTFree_R closes the real FFT specification structure.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFFTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.

FFTInit_R, FFTInit_C

Initializes an FFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsFFTInit_R_16s(IppsFFTSpec_R_16s** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_R_32s(IppsFFTSpec_R_32s** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_R_32f(IppsFFTSpec_R_32f** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_R_64f(IppsFFTSpec_R_64f** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_R_16s32s(IppsFFTSpec_R_16s32s** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

Case 2: Operation on complex signal

```
IppStatus ippsFFTInit_C_16s(IppsFFTSpec_C_16s** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_C_32s(IppsFFTSpec_C_32s** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

```
IppStatus ippsFFTInit_C_32f(IppsFFTSpec_C_32f** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_C_64f(IppsFFTSpec_C_64f** ppFFTSpec, int order, int flag,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_C_16sc(IppsFFTSpec_C_16sc** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_C_32sc(IppsFFTSpec_C_32sc** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_C_32fc(IppsFFTSpec_C_32fc** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsFFTInit_C_64fc(IppsFFTSpec_C_64fc** ppFFTSpec, int order, int
    flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

Parameters

order	FFT order. The input signal length is $N=2^{order}$.
flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
ppFFTSpec	Double pointer to the FFT specification structure to be created.
pSpec	Pointer to the area for the FFT specification structure.
pSpecBuffer	Pointer to the work buffer.

Description

The functions `ippsFFTInit_R` and `ippsFFTInit_C` are declared in the `ipps.h` file. These functions initialize the FFT specification structure *ppFFTSpec* with the following parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The *order* argument defines the transform length. Thus the input and output signals are 2^{order}-length arrays.

Before calling these functions the memory must be allocated for the FFT specification structure and the work buffer (if it is required). The size of the FFT specification structure and the work buffer should be computed by the either `ippsFFTGetSize_C` or `ippsFFTGetSize_R`.

If the work buffer is not used, the parameter *pSpecBuffer* can be `NULL`. If the working buffer is used, the parameter *pSpecBuffer* can not be `NULL`.

ippsFFTInit_C. The function `ippsFFTInit_C` initializes the complex FFT specification structure.

ippsFFTInit_R. The function `ippsFFTInit_R` initializes the real FFT specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsFftOrderErr`	Indicates an error when the *order* value is incorrect.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.

FFTGetSize_R, FFTGetSize_C

Computes sizes of the FFT specification structure and required working buffers.

Syntax

Case 1: Operation on real signal

```
IppStatus, ippsFFTGetSize_R_16s(int order, int flag, IppHintAlgorithm hint,
    int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_R_32s(int order, int flag, IppHintAlgorithm hint, int*
    pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_R_32f(int order, int flag, IppHintAlgorithm hint, int*
    pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_R_64f(int order, int flag, IppHintAlgorithm hint, int*
    pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_R_16s32s(int order, int flag, IppHintAlgorithm hint,
    int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

Case 2: Operation on complex signal

```
IppStatus, ippsFFTGetSize_C_16s(int order, int flag, IppHintAlgorithm hint,
    int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_C_32s(int order, int flag, IppHintAlgorithm hint, int*
    pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_32f(int order, int flag, IppHintAlgorithm hint, int*
    pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_C_64f(int order, int flag, IppHintAlgorithm hint, int*
    pSpecSize, int* pSpecBufferSize, int* pBufferSize);

IppStatus ippsFFTGetSize_C_16sc(int order, int flag, IppHintAlgorithm hint,
    int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_C_32sc(int order, int flag, IppHintAlgorithm hint,
    int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_C_32fc(int order, int flag, IppHintAlgorithm hint,
    int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsFFTGetSize_C_64fc(int order, int flag, IppHintAlgorithm hint,
    int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

Parameters

order	FFT order. The input signal length is $N=2^{order}$.
flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
pSpecSize	Pointer to the FFT specification structure size value.
pSpecBufferSize	Pointer to the buffer size value for FFT initialization function.
pBufferSize	Pointer to the size value of the FFT external work buffer.

Description

The functions `ippsFFTGetSize_R` and `ippsFFTGetSize_C` are declared in the `ipps.h` file. These functions computes the size of FFT specification structure, the work buffer size for the FFT structure initialization functions `ippsFFTInit_C` and `ippsFFTInit_R` as well as the size of the FFT work buffer for the flavors of functions `ippsFFTFwd` and `ippsFFTInv`. Their values in bytes are stored in *pSpecSize*, *pSpecBufferSize*, and *pBufferSize* respectively.

ippsFFTGetSize_C. The function `ippsFFTGetSize_C` is used for complex flavors of the FFT functions.

ippsFFTGetSize_R. The function `ippsFFTGetSize_R` is used for real flavors of the FFT functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsFftOrderErr	Indicates an error if the *order* value is incorrect.
ippStsFftFlagErr	Indicates an error if the *flag* value is incorrect.

FFTGetBufSize_R, FFTGetBufSize_C

Computes the size of the FFT work buffer.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsFFTGetBufSize_R_16s(const IppsFFTSpec_R_16s* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_R_32s(const IppsFFTSpec_R_32s* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_R_32f(const IppsFFTSpec_R_32f* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_R_64f(const IppsFFTSpec_R_64f* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_R_16s32s(const IppsFFTSpec_R_16s32s* pFFTSpec, int*
    pBufferSize);
```

Case 2: Operation on complex signal

```
IppStatus ippsFFTGetBufSize_C_16s(const IppsFFTSpec_C_16s* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_C_32s(const IppsFFTSpec_C_32s* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_C_32f(const IppsFFTSpec_C_32f* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_C_64f(const IppsFFTSpec_C_64f* pFFTSpec, int*
    pBufferSize);
```

```
IppStatus ippsFFTGetBufSize_C_16sc(const IppsFFTSpec_C_16sc* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_C_32sc(const IppsFFTSpec_C_32sc* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_C_32fc(const IppsFFTSpec_C_32fc* pFFTSpec, int*
    pBufferSize);
IppStatus ippsFFTGetBufSize_C_64fc(const IppsFFTSpec_C_64fc* pFFTSpec, int*
    pBufferSize);
```

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pBufferSize	Pointer to the size value of the FFT external work buffer.

Description

The functions `ippsFFTGetBufSize_R` and `ippsFFTGetBufSize_C` are declared in the `ipps.h` file. These functions compute the size in bytes of the external work buffer that can be used for the functions that perform FFT, and stores the result in *pBufferSize*. To compute the size of the buffer the corresponding FFT specification structure *pFFTSpec* should be specified.

ippsFFTGetBufSize_C. The function `ippsFFTGetBufSize_C` is used for complex flavors of the FFT functions.

ippsFFTGetBufSize_R. The function `ippsFFTGetBufSize_R` is used for real flavors of the FFT functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pFFTSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pFFTSpec* is incorrect.

FFTFwd_CToC

Computes the forward fast Fourier transform (FFT) of a complex signal.

Syntax

Case 1: Not-in-place operation on real data type

```
IppStatus ippsFFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u*
    pBuffer);
IppStatus ippsFFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u*
    pBuffer);
IppStatus ippsFFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsFFTSpec_C_16s* pFFTSpecx, int
    scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_32s_Sfs(const Ipp32s* pSrcRe, const Ipp32s* pSrcIm,
    Ipp32s* pDstRe, Ipp32s* pDstIm, const IppsFFTSpec_C_32s* pFFTSpec, int
    scaleFactor, Ipp8u* pBuffer);
```

Case 2: Not-in-place operation on complex data type

```
IppStatus ippsFFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
    IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, const
    IppsFFTSpec_C_32sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: In-place operation on real data type

```
IppStatus ippsFFTFwd_CToC_32f_I(Ipp32f* pSrcDstRe, Ipp32f* pSrcDstIm, const
    IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64f_I(Ipp64f* pSrcDstRe, Ipp64f* pSrcDstIm, const
    IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_16s_ISfs(Ipp16s* pSrcDstRe, Ipp16s* pSrcDstIm, const
    IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_32s_ISfs(Ipp32s* pSrcDstRe, Ipp32s* pSrcDstIm, const
    IppsFFTSpec_C_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 4: In-place operation on complex data type

```
IppStatus ippsFFTFwd_CToC_32fc_I(Ipp32fc* pSrcDst, const IppsFFTSpec_C_32fc*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64fc_I(Ipp64fc* pSrcDst, const IppsFFTSpec_C_64fc*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_16sc_ISfs(Ipp16sc* pSrcDst, const IppsFFTSpec_C_16sc*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_32sc_ISfs(Ipp32sc* pSrcDst, const IppsFFTSpec_C_32sc*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pSrcDst	Pointer to the input and output array containing complex values (for the in-place operation).
pSrcDstRe	Pointer to the input and output array containing real parts of the signal(for the in-place operation).
pSrcDstIm	Pointer to the input and output array containing imaginary parts of the signal(for the in-place operation).
pBuffer	Pointer to the external work buffer, can be `NULL`.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsFFTFwd_CToC` is declared in the `ipps.h` file. This function computes the forward FFT of a complex signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure should be initialized by the either `ippsFFTInitAlloc_C` or `ippsFFTInit_C`.

The functions using the complex data type, e.g., with the `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*. Their in-place flavors use the complex array *pSrcDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with the `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively. Their in-place flavors uses separate real and imaginary arrays *pSrcDstRe* and *pSrcDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. This can increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is same for both consecutive operations.
If the FFT structure is initialized by the functions `ippsFFTInitAlloc_C` , then the size of the external buffer should be previously computed by the function `ippsFFTGetBufSize_C`.
If the FFT structure is initialized by the functions `ippsFFTInit_C`, then the size of the external buffer is returned by this function.

The length of the FFT must be a power of 2.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers with exception of *pBuffer* is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

FFTInv_CToC

Computes the inverse fast Fourier transform (FFT) of a complex signal.

Syntax

Case 1: Not-in-place operation on real data type

```
IppStatus ippsFFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u*
    pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u*
    pBuffer);
IppStatus ippsFFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe,
    const Ipp16s* pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm,
    const IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32s_Sfs(const Ipp32s* pSrcRe, const Ipp32s* pSrcIm,
    Ipp32s* pDstRe, Ipp32s* pDstIm, const IppsFFTSpec_C_32s* pFFTSpec, int
    scaleFactor, Ipp8u* pBuffer);
```

Case 2: Not-in-place operation on complex data type

```
IppStatus ippsFFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
    IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, const
    IppsFFTSpec_C_32sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: In-place operation on real data type

```
IppStatus ippsFFTInv_CToC_32f_I(Ipp32f* pSrcDstRe, Ipp32f* pSrcDstIm, const
    IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64f_I(Ipp64f* pSrcDstRe, Ipp64f* pSrcDstIm, const
    IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_16s_ISfs(Ipp16s* pSrcDstRe, Ipp16s* pSrcDstIm, const
    IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32s_ISfs(Ipp32s* pSrcDstRe, Ipp32s* pSrcDstIm, const
    IppsFFTSpec_C_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 4: In-place operation on complex data type

```
IppStatus ippsFFTInv_CToC_32fc_I(Ipp32fc* pSrcDst, const IppsFFTSpec_C_32fc*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64fc_I(Ipp64fc* pSrcDst, const IppsFFTSpec_C_64fc*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_16sc_ISfs(Ipp16sc* pSrcDst, const IppsFFTSpec_C_16sc*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32sc_ISfs(Ipp32sc* pSrcDst, const IppsFFTSpec_C_32sc*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pSrcDst	Pointer to the input and output array containing complex values (for the in-place operation).
pSrcDstRe	Pointer to the input and output array containing real parts of the signal(for the in-place operation).
pSrcDstIm	Pointer to the input and output array containing imaginary parts of the signal(for the in-place operation).
pBuffer	Pointer to the external work buffer, can be `NULL`.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsFFTInv_CToC` is declared in the `ipps.h` file. This function computes the inverse FFT of a complex signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure should be initialized by the either `ippsFFTInitAlloc_C` or `ippsFFTInit_C`.

The functions using the complex data type, e.g., with the `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*. Their in-place flavors use the complex array *pSrcDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with the `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively. Their in-place flavors uses separate real and imaginary arrays *pSrcDstRe* and *pSrcDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. This can increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is same for both consecutive operations.
If the FFT structure is initialized by the functions ippsFFTInitAlloc_C , then the size of the external buffer should be previously computed by the function ippsFFTGetBufSize_C. If the FFT structure is initialized by the functions ippsFFTInit_C, then the size of the external buffer is returned by this function.

The length of the FFT must be a power of 2.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers with exception of *pBuffer* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

FFTFwd_RToPack, FFTFwd_RToPerm, FFTFwd_RToCCS

Computes the forward or inverse fast Fourier transform (FFT) of a real signal.

Syntax

Case 1: Not-in-place operation, result in Pack Format

```
IppStatus ippsFFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: In-place operation, result in Pack Format

```
IppStatus ippsFFTFwd_RToPack_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: Not-in-place operation, result in Perm Format

```
IppStatus ippsFFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPerm_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 4: In-place operation, result in Perm Format

```
IppStatus ippsFFTFwd_RToPerm_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPerm_64f_I(Ipp64f* pSrcDst const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPerm_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPerm_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 5: Not-in-place operation, result in CCS Format

```
IppStatus ippsFFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst,
    const IppsFFTSpec_R_16s32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 6: In-place operation, result in CCS Format

```
IppStatus ippsFFTFwd_RToCCS_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array.
pDst	Pointer to the output array containing packed complex values.
pSrcDst	Pointer to the input and output arrays for the in-place operation.
pBuffer	Pointer to the external work buffer, can be NULL.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The functions ippsFFTFwd_RToPerm, ippsFFTFwd_RToPack, and ippsFFTFwd_RToCCS are declared in the ipps.h file. They compute the forward FFT of a real signal and store the result in Pack, Perm, or CCS packed formats respectively. The transform is performed in accordance with the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure should be initialized by the corresponding flavors of either ippsFFTInitAlloc_R or ippsFFTInit_C. The length of the FFT must be a power of 2.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. This can increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is same for both consecutive operations.
If the FFT structure is initialized by the functions ippsFFTInitAlloc_R , then the size of

the external buffer should be previously computed by the function ippsFFTGetBufSize_R. If the FFT structure is initialized by the functions ippsFFTInit_R, then the size of the external buffer is returned by this function.

ippsFFTFwd_RToPack. This function computes the forward FFT and stores the result in Pack format.

ippsFFTFwd_RToPerm. This function computes the forward FFT and stores the result in Perm format.

ippsFFTFwd_RToCCS. This function computes the forward FFT and stores the result in CCS format.

Table 7-4 and Table 7-5 show how the output result is represented in each of the packed format. The code Example 7-5 shows how to initialize the specification structure and call the function ippsFFTFwd_RToCCS_32f.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers with exception of *pBuffer* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

Example 7-5 Using the `ippsFFTFwd_RToCCS` Function

```
IppStatus fft( void ) {
     Ipp32f x[8], X[10];
     int n;
     IppStatus status;
     IppsFFTSpec_R_32f* spec;
     status = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
         ippAlgHintNone );
     for(n=0; n<8; ++n) x[n] = (float)cos(IPP_2PI *n *16/64);
     status = ippsFFTFwd_RToCCS_32f( x, X, spec, NULL );
     ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
     ippsFFTFree_R_32f( spec );
     printf_32f("fft magn =", x, 4, status );
     return status;
}
Output:
    fft magn = 0.000000 0.000000 4.000000 0.000000
Matlab* Analog:
    >> n=0:7; x=sin(2*pi*n*16/64); X=abs(fft(x)); X(1:4)
```

FFTInv_PackToR, FFTInv_PermToR, FFTInv_CCSToR

Computes the inverse fast Fourier transform (FFT) of a real signal.

Syntax

Case 1: Not-in-place operation on input data in `Pack` format

```
IppStatus ippsFFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: In-place operation on input data in `Pack` format

```
IppStatus ippsFFTInv_PackToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: Not-in-place operation on input data in `Perm` format

```
IppStatus ippsFFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 4: In-place operation on input data in `Perm` format

```
IppStatus ippsFFTInv_PermToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_32s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 5 Not-in-place operation on input data in `CCS` format

```
IppStatus ippsFFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst,
    const IppsFFTSpec_R_16s32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 6: In-place operation on input data in CCS format

```
IppStatus ippsFFTInv_CCSToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
    pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
    pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array.
pDst	Pointer to the output array containing real values.
pSrcDst	Pointer to the input and output for the in-place operation.
pBuffer	Pointer to the external work buffer, can be NULL.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The functions ippsFFTInv_PermToR, ippsFFTInv_PackToR, and ippsFFTInv_CCSToR are declared in the ipps.h file. They compute the inverse FFT of data stored in Pack, Perm, or CCS packed formats respectively. The transform is performed in accordance with the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure should be initialized by the corresponding flavors of either ippsFFTInitAlloc_R or ippsFFTInit_R. The length of the FFT must be a power of 2.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. This can increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is same for both consecutive operations.
If the FFT structure is initialized by the functions ippsFFTInitAlloc_R , then the size of the external buffer should be previously computed by the function ippsFFTGetBufSize_R. If the FFT structure is initialized by the functions ippsFFTInit_R, then the size of the external buffer is returned by this function.

ippsFFTInv_PackToR. This function computes the inverse FFT of input data in Pack format.

ippsFFTInv_PermToR. This function computes the inverse FFT of input data in Perm format.

ippsFFTInv_CCSToR. This function computes the inverse FFT of input data in CCS format.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers with exception of *pBuffer* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

Discrete Fourier Transform Functions

The functions described in this section compute the forward and inverse discrete Fourier transform of real and complex signals. The DFT is less efficient than the fast Fourier transform, however the length of the vector transformed by the DFT can be arbitrary.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method. The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in Perm, Pack, or CCS format.

To use the DFT functions, you should initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms.

For more information about the fast computation of the discrete Fourier transform, see [Mit93], section 8-2, *Fast Computation of the DFT*.

A special set of Intel IPP functions provides the so called “out-of-order” DFT of the complex signal. In this case, the elements in frequency domain for both forward and inverse transforms can be re-ordered to speed-up the computation of the transforms. This re-ordering is hidden from the user and can be different in different implementations of the functions. However, reversibility of each pair of functions for forward/inverse transforms is ensured.

DFTInitAlloc_R, DFTInitAlloc_C

Initializes the DFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsDFTInitAlloc_R_16s(IppsDFTSpec_R_16s** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_R_32f(IppsDFTSpec_R_32f** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_R_64f(IppsDFTSpec_R_64f** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);
```

Case 2: Operation on complex signal

```
IppStatus ippsDFTInitAlloc_C_16s(IppsDFTSpec_C_16s** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_32f(IppsDFTSpec_C_32f** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_64f(IppsDFTSpec_C_64f** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_C_16sc(IppsDFTSpec_C_16sc** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_32fc(IppsDFTSpec_C_32fc** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_C_64fc(IppsDFTSpec_C_64fc** ppDFTSpec, int len, int
    flag, IppHintAlgorithm hint);
```

Parameters

flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
len	Length of the DFT transform.
ppDFTSpec	Double pointer to the DFT specification structure to be created.

Description

The functions `ippsDFTInitAlloc_R` and `ippsDFTInitAlloc_C` are declared in the `ipps.h` file. These functions create and initialize the DFT specification structure *ppDFTSpec* with the following parameters: the transform *len*, the normalization *flag*, and the specific code *hint*. The *len* argument defines the transform length.

ippsDFTInitAlloc_R. The function `ippsDFTInitAlloc_R` initializes the real DFT specification structure.

ippsDFTInitAlloc_C. The function `ippsDFTInitAlloc_C` initializes the complex DFT specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *ppDFTSpec* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

DFTFree_R, DFTFree_C

Closes the DFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsDFTFree_R_16s(IppsDFTSpec_R_16s* pDFTSpec);
IppStatus ippsDFTFree_R_32f(IppsDFTSpec_R_32f* pDFTSpec);
IppStatus ippsDFTFree_R_64f(IppsDFTSpec_R_64f* pDFTSpec);
```

Case 2: Operation on complex signal

```
IppStatus ippsDFTFree_C_16s(IppsDFTSpec_C_16s* pDFTSpec);
IppStatus ippsDFTFree_C_32f(IppsDFTSpec_C_32f* pDFTSpec);
IppStatus ippsDFTFree_C_64f(IppsDFTSpec_C_64f* pDFTSpec);

IppStatus ippsDFTFree_C_16sc(IppsDFTSpec_C_16sc* pDFTSpec);
IppStatus ippsDFTFree_C_32fc(IppsDFTSpec_C_32fc* pDFTSpec);
IppStatus ippsDFTFree_C_64fc(IppsDFTSpec_C_64fc* pDFTSpec);
```

Parameters

pDFTSpec	Pointer to the DFT specification structure.

Description

The function `ippsDFTFree` is declared in the `ipps.h` file. This function closes the DFT specification structure *pDFTSpec* by freeing all memory associated with the specification created by `ippsDFTInitAlloc_C` or `ippsDFTInitAlloc_R`. Call `ippsDFTFree` after the transform is completed.

`ippsDFTFree_R`. The function `ippsDFTFree_R` closes the real DFT specification structure.

`ippsDFTFree_C`. The function `ippsDFTFree_C` closes the complex DFT specification structure.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDFTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.

DFTGetBufSize_R, DFTGetBufSize_C

Computes the size of the DFT work buffer.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsDFTGetBufSize_R_16s(const IppsDFTSpec_R_16s* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_R_32f(const IppsDFTSpec_R_32f* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_R_64f(const IppsDFTSpec_R_64f* pDFTSpec, int* pSize);
```

Case 2: Operation on complex signal

```
IppStatus ippsDFTGetBufSize_C_16s(const IppsDFTSpec_C_16s* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_C_32f(const IppsDFTSpec_C_32f* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_C_64f(const IppsDFTSpec_C_64f* pDFTSpec, int* pSize);

IppStatus ippsDFTGetBufSize_C_16sc(const IppsDFTSpec_C_16sc* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_C_32fc(const IppsDFTSpec_C_32fc* pDFTSpec, int* pSize);
IppStatus ippsDFTGetBufSize_C_64fc(const IppsDFTSpec_C_64fc* pDFTSpec, int* pSize);
```

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSize	Pointer to the DFT work buffer size value.

Description

The functions ippsDFTGetBufSize_R and ippsDFTGetBufSize_C are declared in the ipps.h file. These functions compute the size in bytes of an external memory buffer for the DFT described by the specification structure *pDFTSpec* and stores it in *pSize*.

To use external buffering, call this function after *pDFTSpec* initialization.

ippsDFTGetBufSize_R. The function ippsDFTGetBufSize_R gets the size of the real DFT work buffer.

ippsDFTGetBufSize_C. The function ippsDFTGetBufSize_C gets the size of the complex DFT work buffer.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDFTSpec* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.

DFTFwd_CToC

Computes the forward discrete Fourier transform of a complex signal.

Syntax

Case 1: Operation on real data type

```
IppStatus ippsDFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f* pDFTSpec, Ipp8u*
    pBuffer);
IppStatus ippsDFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f* pDFTSpec, Ipp8u*
    pBuffer);
IppStatus ippsDFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s* pDFTSpec, int
    scaleFactor, Ipp8u* pBuffer);
```

Case 2: Operation on complex data type

```
IppStatus ippsDFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
    IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDFTFwd_CToC` is declared in the `ipps.h` file. This function computes the forward DFT of a complex signal according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, e.g., with `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the corresponding function `ippsDFTGetBufSize_C` prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where *k* is the index of elements in the frequency domain, *n* is the index of elements in the time domain, *N* is the input signal *len*, and *A* is a multiplier defined by *flag*. Also, *x(n)* is *pSrc[n]* and *X(k)* is *pDst[k]*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DFTInv_CToC

Computes the inverse discrete Fourier transform of a complex signal.

Syntax

Case 1: Operation on real data type

```
IppStatus ippsDFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f* pDFTSpec, Ipp8u*
    pBuffer);
IppStatus ippsDFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f* pDFTSpec, Ipp8u*
    pBuffer);
IppStatus ippsDFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s* pDFTSpec, int
    scaleFactor, Ipp8u* pBuffer);
```

Case 2: Operation on complex data type

```
IppStatus ippsDFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    const IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
    IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDFTInv_CToC` is declared in the `ipps.h` file. This function computes the inverse DFT of a complex signal according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, e.g., with `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the corresponding function `ippsDFTGetBufSize_C` prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The inverse DFT functionality can be described as follows:

$$x(n) = B \sum_{k=0}^{N-1} X(k) \cdot \exp\left(j2\pi\frac{kn}{N}\right),$$

where *k* is the index of elements in the frequency domain, *n* is the index of elements in the time domain, *N* is the input signal *len*, and *B* is a multiplier defined by *flag*. Also, *x(n)* is *pDst[n]* and *X(k)* is *pSrc[k]*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DFTFwd_RToPack, DFTFwd_RToPerm, DFTFwd_RToCCS

Computes the forward discrete Fourier transform of a real signal.

Syntax

Case 1: Result in Pack format

```
IppStatus ippsDFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: Result in Perm format

```
IppStatus ippsDFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: Result in CCS format

```
IppStatus ippsDFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing real values .
pDst	Pointer to the output array containing packed complex values.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsDFTFwd` is declared in the `ipps.h` file. It compute the forward DFT of a real signal according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

The result of the forward transform (i.e. in the frequency-domain) of real signals is represented in several possible packed formats: `Pack`, `Perm`, or `CCS`. The data can be packed due to the symmetry property of the DFT transform of a real signal. Table 7-4 and Table 7-5 show how the output result is represented in each of the packed format.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the corresponding function `ippsDFTGetBufSize_R` prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where *k* is the index of elements in the frequency domain, *n* is the index of elements in the time domain, *N* is the input signal *len*, and *A* is a multiplier defined by *flag*. Also, *x(n)* is *pSrc[n]* and *X(k)* is *pDst[k]*.

ippsDFTFwd_PackToR. This function computes the forward DFT and stores the result in Pack format.

ippsDFTFwd_PermToR. This function computes the forward DFT and stores the result in Perm format.

ippsDFTFwd_RToCCS. This function computes the forward DFT and stores the result in CCS format.

Example 7-6 shows how to initialize the specification and call ippsDFTFwd_RToCCS_32f.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

Example 7-6 Using the `ippsDFTFwd_RToCCS` Function

```
IppStatus dft( void ) {
     Ipp32f x[7], X[8];
     int n;
     IppStatus status;
     IppsDFTSpec_R_32f* spec;
     status = ippsDFTInitAlloc_R_32f(&spec, 7, IPP_FFT_DIV_INV_BY_N,
          ippAlgHintNone);
     for( n=0; n<7; ++n ) x[n] = (float) cos(IPP_2PI * n * 14 / 49);
     status = ippsDFTFwd_RToCCS_32f( x, X, spec, NULL );
     ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
     ippsDFTFree_R_32f( spec );
     printf_32f("dft magn =", x, 4, status );
     return status;
}
Output:
    dft magn =  0.000000 0.000000 3.500000 0.000000
Matlab* analog:
    >> N=7;F=14/49;n=0:N-1;x=cos(2*pi*n*F);y=abs(fft(x));y(1:4)
```

DFTInv_PackToR, DFTInv_PermToR, DFTInv_CCSToR

Computes the inverse discrete Fourier transform of a real signal.

Syntax

Case 1: Input data in `Pack` format

```
IppStatus ippsDFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 2: Input data in Perm format

```
IppStatus ippsDFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Case 3: Input data in CCS format

```
IppStatus ippsDFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing packed complex values.
pDst	Pointer to the output array containing real values.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The ippsDFTInv function is declared in the ipps.h file. They compute the inverse DFT according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

The input data (i.e. in the frequency-domain) are represented in several possible packed formats: Pack, Perm, or CCS. Table 7-4 and Table 7-5 show how the input complex data can be represented in each of the packed format.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the corresponding function ippsDFTGetBufSize_R prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The inverse DFT functionality can be described as follows:

$$x(n) = B \sum_{k=0}^{N-1} X(k) \cdot \exp\left(j2\pi\frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *len*, and B is a multiplier defined by *flag*. Also, $x(n)$ is *pDst[n]* and $X(k)$ is *pSrc[k]*.

ippsDFTInv_PackToR. This function computes the inverse DFT or input data in `Pack` format.

ippsDFTInv_PermToR. This function computes the inverse DFT for input data in `Perm` format.

ippsDFTInv_CCSToR. This function computes the inverse DFT or input data in `CCS` format.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DFTOutOrdInitAlloc_C

Initializes the out-of-order discrete Fourier transform structure.

Syntax

```
IppStatus ippsDFTOutOrdInitAlloc_C_32fc(IppsDFTOutOrdSpec_C_32fc** ppDFTSpec,
    int len, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTOutOrdInitAlloc_C_64fc(IppsDFTOutOrdSpec_C_64fc** ppDFTSpec,
    int len, int flag, IppHintAlgorithm hint);
```

Parameters

ppDFTSpec	Double pointer to the DFT specification structure to be created.
flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
len	Length of the DFT transform.

Description

The function `ippsDFTOutOrdInitAlloc_C` is declared in the `ipps.h` file. This function creates and initializes the specification structure *pDFTSpec* for the out-of-order DFT with the following parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pFFTSpec* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

DFTOutOrdFree_C

Closes the out-of-order discrete Fourier transform structure.

Syntax

```
IppStatus ippsDFTOutOrdFree_C_32fc(IppsDFTOutOrdSpec_C_32fc* pDFTSpec);
IppStatus ippsDFTOutOrdFree_C_64fc(IppsDFTOutOrdSpec_C_64fc* pDFTSpec);
```

Parameters

pDFTSpec	Pointer to the DFT specification structure to be closed.

Description

The function `ippsDFTOutOrdFree` is declared in the `ipps.h` file. This function closes the specification structure *pDFTSpec* for the out-of-order DFT by freeing all memory associated with the specification created by `ippsDFTOutOrdInitAlloc_C`. Call `ippsDFTOutOrdFree` function after the transform is completed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDFTSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pDFTSpec* is incorrect.

DFTOutOrdGetBufSize_C

Computes the size of the work buffer for out-of-order discrete Fourier transform.

Syntax

```
IppStatusippsDFTOutOrdGetBufSize_C_32fc(constIppsDFTOutOrdSpec_C_32fc*pDFTSpec,
    int* pSize);
IppStatusippsDFTOutOrdGetBufSize_C_64fc(constIppsDFTOutOrdSpec_C_64fc*pDFTSpec,
    int* pSize);
```

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSize	Pointer to the DFT work buffer size value.

Description

The function `ippsDFTOutOrdGetBufSize` is declared in the `ipps.h` file. This function computes the size in bytes of an external memory buffer for the out-of-order DFT described by the specification structure *pDFTSpec*, and stores it in *pSize*.

To use external buffering, call this function after *pDFTSpec* initialization.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pDFTSpec* is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pDFTSpec* is incorrect.

DFTOutOrdFwd_CToC

Computes the forward out-of-order discrete Fourier transform.

Syntax

```
IppStatus ippsDFTOutOrdFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

Parameters

pSrc	Pointer to the source data.
pDst	Pointer to the output data.
pDFTSpec	Pointer to the DFT specification structure.
pBuffer	Pointer to the work buffer.

Description

The function ippsDFTOutOrdFwd_CToC is declared in the ipps.h file. This function computes the forward out-of-order DFT of a complex signal *pSrc* according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*, and store the result in *pDst*.

This function is analogous to ippsDFTFwd_CToC and has the similar functionality. The difference is that the elements in frequency domain *pDst* can be rearranged if this helps to speed up computations. This procedure is hidden from the user and depends on the specific implementation of the functions.However, the reversibility of each pair of functions for forward/inverse transforms is ensured.

The *pBuffer* argument provides the DFT functions with the necessary memory and allows to avoid memory allocation within the functions. The buffer helps to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the function ippsDFTOutOrdGetBufSize_C prior to using the DFT computation functions. If a NULL pointer is passed, memory will be allocated by the DFT computation functions internally.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicatesanerrorwhenoneofthespecifiedpointersis NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DFTOutOrdInv_CToC

Computes the inverse out-of-order discrete Fourier transform.

Syntax

```
IppStatus ippsDFTOutOrdInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
    IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
    IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

Parameters

pSrc	Pointer to the source data.
pDst	Pointer to the output data.
pDFTSpec	Pointer to the DFT specification structure.
pBuffer	Pointer to the work buffer.

Description

The function `ippsDFTOutOrdInv_CToC` is declared in the `ipps.h` file. This function computes the inverse out-of-order DFT of a complex signal *pSrc* according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*, and store the result in *pDst*.

This function is analogous to `ippsDFTInv_CToC` and has the similar functionality. The difference is that the elements in frequency domain *pSrc* can be rearranged if this helps to speed up computations. This procedure is hidden from the user and depends on the specific implementation of the functions.However, the reversibility of each pair of functions for forward/inverse transforms is ensured.

The *pBuffer* argument provides the DFT functions with the necessary memory and allows to avoid memory allocation within the functions. The buffer helps to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the function `ippsDFTOutOrdGetBufSize_C` prior to using the DFT computation functions. If a `NULL` pointer is passed, memory will be allocated by the DFT computation functions internally.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicatesanerrorwhenoneofthespecifiedpointersis`NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

DFT for a Given Frequency (Goertzel) Functions

The functions described in this section compute a single or a number of the discrete Fourier transforms for a given frequency. Note that the DFT exists only for the following normalized frequencies: 0, 1/*N*, 2/*N*,... (*N*-1)/*N*, where *N* is the number of time domain samples. Therefore you must select the frequency value from the above set.

These Intel IPP functions use a Goertzel algorithm [Mit98] and are more efficient when a small number of DFT values is needed.

Some of the functions compute two values, not one. The applications computing several values, e.g., the dual-tone multi frequency signal detection, work faster, especially on Intel® processors with SIMD instructions.

Goertz

Computes the discrete Fourier transform for a given frequency for a single complex signal.

Syntax

```
IppStatus ippsGoertz_32f(const Ipp32f* pSrc, int len, Ipp32fc* pVal, Ipp32f
    rFreq);
IppStatus ippsGoertz_64f(const Ipp64f* pSrc, int len, Ipp64fc* pVal, Ipp64f
    rFreq);
IppStatus ippsGoertz_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pVal, Ipp32f
    rFreq);
IppStatus ippsGoertz_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pVal, Ipp64f
    rFreq);
IppStatus ippsGoertz_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16sc* pVal, Ipp32f
    rFreq, int scaleFactor);
IppStatus ippsGoertz_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pVal,
    Ipp32f rFreq, int scaleFactor);
IppStatus ippsGoertzQ15_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pVal,
    Ipp16s rFreqQ15, int scaleFactor);
```

Parameters

pSrc	Pointer to the input complex data vector.

len	Number of elements in the vector.
pVal	Pointer to the output DFT value.
rFreq	Single relative frequency value [0, 1.0).
rFreqQ15	Single relative frequency value in Q15 format [0, 32767].
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsGoertz` is declared in the `ipps.h` file. This function computes a DFT for a complex input *len*-length signal *pSrc* for a given frequency *rFreq*, and stores the result in *pVal*.

`ippsGoertzQ15`. This function operates with relative frequency in Q15 format. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 1.0).

The functionality of the Goertzel algorithm can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where k/N is the normalized *rFreq* value for which the DFT is computed.

Example 7-7 illustrates the use of Goertzel functions for selecting the magnitudes of a given frequency when computing DFTs.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsRelFreqErr`	Indicates an error when *rFreq* is out of range.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example 7-7 Using Goertzel Functions for Selecting Magnitudes of a Given Frequency

```
IppStatus goertzel( void ) {
#undef  LEN
#define LEN 100
     IppStatus status;
     Ipp32fc *x = ippsMalloc_32fc( LEN ), y;
     int n;
     ///generate a signal of 60 Hz freq that
     /// is sampled with 400 Hz freq
     for( n=0; n<LEN; ++n) {
        x[n].re =(Ipp32f)sin(IPP_2PI * n * 60 / 400);
        x[n].im = 0;
     }
     status = ippsGoertz_32fc( x, LEN, &y, 60.0f / 400 );
     printf_32fc("goertz =", &y, 1, status );
     ippsFree( x );
     return status;
}
Output:
    goertz =  {0.000090,-50.000008}
Matlab* Analog
    >> N=100;F=60/400;n=0:N-1;x=sin(2*pi*n*F);y=fft(x);n=N*F;y(n+1)
```

GoertzTwo

Computes two discrete Fourier transforms for a given frequency for a single complex signal.

Syntax

```
IppStatus ippsGoertzTwo_32f(const Ipp32f* pSrc, int len, Ipp32fc val[2], const
    Ipp32f rFreq[2]);
IppStatus ippsGoertzTwo_64f(const Ipp64f* pSrc, int len,  Ipp64fc val[2], const
    Ipp64f rFreq[2]);
IppStatus ippsGoertzTwo_32fc(const Ipp32fc* pSrc, int len, Ipp32fc val[2],
    const Ipp32f rFreq[2]);
```

```
IppStatus ippsGoertzTwo_64fc(const Ipp64fc* pSrc, int len, Ipp64fc val[2],
    const Ipp64f rFreq[2]);
IppStatus ippsGoertzTwo_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc val[2],
    const Ipp32f rFreq[2], int scaleFactor);
IppStatus ippsGoertzTwoQ15_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc
    val[2], const Ipp16s rFreqQ15[2], int scaleFactor);
```

Parameters

pSrc	Pointer to the input complex data vector.
len	Number of elements in the vector.
pVal	Array of the output DFT values.
rFreq	Array of two relative frequency values [0, 1.0).
rFreqQ15	Array of two relative frequency values in Q15 format [0, 32767].
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsGoertzTwo` is declared in the `ipps.h` file. This function computes two DFTs for a complex input *len* -length signal *pSrc* for two given frequencies *rFreq*, and stores the result in the output array *val*. The computation of two DFTs on an Intel® Pentium® III processor is performed at the same speed as one. Therefore, the applications computing several DFTs are faster.

The functionality of the Goertzel algorithm can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where k/N is one of the normalized *rFreq* values for which the DFTs are computed.

ippsGoertzTwoQ15. This function operates with relative frequencies in Q15 format. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 1.0).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the pointers is `NULL`.
`ippStsRelFreqErr`	Indicates an error when *rFreq* is out of range.
`ippStsSizeErr`	Indicates an error when *len* is less or equal to 0.

Discrete Cosine Transform Functions

This section describes the functions that compute the discrete cosine transform (DCT) of a signal. DCT functions used in the Intel IPP signal processing data-domain implement the modified computation algorithm proposed in [Rao90].

DCTFwdInitAlloc

Allocates memory and initializes the forward DCT structure.

Syntax

```
IppStatus ippsDCTFwdInitAlloc_16s(IppsDCTFwdSpec_16s** ppDCTSpec, int len,
    IppHintAlgorithm hint);
IppStatus ippsDCTFwdInitAlloc_32f(IppsDCTFwdSpec_32f** ppDCTSpec, int len,
    IppHintAlgorithm hint);
IppStatus ippsDCTFwdInitAlloc_64f(IppsDCTFwdSpec_64f** ppDCTSpec, int len,
    IppHintAlgorithm hint);
```

Parameters

ppDCTSpec	Double pointer to the forward DCT specification structure to be created.
len	Number of samples in the DCT.
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."

Description

The function `ippsDCTFwdInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the forward DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

The forward DCT specification structure is used by the function `ippsDCTFwd`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *ppDCTSpec* pointer is `NULL`.

ippStsSizeErr	Indicates an error if *len* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error if memory allocation fails.

DCTInvInitAlloc

Allocates memory and initializes the inverse DCT structure.

Syntax

```
IppStatus ippsDCTInvInitAlloc_16s(IppsDCTInvSpec_16s** ppDCTSpec, int len,
    IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_32f(IppsDCTInvSpec_32f** ppDCTSpec, int len,
    IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_64f(IppsDCTInvSpec_64f** ppDCTSpec, int len,
    IppHintAlgorithm hint);
```

Parameters

ppDCTSpec	Double pointer to the inverse DCT specification structuree to be created.
len	Number of samples in the DCT.
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."

Description

The function ippsDCTInvInitAlloc is declared in the ipps.h file. This function allocates memory and initializes the inverse DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

The inverse DCT specification structure is used by the function ippsDCTInv.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *ppDCTSpec* pointer is NULL.
ippStsSizeErr	Indicates an error if *len* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error if memory allocation fails.

DCTFwdFree

Closes the forward DCT structure.

Syntax

```
IppStatus ippsDCTFwdFree_16s(IppsDCTFwdSpec_16s* pDCTSpec);
IppStatus ippsDCTFwdFree_32f(IppsDCTFwdSpec_32f* pDCTSpec);
IppStatus ippsDCTFwdFree_64f(IppsDCTFwdSpec_64f* pDCTSpec);
```

Parameters

pDCTSpec	Pointer to the forward DCT specification structure.

Description

The function `ippsDCTFwdFree` is declared in the `ipps.h` file. This function closes the forward DCT specification structure *pDCTSpec* initialized by the function `ippsDCTFwdInitAlloc`. Call the function `ippsDCTFwdFree` after the transform is completed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pDCTSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error if the specification identifier *pDCTSpec* is incorrect.

DCTInvFree

Closes the inverse discrete cosine transform structure.

Syntax

```
IppStatus ippsDCTInvFree_16s(IppsDCTInvSpec_16s* pDCTSpec);
IppStatus ippsDCTInvFree_32f(IppsDCTInvSpec_32f* pDCTSpec);
IppStatus ippsDCTInvFree_64f(IppsDCTInvSpec_64f* pDCTSpec);
```

Parameters

pDCTSpec	Pointer to the inverse DCT specification structure.

Description

The function `ippsDCTInvFree` is declared in the `ipps.h` file. This function closes the inverse DCT specification structure *pDCTSpec* initialized by the function `ippsDCTInvInitAlloc`. Call the function `ippsDCTInvFree` after the transform is completed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDCTSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pDCTSpec* is incorrect.

DCTFwdGetBufSize

Computes the size of the forward DCT work buffer.

Syntax

```
IppStatus ippsDCTFwdGetBufSize_32f(const IppsDCTFwdSpec_32f* pDCTSpec, int*
    pBufferSize);
IppStatus ippsDCTFwdGetBufSize_16s(const IppsDCTFwdSpec_16s* pDCTSpec, int*
    pBufferSize);
IppStatus ippsDCTFwdGetBufSize_64f(const IppsDCTFwdSpec_64f* pDCTSpec, int*
    pBufferSize);
```

Parameters

pDCTSpec	Pointer to the forward DCT specification structure.
pBufferSize	Pointer to the forward DCT work buffer size value.

Description

The function `ippsDCTFwdGetBufSize` is declared in the `ipps.h` file. This function computes in bytes the size *pBufferSize* of the work buffer for the forward DCT specified by the structure *pDCTSpec* .

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDCTSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pDCTSpec* is incorrect.

DCTInvGetBufSize

Computes the size of the inverse DCT work buffer.

Syntax

```
IppStatus ippsDCTInvGetBufSize_16s(const IppsDCTInvSpec_16s* pDCTSpec, int*
    pBufferSize);
IppStatus ippsDCTInvGetBufSize_32f(const IppsDCTInvSpec_32f* pDCTSpec, int*
    pBufferSize);
IppStatus ippsDCTInvGetBufSize_64f(const IppsDCTInvSpec_64f* pDCTSpec, int*
    pBufferSize);
```

Parameters

pDCTSpec	Pointer to the DCT specification structure.
pBufferSize	Pointer to the DCT work buffer size value.

Description

The function `ippsDCTInvGetBufSize` is declared in the `ipps.h` file. This function computes in bytes the size *pBufferSize* of the work buffer for the inverse DCT specified by the structure *pDCTSpec* .

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pDCTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDCTSpec* is incorrect.

DCTFwdInit

Initializes the forward discrete cosine transform structure.

Syntax

```
IppStatus ippsDCTFwdInit_16s(IppsDCTFwdSpec_16s** ppDCTSpec, int len,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsDCTFwdInit_32f(IppsDCTFwdSpec_32f** ppDCTSpec, int len,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsDCTFwdInit_64f(IppsDCTFwdSpec_64f** ppDCTSpec, int len,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

Parameters

ppDCTSpec	Double pointer to the forward DCT specification structure to be created.
len	Number of samples in the DCT.
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
pSpec	Pointer to the area for the DCT specification structure.
pSpecBuffer	Pointer to the additional work buffer.

Description

The function ippsDCTFWDInit is declared in the ipps.h file. This function initializes the forward DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

Before calling this function the memory must be allocated for the DCT specification structure and the work buffer (if it is required). The size of the DFT specification structure and the work buffer should be computed by the either ippsDCTFwdGetSize.

If the work buffer is not used, the parameter *pSpecBuffer* can be NULL. If the working buffer is used, the parameter *pSpecBuffer* can not be NULL.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

DCTInvInit

Initializes the inverse discrete cosine transform structure.

Syntax

```
IppStatus ippsDCTInvInit_16s(IppsDCTInvSpec_16s** ppDCTSpec, int len,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsDCTInvInit_32f(IppsDCTInvSpec_32f** ppDCTSpec, int len,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
IppStatus ippsDCTInvInit_64f(IppsDCTInvSpec_64f** ppDCTSpec, int len,
    IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

Parameters

ppDCTSpec	Double pointer to the inverse DCT specification structure to be created.
len	Number of samples in the DCT.
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
pSpec	Pointer to the area for the DCT specification structure.
pSpecBuffer	Pointer to the work buffer.

Description

The function ippsDCTInvInit is declared in the ipps.h file. This function initializes in the buffer *pSpec* the inverse DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

Before calling this function the memory must be allocated for the DCT specification structure and the work buffer (if it is required). The size of the DFT specification structure and the work buffer should be computed by the either ippsDCTInvGetSize.

If the work buffer is not used, the parameter *pSpecBuffer* can be NULL. If the working buffer is used, the parameter *pSpecBuffer* can not be NULL.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

DCTFwdGetSize

Computes the size of all buffers required for the forward DCT.

Syntax

```
IppStatus ippsDCTFwdGetSize_16s(int len, IppHintAlgorithm hint, int* pSpecSize,
    int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsDCTFwdGetSize_32f(int len, IppHintAlgorithm hint, int* pSpecSize,
    int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsDCTFwdGetSize_64f(int len, IppHintAlgorithm hint, int* pSpecSize,
    int* pSpecBufferSize, int* pBufferSize);
```

Parameters

len	Number of samples in the DCT.
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
pSpecSize	Pointer to the size of the forward DCT specification structure.
pSpecBufferSize	Pointer to the size of the work buffer for the initialization function.
pBufferSize	Pointer to the size of the forward DCT work buffer.

Description

The function ippsDCTFwdGetSize is declared in the ipps.h file. This function computes the size *pSpecSize* for the forward DCT structure with the following parameters: the transform *len*, and the specific code *hint*.

Additionally the function computes the size *pSpecBufferSize* of the work buffer for the initialization function ippsDCTFwdInit, and the size *pBufferSize* of the work buffer for the function ippsDCTFwd.

The function ippsDCTFwdGetSize should be called prior to them.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

DCTInvGetSize

Computes the size of all buffers required for the inverse DCT.

Syntax

```
IppStatus ippsDCTInvGetSize_16s(int len, IppHintAlgorithm hint, int* pSpecSize,
    int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsDCTInvGetSize_32f(int len, IppHintAlgorithm hint, int* pSpecSize,
    int* pSpecBufferSize, int* pBufferSize);
IppStatus ippsDCTInvGetSize_64f(int len, IppHintAlgorithm hint, int* pSpecSize,
    int* pSpecBufferSize, int* pBufferSize);
```

Parameters

len	Number of samples in the DCT.
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
pSpecSize	Pointer to the size of the forward DCT specification structure.
pSpecBufferSize	Pointer to the size of the work buffer for the initialization function.
pBufferSize	Pointer to the size of the forward DCT work buffer.

Description

The function ippsDCTInvGetSize is declared in the ipps.h file. This function computes in bytes the size *pSpecSize* of the external buffer for the inverse DCT structure with the following parameters: the transform *len*, and the specific code *hint*.

Additionally the function computes the size *pSpecBufferSize* of the work buffer for the initialization function ippsDCTInvInit, and the size *pBufferSize* of the work buffer for the function ippsDCTInv.

The function ippsDCTFwdGetSize should be called prior to them.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

DCTFwd

Computes the forward discrete cosine transform of a signal.

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTFwdSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDCTFwdSpec_32f* pDCTSpec, Ipp8u* pBuffer);
IppStatus ippsDCTFwd_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDCTFwdSpec_64f* pDCTSpec, Ipp8u* pBuffer);
```

Case 2: In-place operation

```
IppStatus ippsDCTFwd_16s_ISfs(Ipp16s* pSrcDst, const IppsDCTFwdSpec_16s*
    pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsDCTFwdSpec_32f* pDCTSpec,
    Ipp8u* pBuffer);
IppStatus ippsDCTFwd_64f_I(Ipp64f* pSrcDst, const IppsDCTFwdSpec_64f* pDCTSpec,
    Ipp8u* pBuffer);
```

Parameters

pDCTSpec	Pointer to the forward DCT specification structure.

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
pBuffer	Pointer to the external work buffer, can be NULL
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function ippsDCTFwd is declared in the ipps.h file. This function computes the forward discrete cosine transform (DCT) of the source signal *pSrc* (*pSrcDst* for in-place operations) in accordance with the specification structure *pDCTSpec* that must be initialized by calling the functions ippsDCTFwdInitAlloc or ippsDCTFwdInit beforehand. The result in stored in the *pDst* (*pSrcDst* for in-place operations).

If *len* is a power of 2, the function use an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *len*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT, $N = len$,

$C(k) = \frac{1}{\sqrt{N}}$ for $k = 0$, $C(k) = \frac{\sqrt{2}}{\sqrt{N}}$ for $k > 0$;

$x(n)$ is *pSrc[n]* and $y(k)$ is *pDst[k]*.

The forward DCT is defined by the formula:

$$y(k) = C(k) \sum_{n=0}^{N-1} x(n) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function requires an additional work buffer *pBuffer* . The size of this buffer should be computed previously using the functions ippsDCTFwdGetBufSize or ippsDCTFwdGetSize. This buffer may also increase performance if the DCT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

Example 7-8 shows how to use the functions `ippsDCTFwd_32f` and `ippsDCTInv_32f` to compress and reconstruct a signal.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the *pDCTSpec*, *pSrc*, *pDst*, *pSrcDst* pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error if the specification identifier *pDCTSpec* is incorrect.
`ippStsMemAllocErr`	Indicates an error if memory allocation fails.

DCTInv

Computes the inverse discrete cosine transform of a signal.

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsDCTInv_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTInvSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDCTInvSpec_32f* pDCTSpec, Ipp8u* pBuffer);
IppStatus ippsDCTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDCTInvSpec_64f* pDCTSpec, Ipp8u* pBuffer);
```

Case 2: In-place operation

```
IppStatus ippsDCTInv_16s_ISfs(Ipp16s* pSrcDst, const IppsDCTInvSpec_16s*
    pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDCTInv_32f_I(Ipp32f* pSrcDst, const IppsDCTInvSpec_32f* pDCTSpec,
    Ipp8u* pBuffer);
IppStatus ippsDCTInv_64f_I(Ipp64f* pSrcDst, const IppsDCTInvSpec_64f* pDCTSpec,
    Ipp8u* pBuffer);
```

Parameters

pDCTSpec	Pointer to the inverse DCT specification structure.
pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
pBuffer	Pointer to the external work buffer, may be NULL.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function ippsDCTInv is declared in the ipps.h file. This function computes the inverse discrete cosine transform (DCT) of the source signal *pSrc* (*pSrcDst* for in-place operations) in accordance with the specification structure *pDCTSpec* that must be initialized by calling the functions ippsDCTInvInitAlloc or ippsDCTInvInit beforehand. The result in stored in the *pDst* (*pSrcDst* for in-place operations).

If *len* is a power of 2, the functions use an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *len*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT, $N = len$,

$C(k) = \frac{1}{\sqrt{N}}$ for $k = 0$, $C(k) = \frac{\sqrt{2}}{\sqrt{N}}$ for $k > 0$;

$x(n)$ is *pDst[n]* and $y(k)$ is *pSrc[k]*.

The inverse DCT is defined by the formula:

$$x(n) = \sum_{k=0}^{N-1} C(k)y(k) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function requires an additional work buffer *pBuffer* . The size of this buffer should be computed previously using the functions ippsDCTInvGetBufSize or ippsDCTInvGetSize. This buffer may also increase performance if the DCT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

Example 7-8 shows how to use the functions ippsDCTFwd_32f and ippsDCTInv_32f to compress and reconstruct a signal.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the *pDCTSpec*, *pSrc*, *pDst*, *pSrcDst* pointers is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDCTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error if memory allocation fails.

Example 7-8 Using ippsDCTFwd and ippsDCTInv to Compress and Reconstruct a Signal

```
void dct( void ) {
#define LEN 256
     Ipp32f x[LEN], y[LEN];
     int n;
     IppsDCTFwdSpec_32f* fspec;
     IppsDCTInvSpec_32f* ispec;
     IppStatus status;
     /// data: Gaussian function, magn =1 and sigma=N/3
     for(n=0; n<LEN; ++n)
         x[n] = (float)(exp(-0.5*(LEN/2-n)*(LEN/2-n)/(LEN/3.0)));
     /// get cosine transform coefficients
     status = ippsDCTFwdInitAlloc_32f( &fspec, LEN, ippAlgHintNone );
     status = ippsDCTFwd_32f( x, y, fspec, NULL );
     ippsDCTFwdFree_32f( fspec );
     /// Set 3/4 of these coefficients to zero
     for(n=LEN/4; n<LEN; ++n) y[n] = 0.0f;
     /// restore signal using len/4 values
     status = ippsDCTInvInitAlloc_32f( &ispec, LEN, ippAlgHintNone );
     status = ippsDCTInv_32f( y, x, ispec, NULL );
     ippsDCTInvFree_32f( ispec );
}
```

Hilbert Transform Functions

The functions described in this section compute a discrete-time analytic signal from a real data sequence using the Hilbert transform. The analytic signal is a complex signal whose real part is a replica of the original data, and imaginary part contains the Hilbert transform. That is, the imaginary part is a version of the original real data with a 90 degrees phase shift. The Hilbert transformed data have the same amplitude and frequency content as the original real data, plus the additional phase information.

HilbertInitAlloc

Initializes the Hilbert transform structure.

Syntax

```
IppStatus ippsHilbertInitAlloc_32f32fc(IppsHilbertSpec_32f32fc** ppSpec, int len,
    IppHintAlgorithm hint);
IppStatus ippsHilbertInitAlloc_16s32fc(IppsHilbertSpec_16s32fc** ppSpec, int len,
    IppHintAlgorithm hint);
IppStatus ippsHilbertInitAlloc_16s16sc(IppsHilbertSpec_16s16sc** ppSpec, int len,
    IppHintAlgorithm hint);
```

Parameters

ppSpec	Double pointer to the Hilbert context structure being initialized.
len	Number of samples in the Hilbert transform.
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."

Description

The function `ippsHilbertInitAlloc` is declared in the `ipps.h` file. This function creates and initializes the Hilbert specification structure *pSpec* with the following parameters: the length of the transform *len*, and the specific code indicator *hint*. Call this function before actually using the Hilbert transform function `ippsHilbert`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSpec* pointer is `NULL`.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

HilbertFree

Closes a Hilbert transform structure.

Syntax

```
IppStatus ippsHilbertFree_32f32fc(IppsHilbertSpec_32f32fc* pSpec);
IppStatus ippsHilbertFree_16s32fc(IppsHilbertSpec_16s32fc* pSpec);
IppStatus ippsHilbertFree_16s16sc(IppsHilbertSpec_16s16sc* pSpec);
```

Parameters

pSpec	Pointer to the Hilbert specification structure to be closed.

Description

The function ippsHilbertFree is declared in the ipps.h file. This function closes the Hilbert specification structure *pSpec* by freeing all memory associated with the specification created by the function ippsHilbertInitAlloc. Call ippsHilbertFree after the transform is completed.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pSpec* is incorrect.

Hilbert

Computes an analytic signal using the Hilbert transform.

Syntax

```
IppStatus ippsHilbert_32f32fc(const Ipp32f* pSrc, Ipp32fc* pDst,
    IppsHilbertSpec_32f32fc* pSpec);
IppStatus ippsHilbert_16s32fc(const Ipp16s* pSrc, Ipp32fc* pDst,
    IppsHilbertSpec_16s32fc* pSpec);
IppStatus ippsHilbert_16s16sc_Sfs(const Ipp16s* pSrc, Ipp16sc* pDst,
    IppsHilbertSpec_16s16sc* pSpec, int scaleFactor);
```

Parameters

pSpec	Pointer to the Hilbert specification structure.
pSrc	Pointer to the vector containing original real data.
pDst	Pointer to the output array containing complex data.
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The function `ippsHilbert` is declared in the `ipps.h` file. This function computes a complex analytic signal *pDst*, which contains the original real signal *pSrc* as its real part and computed Hilbert transform as its imaginary part. The Hilbert transform is performed according to the *pSpec* specification parameters: the number of samples *len*, and the specific code *hint*. The input data is zero-padded or truncated to the size of *len* as appropriate.
For integer data types, the output result is scaled according to the *scaleFactor* value, which ensures that the output signal range and precision are retained.

Example 7-9 shows how to initialize the specification structure and use the function `ippsHilbert_32f32fc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.

ippStsContextMatchErr	Indicates an error when the specification identifier *pSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

Example 7-9 Using Hilbert Functions

```
IppStatus hilbert(void) {
    Ipp32f x[10];
    Ipp32fc y[10];
    int n;
    IppStatus status;
    IppsHilbertSpec_32f32fc* spec;

    status = ippsHilbertInitAlloc_32f32fc(&spec, 10, ippAlgHintNone);
    for(n = 0; n < 10; n ++) {
        x[n] = (Ipp32f)cos(IPP_2PI * n * 2 / 9);
    }
    status = ippsHilbert_32f32fc(x, y, spec);
    ippsMagnitude_32fc((Ipp32fc*)y, x, 5);
    ippsHilbertFree_32f32fc(spec);
    printf_32f("hilbert magn =", x, 5, status);
    return status;
}

Output:
hilbert magn = 1.0944 1.1214 1.0413 0.9707 0.9839

Matlab* Analog:
>> n=0:9; x=cos(2*pi*n*2/9); y=abs(hilbert(x)); y(1:5)
```

Wavelet Transform Functions

This section describes the wavelet transform functions implemented in Intel IPP.

In signal processing, signals can represented in both frequency and time-frequency domains. In many cases the wavelet transforms become an alternative to short time Fourier transforms.

The discrete wavelet signal can be considered as a set of the coefficients $a_{i,k}$ with two indices, one of which is a “frequency” characteristic and the other is a time localization. The coefficient value corresponds to the localized wave amplitude or to one of basis transform functions. The “frequency” index shows the time scale of the localized wave. Function bases originated from one local wave by decreasing the wave by 2^n in time are the most widely used. Such transforms can be used for building very efficient implementations called fast wavelet transforms by analogy with fast Fourier transforms. Figure 7-1 shows how the time and frequency plane is divided into areas that correspond to the local wave amplitudes. This kind of transforms is implemented in the Intel IPP and referred to as the discrete wavelet transform (DWT).

The DWT is one of the wavelet analysis methods that stem from the basis functions related to the scale factor 2. Thus, there is a basic common element shared by the DWT and the other packet analysis methods.

Likewise another basic element for signal reconstruction or synthesis can be defined, called the one-level inverse DWT. Figure 7-2 shows the diagram of the forward DWT which allows to switch to time-frequency representation shown in Figure 7-1. The diagram includes three levels of decomposition. Figure 7-3 shows the corresponding procedure of signal reconstruction based on the elementary one-level inverse transform.

The implementation of discrete multi-scale transforms is based on the use of interpolation and decimation filters with the resampling factor 2. The basis of the multi-scale signal decomposition and reconstruction functions uniquely defines the filter parameters. The Intel IPP multi-scale transform functions use filters with finite impulse response.

The Primitives contains two sets of functions.

- Transforms designed for fixed filter banks. These transforms yield the highest performance.
- Transforms that enable the user to work with arbitrary filters. These functions use effective polyphase filtration algorithms. The transform interface gives the option of processing the data in blocks, including in real-time applications.

Figure 7-1 Wavelet Decomposition Coefficients in Time-Frequency Domain

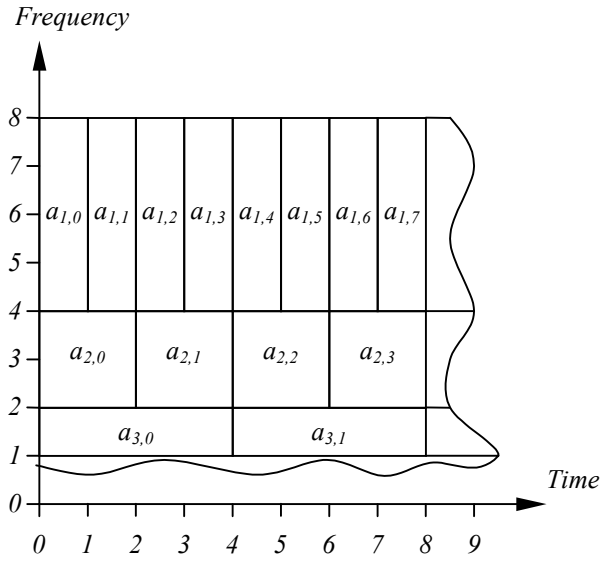


Figure 7-2 Three-Level Discrete Wavelet Decomposition

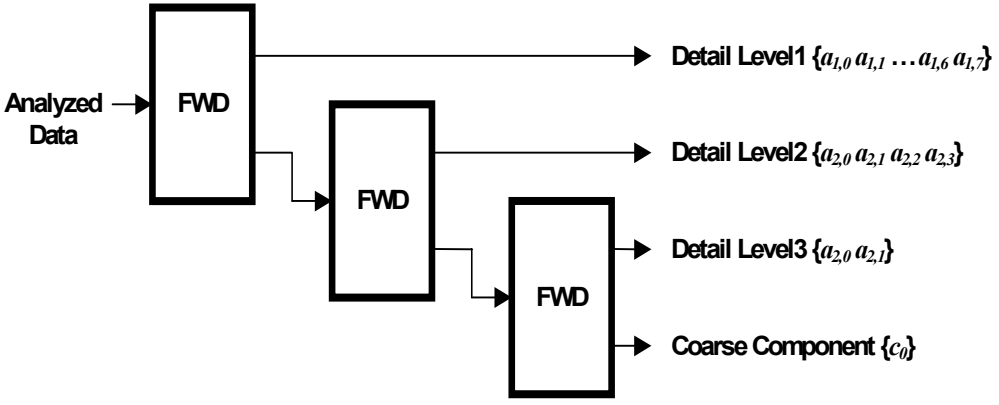
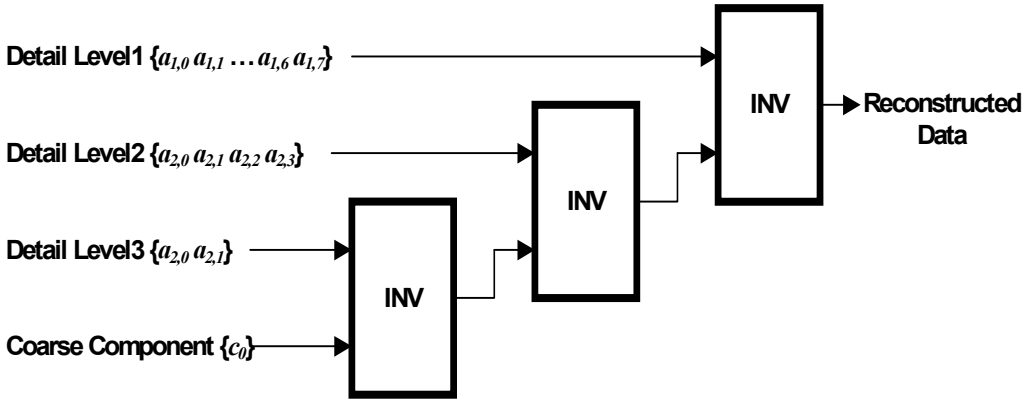


Figure 7-3 Three-Level Discrete Wavelet Reconstruction



Transforms for Fixed Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for fixed filter banks.

WTHaarFwd, WTHaarInv

Performs forward or inverse single-level discrete wavelet Haar transforms.

Syntax

Case 1: Forward transform

```
IppStatus ippsWTHaarFwd_8s(const Ipp8s* pSrc, int lenSrc, Ipp8s* pDstLow,
    Ipp8s* pDstHigh);
IppStatus ippsWTHaarFwd_16s(const Ipp16s* pSrc, int lenSrc, Ipp16s* pDstLow,
    Ipp16s* pDstHigh);
```

```
IppStatus ippsWTHaarFwd_32s(const Ipp32s* pSrc, int lenSrc, Ipp32s* pDstLow,
    Ipp32s* pDstHigh);
IppStatus ippsWTHaarFwd_64s(const Ipp64s* pSrc, int lenSrc, Ipp64s* pDstLow,
    Ipp64s* pDstHigh);
IppStatus ippsWTHaarFwd_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh);
IppStatus ippsWTHaarFwd_64f(const Ipp64f* pSrc, int lenSrc, Ipp64f* pDstLow,
    Ipp64f* pDstHigh);


IppStatus ippsWTHaarFwd_8s_Sfs(const Ipp8s* pSrc, int lenSrc, Ipp8s* pDstLow,
    Ipp8s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_16s_Sfs(const Ipp16s* pSrc, int lenSrc, Ipp16s*
    pDstLow, Ipp16s* pDstHigh, int scaleFactor );
IppStatus ippsWTHaarFwd_32s_Sfs(const Ipp32s* pSrc, int lenSrc, Ipp32s*
    pDstLow, Ipp32s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_64s_Sfs(const Ipp64s* pSrc, int lenSrc, Ipp64s*
    pDstLow, Ipp64s* pDstHigh, int scaleFactor);
```

Case 2: Inverse transform

```
IppStatus ippsWTHaarInv_8s(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh, Ipp8s*
    pDst, int lenDst);
IppStatus ippsWTHaarInv_16s(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh,
    Ipp16s* pDst, int lenDst);
IppStatus ippsWTHaarInv_32s(const Ipp32s* pSrcLow, const Ipp32s* pSrcHigh,
    Ipp32s* pDst, int lenDst);
IppStatus ippsWTHaarInv_64s(const Ipp64s* pSrcLow, const Ipp64s* pSrcHigh,
    Ipp64s* pDst, int lenDst);
IppStatus ippsWTHaarInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
    Ipp32f* pDst, int lenDst);
IppStatus ippsWTHaarInv_64f(const Ipp64f* pSrcLow, const Ipp64f* pSrcHigh,
    Ipp64f* pDst, int lenDst);


IppStatus ippsWTHaarInv_8s_Sfs(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh,
    Ipp8s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_16s_Sfs(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh,
    Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_32s_Sfs(const Ipp32s* pSrcLow, const Ipp32s* pSrcHigh,
    Ipp32s* pDst, int lenDst, int scaleFactor);
```

```
IppStatus ippsWTHaarInv_64s_Sfs(const Ipp64s* pSrcLow, const Ipp64s* pSrcHigh,
    Ipp64s* pDst, int lenDst, int scaleFactor);
```

Parameters

pSrc	Pointer to the array with the input signal for the `ippsWTHaarFwd` function.
lenSrc	Number of elements in the source vector *pSrc*.
pDstLow	Pointer to the array with the coarse “low frequency” components of the output of the `ippsWTHaarFwd` function.
pDstHigh	Pointer to the array with the detail “high frequency” components of the output of the `ippsWTHaarFwd` function.
pSrcLow	Pointer to the array with the coarse “low frequency” components of the input to the `ippsWTHaarInv` function.
pSrcHigh	Pointer to the array with the detail “high frequency” components of the input to the `ippsWTHaarInv` function.
pDst	Pointer to the array with the the output signal of the `ippsWTHaarInv` function.
lenDst	Number of elements in the destination vector *pDst*
scaleFactor	Refer to Integer Scaling in Chapter 2.

Description

The functions `ippsWTHaarFwd` and `ippsWTHaarInv` are declared in the `ipps.h` file. These functions perform forward and inverse single-level discrete Haar transforms. These transforms are orthogonal and reconstruct the original signal perfectly.

The forward transform can be considered as wavelet signal decomposition with lowpass decimation filter coefficients {1/2, 1/2} and highpass decimation filter coefficients {1/2, -1/2}.

The inverse transform is represented as wavelet signal reconstruction with lowpass interpolation filter coefficients {1, 1} and highpass interpolation filter coefficients {-1, 1}.

The decomposition filter coefficients are frequency response normalized to provide the same value range for both input and output signals. Thus, the amplitude of the low pass filter frequency response is 1 for zero-valued frequency, and the amplitude of the high pass filter frequency response is also 1 for the frequency value near to 0.5.

As the absolute values of the interpolation filter coefficients are equal to 1, the reconstruction of the signal requires few operations. It is well suited for usage in data compression applications. As the decomposition filter coefficients are powers of 2, the integer functions perform lossless decomposition with the *scaleFactor* value equal to -1. To avoid saturation, use higher-precision data types.

Note that the filter coefficients can be power spectral response normalized, see [Str96] for more information. Thus, the decomposition filter coefficients are $\{2^{-1/2}, 2^{-1/2}\}$ and $\{2^{-1/2}, -2^{-1/2}\}$; accordingly; the reconstruction filter coefficients are $\{2^{-1/2}, 2^{-1/2}\}$ and $\{-2^{-1/2}, 2^{-1/2}\}$.

In the following definition of the forward single-level discrete Haar transform, $N=lenSrc$. The coarse "low-frequency" component $c(k)$ is *pDstLow[k]* and the detail "high-frequency" component $d(k)$ is *pDstHigh[k]*; also $x(2k)$ and $x(2k+1)$ are even and odd values of the input signal *pSrc*, respectively.

$$c(k)=(x(2k)+x(2k+1))/2$$

$$d(k)=(x(2k+1)-x(2k))/2$$

In the inverse direction, $N=lenDst$. The coarse "low-frequency" component $c(k)$ is *pSrcLow[k]* and the detail "high-frequency" component $d(k)$ is *pSrcHigh[k]*; also $y(2i)$ and $y(2i+1)$ are even and odd values of the output signal *pDst*, respectively.

$$y(2i)=c(i)-d(i)$$

$$y(2i+1)=c(i)+d(i)$$

For even length N, $0\le k<N/2$ and $0\le i<N/2$. Also, "low-frequency" and "high-frequency" components are of size $N/2$ for both original and reconstructed signals. The total length of components is equal to the signal length N.

In case of odd length N, the vector is considered as a vector of the extended length $N+1$ whose two last elements are equal to each other $x[N]=x[N-1]$. The last elements of the coarse and detail components of the decomposed signal are defined as follows:

$$c((N+1)/2-1)=x(N-1)$$

$$d((N+1)/2-1)=0$$

Correspondingly, the last element of the reconstructed signal is defined as:

$$y(N)=y(N-1)=c((N+1)/2-1)$$

For odd length N, $0 \le k < (N-1)/2$ and $0 \le i < (N-1)/2$, assuming that $c((N+1)/2-1) = x(N-1)$ and $y(N-1) = c((N+1)/2-1)$. The “low-frequency” component is of size $(N+1)/2$. The “high-frequency” component is of size $(N-1)/2$, because the last element $d((N+1)/2-1)$ is always equal to 0. The total length of components is also N.

Such an approach applies continuation of boundaries for filters having the symmetry properties, see [Brislawn94].

When performing block mode transforms, take into consideration that for decomposition and reconstruction of even-length signals no extrapolations at the boundaries is used. In case of odd-length signals, a symmetric continuation of the signal boundary with the last point replica is applied.

When it is necessary to have a continuous set of output blocks, all the input blocks are to be of even length, besides the last one (which can be either of odd or even length). Thus, if the whole amount of elements is odd, only the last block can be of odd length.

ippsWTHaarFwd. The function `ippsWTHaarFwd` performs the forward single-level discrete Haar transform of a *lenSrc* -length signal *pSrc* and stores the decomposed coarse “low-frequency” components in *pDstLow*, and the detail “high-frequency” components in *pDstHigh*.

ippsWTHaarInv. The function `ippsWTHaarInv` performs the inverse single-level discrete Haar transform of the coarse “low-frequency” components *pSrcLow* and detail “high-frequency” components *pSrcHigh*, and stores the reconstructed signal in the *lenDst* -length vector *pDst*.

Example 7-10 illustrates the use of the function `ippsWTHaarFwd_32f`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than 4 for the function `ippsWinBlackmanOpt` and less than 3 for all other functions of the family.

Example 7-10 Using the ippsWTHaarFwd Function

```
IppStatus wthaar(void) {
      Ipp32f x[8], lo[4], hi[4];
      IppStatus status;
      ippsSet_32f(7, x, 8);  --x[4];
      status = ippsWTHaarFwd_32f(x, 8, lo, hi);
      printf_32f("WT Haar low  =", lo, 4, status);
      printf_32f("WT Haar high =", hi, 4, status);
      return status;
}
Output:
    WT Haar low  =  7.000000 7.000000 6.500000 7.000000
    WT Haar high =  0.000000 0.000000 0.500000 0.000000
```

See Also

For more information on wavelet transforms see [Strang96] and [Brislawn94].

Transforms for User Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for user filter banks.

WTFwdInitAlloc, WTInvInitAlloc

Initializes the wavelet transform structure.

Syntax

Case 1: Forward transform

```
IppStatus ippsWTFwdInitAlloc_32f(IppsWTFwdState_32f** ppState, const Ipp32f*
    pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int
    offsHigh);
IppStatus ippsWTFwdInitAlloc_8s32f(IppsWTFwdState_8s32f** ppState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_8u32f(IppsWTFwdState_8u32f** ppState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_16s32f(IppsWTFwdState_16s32f** ppState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_16u32f(IppsWTFwdState_16u32f** ppState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
```

Case 2: Inverse transform

```
IppStatus ippsWTInvInitAlloc_32f(IppsWTInvState_32f** ppState, const Ipp32f*
    pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int
    offsHigh);
IppStatus ippsWTInvInitAlloc_32f8s(IppsWTInvState_32f8s** ppState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInitAlloc_32f8u(IppsWTInvState_32f8u** ppState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f16s(IppsWTInvState_32f16s** ppState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f16u(IppsWTInvState_32f16u** ppState, const
    Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
    lenHigh, int offsHigh);
```

Parameters

pState	Pointer to the pointer to the allocated and initialized state structure.
pTapsLow	Pointer to the vector of lowpass filter taps.
lenLow	Number of taps in the lowpass filter.
offsLow	Additional delay (offset) of the lowpass filter.
pTapsHigh	Pointer to the vector of highpass filter taps.
lenHigh	Number of taps in the highpass filter.
offsHigh	Additional delay (offset) of the highpass filter.

Description

The functions `ippsWTFwdInitAlloc` and `ippsWTInvInitAlloc` are declared in the `ipps.h` file. These functions create and initialize the WT state structure *ppState* with the following parameters: the lowpass and highpass filter taps *pTapsLow* and *pTapsHigh*, lengths *lenLow* and *lenHigh*, input additional delays *offsLow* and *offsHigh*, respectively.

ippsWTFwdInitAlloc. The function `ippsWTFwdInitAlloc` initializes the forward WT state structure.

ippsWTInvInitAlloc. The function `ippsWTInvInitAlloc` initializes the inverse WT state structure.

Application Notes

These functions allocate memory for the wavelet state structure, initialize it, and returns the *ppState* pointer to the state structure. The initialization procedures are implemented separately for forward and inverse transforms. To perform both forward and inverse wavelet transforms, create two separate state structures. In general, the meanings of

initialization parameters of forward and inverse transforms are similar. Each function has parameters describing of a pair of filters. The forward transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of analysis filters. The inverse transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of synthesis filters. Besides lengths and sets of taps the functions allow to specify an additional delay *offsLow* and *offsHigh* for each filter. The adjustable values of delays allow to synchronize:

- group delays for highpass and lowpass filters;
- delays between data of different levels in multilevel decomposition and reconstruction algorithms.

For more information about using these parameters, see descriptions of the functions ippsWTFwd and ippsWTInv in this chapter. The minimum allowed value of the additional delay for the forward transform is -1. For the inverse transform the delay values must be greater or equal to 0. See descriptions of the functions ippsWTFwd and ippsWTInv in this chapter for an example showing how to choose additional delay values. The initialization functions copy filter taps into the state structure *pState*. So all the memory referred to with the pointers can be freed or modified after the functions finished operating. In case of the memory shortage, the function sets a zero pointer to the structure.

Boundaries extrapolation. Typically, reversible wavelet transforms of a bounded signal require data extrapolation towards one or both sides. All internal delay lines are set to zero at the stage of initialization. To set a non-zero signal prehistory, call the function ippsWTFwdSetDlyLine. When processed an entire limited data set, data extrapolation may be performed both towards the start and the end of the data vector. For that, the source data and their initial extrapolation are used to form the delay line, the rest of the signal is subdivided into the main block and the signal end. The signal end data and their extrapolation are used to form the last block.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *ppState*, *pTapsHigh*, or *pTapsLow* pointer is NULL.
ippStsSizeErr	Indicates an error when *lenLow* or *lenHigh* is less than or equal to 0.
ippStsWtOffsetErr	Indicates an error when the filter delay *offsLow* or *offsHigh* is less than -1 for the forward transform; and is less than 0 for the inverse transform.

WTFwdFree, WTInvFree

Closes a wavelet transform structure.

Syntax

Case 1: Forward transform

```
IppStatus ippsWTFwdFree_32f(IppsWTFwdState_32f* pState);
IppStatus ippsWTFwdFree_8s32f(IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwdFree_8u32f(IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwdFree_16s32f(IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwdFree_16u32f(IppsWTFwdState_16u32f* pState);
```

Case 2: Inverse transform

```
IppStatus ippsWTInvFree_32f(IppsWTInvState_32f* pState);
IppStatus ippsWTInvFree_32f8s(IppsWTInvState_32f8s* pState);
IppStatus ippsWTInvFree_32f8u(IppsWTInvState_32f8u* pState);
IppStatus ippsWTInvFree_32f16s (IppsWTInvState_32f16s* pState);
IppStatus ippsWTInvFree_32f16u(IppsWTInvState_32f16u* pState);
```

Parameters

pState	Pointer to the state structure to be closed.

Description

The functions `ippsWTFwdFree` and `ippsWTInvFree` are declared in the `ipps.h` file. These functions close the WT state structure *pState* by freeing all the internal memory associated with the state created by `ippsWTFwdInitAlloc` or `ippsWTInvInitAlloc`. Call either `ippsWTFwdFree` or `ippsWTInvFree` after the transform is completed. If the *pState* pointer is `NULL`, the function performs no operation and returns the `ippStsNullPtrErr` status.

ippsWTFwdFree. The function `ippsWTFwdFree` closes the forward WT state structure.

ippsWTInvFree. The function `ippsWTInvFree` closes the inverse WT state structure.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pState* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

WTFwd

Computes the forward wavelet transform.

Syntax

```
IppStatus ippsWTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDstLow, Ipp32f* pDstHigh,
    int dstLen, IppsWTFwdState_32f* pState);
IppStatus ippsWTFwd_8s32f(const Ipp8s* pSrc, Ipp32f* pDstLow, Ipp32f* pDstHigh,
    int dstLen, IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwd_8u32f(const Ipp8u* pSrc, Ipp32f* pDstLow, Ipp32f* pDstHigh,
    int dstLen, IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwd_16s32f(const Ipp16s* pSrc, Ipp32f* pDstLow, Ipp32f*
    pDstHigh, int dstLen, IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwd_16u32f(const Ipp16u* pSrc, Ipp32f* pDstLow, Ipp32f*
    pDstHigh, int dstLen, IppsWTFwdState_16u32f* pState);
```

Parameters

pSrc	Pointer to the vector which holds the input signal for decomposition.
pDstLow	Pointer to the vector which holds output coarse “low frequency” components.
pDstHigh	Pointer to the vector which holds output detail “high frequency” components.
dstLen	Number of elements in the vectors *pDstHigh* and *pDstLow*.
pState	Pointer to the state structure.

Description

The function ippsWTFwd is declared in the ipps.h file. This function computes the forward wavelet transform. The function transforms the (2 * *dstLen*) -length source data block *pSrc* into “low frequency” components *pDstLow* and “high frequency” components *pDstHigh*. The transform parameters are specified in the state structure *pState*.

Application Notes

These functions perform the one-level forward discrete multi-scale transform. An equivalent transform diagram is shown in Figure 7-4. The input signal is divided into the "low frequency" and "high frequency" components. The transfer characteristics of filters are defined by the coefficients set at the initialization stage. The functions are designed for the block processing of data; the transform state structure *pState* contains all needed filter delay lines. Besides these main delay lines each function has an additional delay line for each filter. Adjustable extra delay lines help synchronize group delay times of both highpass and lowpass filters. Moreover, in multilevel systems of signal decomposition delays between different decomposition levels may also be synchronized.

Input and output data block lengths. The functions are designed to decompose signal blocks of even length, therefore, these functions have one parameter only, i.e. the length of input components. The length of the input block must be double the size of each component.

Filter group delays synchronization. Some applications may require synchronization of highpass and lowpass filter time responses. A typical example of this synchronization is synchronizing symmetrical filters of different length.

Below follows an example of bi orthogonal set of spline filters of respective length of 6 and 2:

```
static const float decLow[6] =
{
    -6.25000000e-002f,
     6.25000000e-002f,
     5.00000000e-001f,
     5.00000000e-001f,
     6.25000000e-002f,
     -6.25000000e-002f
};


static const float decHigh[2] =
{
    -5.00000000e-001f,
     5.00000000e-001f
};
```

In this case the lowpass filter gives a delay two samples longer than the highpass filter, which is exactly what the difference between additional initialization function delays should be. The following values must be selected to ensure minimum common signal delay,

offsLow = -1, *offsHigh* = -1 + 2 = 1. In this case the group times of filter delays are balanced by additional delays. The total delay time is equal to the lowpass filter group delay which has the value of two samples in the decomposition stage in the original signal time frame.

NOTE. Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, forward transform additional delays must be uniformly even for faultless signal reconstruction.

Multilevel decomposition algorithm. The implementation of multilevel decomposition algorithms may require synchronization of signal delays across components of different levels.

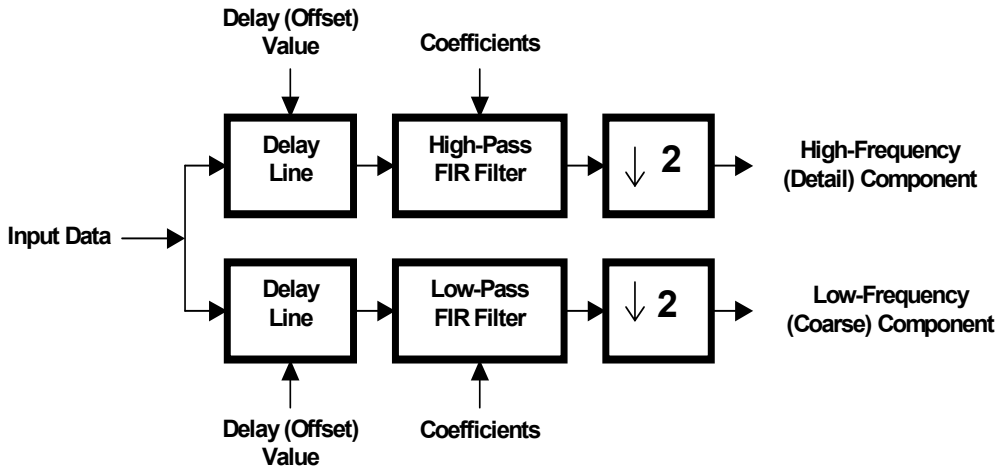
This is illustrated in the example of the three-level decomposition shown in Figure 7-2. Assume that for transformation the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. Since group delay definitely needs to be synchronized, for the last level select additional filter delays *offsLow3* = -1, *offsHigh3* = 1. Total delay at the last stage of decomposition for this set of filters is two samples. This value corresponds to the time scale of the input of the last stage of decomposition. In order to ensure an equivalent delay of the “detail” part on the second level, the delay must be increased by 2*2 samples. Respective values of additional delays for the second level is equal to *offsLow2* = -1, *offsHigh2* = *offsHigh3* + 4 = 5. A greater value of the “high frequency” component delay needs to be selected for the first level of decomposition, *offsLow1* = -1, *offsHigh1* = *offsHigh2* + 2*4 = 13.

Total delay for three levels of decomposition is equal to 12 samples.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicatesanerrorwhenoneofthespecifiedpointersis NULL.
ippStsContextMatchErr	Indicates an error when the state identifier *pState* is incorrect.
ippStsSizeErr	Indicates an error when *dstLen* or *srcLen* is less than or equal to 0.

Figure 7-4 One Level Forward Wavelet Transform



WTFwdSetDlyLine, WTFwdGetDlyLine

Sets and gets the delay lines of the forward wavelet transform.

Syntax

```
IppStatus ippsWTFwdSetDlyLine_32f(IppsWTFwdState_32f* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8s32f(IppsWTFwdState_8s32f* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8u32f(IppsWTFwdState_8u32f* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_16s32f(IppsWTFwdState_16s32f* pState, const
    Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTFwdSetDlyLine_16u32f(IppsWTFwdState_16u32f* pState, const
    Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);


IppStatus ippsWTFwdGetDlyLine_32f(IppsWTFwdState_32f* pState, Ipp32f* pDlyLow,
    Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_8s32f(IppsWTFwdState_8s32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_8u32f(IppsWTFwdState_8u32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_16s32f(IppsWTFwdState_16s32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_16u32f(IppsWTFwdState_16u32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
```

Parameters

pState	Pointer to the state structure.
pDlyLow	Pointer to the vector which holds the delay lines for “low frequency” components.
pDlyHigh	Pointer to the vector which holds the delay lines for “high frequency” components.

Description

The functions `ippsWTFwdSetDlyLine` and `ippsWTFwdGetDlyLine` are declared in the `ipps.h` file. These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

ippsWTFwdSetDlyLine. The functions `ippsWTFwdSetDlyLine` sets the delay line values of the forward WT state.

ippsWTFwdGetDlyLine. The functions `ippsWTFwdGetDlyLine` gets the delay line values of the forward WT state.

Application Notes

These functions are designed to shape the signal prehistory, save and reconstruct delay lines. Delay lines are implemented separately for highpass and lowpass filters, which gives the option of getting independent signal prehistories for each filter.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the initial signal fed into the forward transform functions, i.e., delay line vectors must be made up of a succession of the signal prehistory counts in the same time frame as the initial signal.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter:

dlyLowLen = *lenLow* + *offsLow* - 1,

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter:

dlyHighLen = *lenHigh* + *offsHigh* - 1,

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the “high frequency” component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are specified by the functions ippsWTFwdInitAlloc and ippsWTInvInitAlloc.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDlyLow* or *pDlyHigh* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

WTInv

Computes the inverse wavelet transform.

Syntax

```
IppStatus ippsWTInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp32f* pDst, IppsWTInvState_32f* pState);
IppStatus ippsWTInv_32f8s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp8s* pDst, IppsWTInvState_32f8s* pState);
IppStatus ippsWTInv_32f8u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp8u* pDst, IppsWTInvState_32f8u* pState);
IppStatus ippsWTInv_32f16s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp16s* pDst, IppsWTInvState_32f16s* pState);
IppStatus ippsWTInv_32f16u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
    srcLen, Ipp16u* pDst, IppsWTInvState_32f16u* pState);
```

Parameters

pSrcLow	Pointer to the vector which holds input coarse “low frequency” components.
pSrcHigh	Pointer to the vector which holds detail “high frequency” components.
srcLen	Number of elements in the vectors *pSrcHigh* and *pSrcLow*.
pDst	Pointer to the vector which holds the output reconstructed signal.
pState	Pointer to the state structure.

Description

The function `ippsWTInv` is declared in the `ipps.h` file. This function computes the inverse wavelet transform. The function transforms the “low frequency” components *pSrcLow* and “high frequency” components *pSrcHigh* into the (2 * *srcLen*)-length destination data block *pDst*. The transform parameters are specified in the state structure *pState*.

Application Notes

These functions are used for one level of inverse multiscale transformation which results in reconstructing the original signal from the two “low frequency” and “high frequency” components. Figure 7-5 shows an equivalent transform algorithm. Two interpolation filters

are used for signal reconstruction; their coefficients are set at the initialization stage. The inverse transform implementation, similar to forward transform implementation, contains additional delay lines needed to synchronize the group time of filter delays and delays across different levels of data reconstruction.

Input and output data block lengths. These functions are designed to reconstruct the blocks of the even length signal. The signal component length must be the input data. The length of the output block of the reconstructed signal must be double the length of each of the components.

Filter group delay synchronization. In this example consider a biorthogonal set of spline filters of length 2 and 6:

```
static const float recLow[2] =
{
     1.00000000e+000f,
     1.00000000e+000f
};
static const float recHigh[6] =
{
    -1.25000000e-001f,
    -1.25000000e-001f,
     1.00000000e+000f,
    -1.00000000e+000f,
     1.25000000e-001f,
     1.25000000e-001f
};
```

This set of filters corresponds to the set of filters considered in a similar section of the description of the forward transform function ippsWTFwd.

Unlike the case described above, this time the highpass filter generates a delay greater by two samples compared against the low frequency filter. The two sample difference should also exist between initialization function additional delays. The following parameters of additional delays need to be selected in order to ensure the minimum total delay, *offsLow* = 2, *offsHigh* = 0. In this case the total delay is equal to the highpass filter group delay, which at the decomposition stage is equal to two samples in the original signal time frame.

Total delay of one level of decomposition and reconstruction is equal to 4 samples, considering the decomposition stage delay.

NOTE. Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, inverse transform additional delays must be uniformly even and opposite to the evenness of the decomposition delays for faultless signal reconstruction.

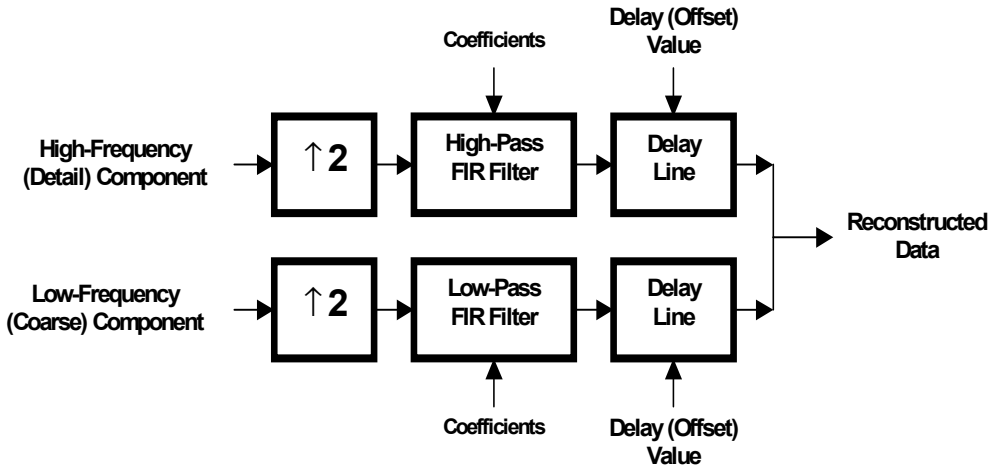
Multilevel reconstruction algorithms. An example of a three-level signal reconstruction algorithm is shown in Figure 7-3. The scheme corresponds to the decomposition scheme described in the section of the description of the forward transform function `ippsWTFwd`. Therefore, for the inverse transform the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. The lowest level filter delays are set to *offsLow3* = 2, *offsHigh3* = 0. The total delay at this stage of reconstruction is equal to two samples. In order to ensure an equivalent delay of the "detail" part in the middle level, the delay must be increased. Respective values of additional delays for the second level are equal to *offsLow2* = 2, *offsHigh2* = *offsHigh3* + 2*2 = 4. A greater value of high frequency component delay needs to be selected for the last level of reconstruction, *offsLow1* = -1, *offsHigh1* = *offsHigh2* + 2*4 = 12.

The total delay for three levels of reconstruction is equal to 12 samples. The total delay of the three-level decomposition and reconstruction cycle is equal to 24 samples.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsStateMatchErr`	Indicates an error when the state identifier *pState* is incorrect.
`ippStsSizeErr`	Indicates an error when *dstLen* or *srcLen* is less than or equal to 0.

Figure 7-5 One Level Inverse Wavelet Transform



WTInvSetDlyLine, WTInvGetDlyLine

Sets and gets the delay lines of the inverse wavelet transform.

Syntax

```
IppStatus ippsWTInvSetDlyLine_32f(IppsWTInvState_32f* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8s(IppsWTInvState_32f8s* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8u(IppsWTInvState_32f8u* pState, const Ipp32f*
    pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f16s(IppsWTInvState_32f16s* pState, const
    Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTInvSetDlyLine_32f16u(IppsWTInvState_32f16u* pState, const
    Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTInvGetDlyLine_32f(IppsWTInvState_32f* pState, Ipp32f* pDlyLow,
    Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f8s(IppsWTInvState_32f8s* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f8u(IppsWTInvState_32f8u* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f16s(IppsWTInvState_32f16s* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f16u(IppsWTInvState_32f16u* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
```

Parameters

pState	Pointer to the state structure.
pDlyLow	Pointer to the vector which holds delay lines for “low frequency” components.
pDlyHigh	Pointer to the vector which holds delay lines for “high frequency” components.

Description

The functions `ippsWTInvSetDlyLine` and `ippsWTInvGetDlyLine` are declared in the `ipps.h` file. These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

ippsWTInvSetDlyLine. The functions `ippsWTInvSetDlyLine` sets the delay line values of the inverse WT state.

ippsWTInvGetDlyLine. The functions `ippsWTInvGetDlyLine` gets the delay line values of the inverse WT state.

Application Notes

These functions set and read delay lines of inverse multiscale transformation. The functions receive or return filter low and high frequency component delay line vectors. The functions may be used to shape previous history of each of the components. Installation functions and read functions together ensure that delay lines from each filter are saved and reconstructed.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the low and high frequency components at the input of the inverse transform functions. Thus, delay line vectors must be made up of a succession of signal prehistory counts in the same time frame as the input components.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter in terms of the C language (integer division by two is used here for simplicity):

dlyLowLen = (*lenLow* + *offsLow* - 1) / 2,

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter in terms of the C language:

dlyHighLen = (*lenHigh* + *offsHigh* - 1) / 2,

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the “high frequency” component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are also described in the section on the transform initialization, see the functions ippsWTFwdInitAlloc, ippsWTInvInitAlloc in this chapter.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDlyLow* or *pDlyHigh* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

Wavelet Transforms Example

The delay line paradigm is well-known interface solution for functions that require some pre-history in the streaming processing. In such application the use of Intel IPP wavelet transform functions is similar to the use of the FIR, IIR, or multi-rate filters. (See also the discussion on the synchronization of low-pass and high-pass filter delays in this chapter.) But very often the wavelet transforms are used to process entire non-streaming data by extending with borders that are suitable for filter bank type that are used in transforms.

The following code example demonstrates how to implement this approach using Intel IPP functions. It performs forward and inverse wavelet transforms of a short vector containing 12 elements. It uses Daubechies filter bank of the order 2 (that allows the perfect reconstruction) and periodical data extension by wrapping.

It is also may be useful as an illustration of how to fill delay line, if you need non-zero pre-history of signal in streaming applications.

Example 7-11 Using Wavelet Transforms Functions

```
#include <stdio.h>
#include "ipp.h"

//   Filter bank for Daubechies, order 2

static const int fwdFltLenL = 4;
static const int fwdFltLenH = 4;

static const float fwdFltL[4] =
{
   -1.294095225509215e-001f,
    2.241438680418574e-001f,
    8.365163037374690e-001f,
    4.829629131446903e-001f
};

static const float fwdFltH[4] =
{
   -4.829629131446903e-001f,
    8.365163037374690e-001f,
   -2.241438680418574e-001f,
   -1.294095225509215e-001f
};
```

Example 7-11 Using Wavelet Transforms Functions

```
static const int invFltLenL = 4;
static const int invFltLenH = 4;

static const float invFltL[4] =
{
    4.829629131446903e-001f,
    8.365163037374690e-001f,
    2.241438680418574e-001f,
   -1.294095225509215e-001f
};

static const float invFltH[4] =
{
   -1.294095225509215e-001f,
   -2.241438680418574e-001f,
    8.365163037374690e-001f,
   -4.829629131446903e-001f
};

// minimal values
static const int fwdFltOffsL = -1;
static const int fwdFltOffsH = -1;

// minimal values, that corresponds to perfect reconstruction
static const int invFltOffsL = 0;
static const int invFltOffsH = 0;
void main(void)
{
    Ipp32f src[] = {1, -10, 324, 48, -483, 4, 7, -5532, 34, 8889, -57, 54};

    Ipp32f low[7];
    Ipp32f high[7];

    int i;

    printf("original:\n");
    for(i = 0; i < 12; i++) printf("%.0f; ", src[i]);
    printf("\n");

    {
    IppsWTFwdState_32f* state;
```

Example 7-11 Using Wavelet Transforms Functions

```
    ippsWTFwdInitAlloc_32f (&state, fwdFltL, fwdFltLenL, fwdFltOffsL,
                                    fwdFltH, fwdFltLenH, fwdFltOffsH);

    // We substitute wrapping extension in "the beginning of stream"
    // Here should be the same pointers for this offsets,
    // but in the general case it may be different

    ippsWTFwdSetDlyLine_32f( state, &src[10], &src[10]);

    // Forward transform

    ippsWTFwd_32f          (src, low, high, 6, state);

    ippsWTFwdFree_32f      (state);

    // print decomposition result

    printf("approx:\n");
    for(i = 0; i < 6; i++) printf("%.4f; ", low[i]);
    printf("\n");

    printf("details:\n");
    for(i = 0; i < 6; i++) printf("%.4f; ", high[i]);
    printf("\n");

    }
   {
    Ipp32f dst[12];

    IppsWTInvState_32f* state;

    ippsWTInvInitAlloc_32f (&state,
        invFltL, invFltLenL, invFltOffsL,
        invFltH, invFltLenH, invFltOffsH);

    // For this particular case (non-shifted reconstruction)
    // here is first data itself,
    // that we need to place to delay line
    // [(invFltLenL + invFltOffsL - 1) / 2] elements for l. filtering
    // [(invFltLenH + invFltOffsH - 1) / 2] elements for h. filtering

    ippsWTInvSetDlyLine_32f( state, low, high);
```

Example 7-11 Using Wavelet Transforms Functions

```
    // Inverse transform
    ippsWTInv_32f          (&low[1], &high[1], 5, dst, state);

    // Here are the substitution of the wrapping extension
    // at the "end of stream" and calculation of last samples of reconstruction
    // We do not use additional buffer and do not copy any data externally,
    // just substitute beginning of input data itself to simulate wrapping

    ippsWTInv_32f          (low, high, 1, &dst[10], state);

    ippsWTInvFree_32f      (state);

    // print reconstruction result
    printf("reconstruction:\n");
    for(i = 0; i < 12; i++) printf("%.0f; ", dst[i]);
    printf("\n");
    }
}
```

After compiling and running it gives the following console output:

```
original:
1; -10; 324; 48; -483; 4; 7; -5532; 34; 8889; -57; 54;
approx:
19.1612; 58.5288; 87.8536; 487.5375; -5766.9277; 7432.4502;
details:
0.9387; 249.9611; -458.6568; 2739.2146; -3025.5576; -2070.5762;
reconstruction:
1; -10; 324; 48; -483; 4; 7; -5532; 34; 8889; -57; 54;
```

The program prints on console the original data, approximation, and details components after forward transform and perfect reconstruction of original data after inverse transform.

Speech Recognition Functions

8

This chapter describes Intel® IPP functions that are designed for use in speech recognition applications.

Table 8-1 gives the full list of functions in this group.

Table 8-1 Intel IPP Speech Recognition Functions

Function Base Name	Operation
Basic Arithmetics Functions	
AddAllRowSum	Calculates the sums of column vectors in a matrix and adds the sums to a vector.
SumColumn	Calculates sums of column vectors in a matrix.
SumRow	Calculates sums of row vectors in a matrix.
SubRow	Subtracts a vector from all matrix rows.
CopyColumn_Indirect	Copies the input matrix with columns redirection.
BlockDMatrixInitAlloc	Initializes the structure that represents a symmetric block diagonal matrix.
BlockDMatrixFree	Deallocates the block diagonal matrix structure.
NthMaxElement	Searches for the N-th maximal element of a vector.
VecMatMul	Multiplies a vector by a matrix.
MatVecMul	Multiplies a matrix by a vector
Feature Processing Functions	
ZeroMean	Subtracts the mean value from the input vector.
CompensateOffset	Removes the DC offset of the input signals.
SignChangeRate	Counts the zero-cross rate for the input signal.
LinearPrediction	Performs linear prediction analysis on the input vector.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
Durbin	Performs Durbin's recursion on an input vector of autocorrelations.
Schur	Calculates reflection coefficients using Schur algorithm.
LPToSpectrum	Calculates smoothed magnitude spectrum.
LPToCepstrum	Calculates cepstrum coefficients from linear prediction coefficients.
CepstrumToLP	Calculates linear prediction coefficients from cepstrum coefficients.
LPToReflection	Calculates the linear prediction reflection coefficients from the linear prediction coefficients.
ReflectionToLP	Calculates the linear prediction coefficients from the linear prediction reflection coefficients.
ReflectionToAR	Converts reflection coefficients to area ratios.
ReflectionToTilt	Calculates tilt for rise/fall/connection parameters.
PitchmarkToF0	Calculates rise and fall amplitude and duration for tilt.
UnitCurve	Calculates tilt for rise and fall coefficicents.
LPToLSP	Calculates line spectrum pairs vector from linear prediction coefficients.
LSPToLP	Converts line spectrum pairs vector to linear prediction coefficients.
MelToLinear	Converts Mel-scaled values to linear scale values.
LinearToMel	Converts linear-scale values to Mel-scale values.
CopyWithPadding	Copies the input signal to the output with zero-padding.
MelFBankGetSize	Gets the size of the Mel-frequency filter bank structure.
MelFBankInit	Initializes the structure for performing the Mel-frequency filter bank analysis.
MelFBankInitAlloc	Initializes the structure and allocates memory for performing the Mel-frequency filter bank analysis.
MelLinFBankInitAlloc	Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.
EmptyFBankInitAlloc	Initializes an empty filter bank structure.
FBankFree	Destroys the structure for the filter bank analysis.
FBankGetCenters	Retrieves the center frequencies of the triangular filter banks.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
FBankSetCenters	Sets the center frequencies of the triangular filter banks.
FBankGetCoeffs	Retrieves the filter bank weight coefficients.
FBankSetCoeffs	Sets the filter bank weight coefficients.
EvalFBank	Performs the filter bank analysis.
DCTLifterGetSize_MulC0	Gets the size of the DCT structure.
DCTLifterInit_MulC0	Initializes the structure to perform DCT and lift the DCT coefficients.
DCTLifterInitAlloc	Initializes the structure and allocates memory to perform DCT and lift the DCT coefficients.
DCTLifterFree	Destroys the structure used for the DCT and liftering.
DCTLifter	Performs the DCT and lifts the DCT coefficients.
NormEnergy	Normalizes a vector of energy values.
SumMeanVar	Calculates both the sum of a the vector and its square sum.
NewVar	Calculates the variances given the sum and square sum accumulators.
RecSqrt	Calculates square roots of a vector and their reciprocals.
AccCovarianceMatrix	Accumulates covariance matrix.
Derivative Functions	
CopyColumn	Copies the input sequence into the output sequence.
EvalDelta	Calculates the derivatives of feature vectors.
Delta	Copies the base features and calculates the derivatives of feature vectors.
DeltaDelta	Copies the base features and calculates their first and second derivatives.
Pitch Super Resolution Functions	
CrossCorrCoeffDecim	Calculates vector of cross correlation coefficients with decimation.
CrossCorrCoeff	Calculates the cross correlation coefficient.
CrossCorrCoeffInterpolation	Calculates interpolated cross correlation coefficient.
Model Evaluation Functions	
AddNRows	Adds N vectors from a vector array.
ScaleLM	Scales vector elements with thresholding.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
LogAdd	Adds two vectors in the logarithmic representation.
LogSub	Subtracts a vector from another vector, in the logarithmic representation.
LogSum	Sums vector elements in the logarithmic representation.
MahDistSingle	Calculates the Mahalanobis distance for a single observation vector.
MahDist	Calculates the Mahalanobis distances for multiple observation vectors.
MahDistMultiMix	Calculates the Mahalanobis distances for multiple means and variances.
LogGaussSingle	Calculates the observation probability for a single Gaussian with an observation vector.
LogGauss	Calculates the observation probability for a single Gaussian with multiple observation vectors.
LogGaussMultiMix	Calculates the observation probability for multiple Gaussian mixture components.
LogGaussMax	Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.
LogGaussMaxMultiMix	Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.
LogGaussAdd	Calculates the likelihood probability for multiple observation vectors.
LogGaussAddMultiMix	Calculates the likelihood probability for multiple Gaussian mixture components.
LogGaussMixture	Calculates the likelihood probability for the Gaussian mixture.
LogGaussMixtureSelect	Calculates the likelihood probability for the Gaussian mixture using Gaussian selection.
BuildSignTable	Fills sign table for Gaussian mixture calculation.
FillShortlist_Row	Fills row-wise shortlist table for Gaussian selection.
FillShortlist_Column	Fills column-wise shortlist table for Gaussian selection.
DTW	Computes the distance between observation and reference vector sequences using Dynamic Time Warping algorithm.
Model Estimation Functions	
MeanColumn	Computes the mean values for the column elements.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
VarColumn	Calculates the variances for the column elements.
MeanVarColumn	Calculates the means and variances for the column elements of a matrix.
WeightedMeanColumn	Computes the weighted mean values for the column elements.
WeightedVarColumn	Computes the weighted variance values for the column elements.
WeightedMeanVarColumn	Computes weighted mean and variance values for the column elements.
NormalizeColumn	Normalizes the matrix columns given the column means and variances.
NormalizeInRange	Normalizes and scales input vector elements.
MeanVarAcc	Accumulates the estimates for the mean and variance re-estimation.
GaussianDist	Calculates the distance between two Gaussians.
GaussianSplit	Splits a single Gaussian component into two with the same variance.
GaussianMerge	Merges two Gaussian probability distribution functions.
Entropy	Calculates entropy of the input vector.
SinC	Calculates sine divided by its argument.
ExpNegSqr	Calculates exponential of the squared argument taken with the inverted sign.
BhatDist	Calculates the Bhattacharia distance between two Gaussians.
UpdateMean	Updates the mean vector in the EM training algorithm.
UpdateVar	Updates the variance vector in the EM training algorithm.
UpdateWeight	Updates the weight values of Gaussian mixtures in the EM training algorithm.
UpdateGConst	Updates the fixed constant in the Gaussian output probability density function.
OutProbPreCalc	Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.
DcsClustLAccumulate	Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.
DcsClustLCompute	Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
Model Adaptation Functions	
AddMulColumn	Adds a weighted matrix column to the other column.
AddMulRow	Adds a weighted vector to the other vector.
QRTransColumn	Performs the QR transformation.
DotProdColumn	Calculates the dot product of two matrix columns.
MulColumn	Multiplies a matrix column by a value.
SumColumnAbs	Calculates the absolute sum of matrix column elements.
SumColumnSqr	Calculates the square sums of weighted matrix column elements.
SumRowAbs	Calculates the absolute sum of the vector elements.
SumRowSqr	Calculates the square sum of weighted vector elements.
SVD, SVDSort	Performs Single Value Decomposition on a matrix.
WeightedSum	Calculates the weighted sums of two input vector elements.
Vector Quantization Functions	
FormVector	Constructs an output vector of multiple streams from codebook entries.
CdbkGetSize	Calculates the size in bytes of the codebook.
CdbkInit	Initializes the structure that contains the codebook.
CdbkInitAlloc	Initializes the codebook structure.
CdbkFree	Destroys the codebook structure.
GetCdbkSize	Retrieves the number of codevectors in the codebook.
GetCodebook	Retrieves the codevectors from the codebook.
VQ	Quantizes the input vectors given a codebook.
SplitVQ	Quantizes a multiple-stream vector given the codebooks.
VQSingle_Sort, VQSingle_Thresh	Quantizes the input vector given a codebook and gets several closest clusters.
FormVectorVQ	Constructs multiple-stream vectors from codebooks, given indexes.
Polyphase Resampling Functions	
ResamplePolyphaseInitAlloc	Initializes the structure for polyphase data resampling.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
ResamplePolyphaseFree	Free structure for polyphase data resampling.
ResamplePolyphase	Resamples input data using polyphase filters.
Advanced Aurora Functions	
SmoothedPowerSpectrumAurora	Calculates smoothed magnitude of the FFT output.
NoiseSpectrumUpdate_Aurora	Updates the noise spectrum.
WienerFilterDesign_Aurora	Calculates an improved transfer function of the adaptive Wiener filter.
MelFBankInitAlloc_Aurora	Initializes the structure for performing the Mel-frequency filter bank analysis.
TabsCalculation_Aurora	Calculates filter coefficients for residual filter.
ResidualFilter_Aurora	Calculates a denoised waveform signal.
WaveProcessing_Aurora	Processes waveform data after noise reduction.
LowHighFilter_Aurora	Calculates low band and high band filters.
HighBandCoding_Aurora	Codes and decodes the high frequency band energy values.
BlindEqualization_Aurora	Equalizes the cepstral coefficients.
DeltaDelta_Aurora	Calculates the first and second derivatives according to ETSI ES 202 050 standard.
VADGetBufSize_Aurora	Queries the memory size for VAD decision.
VADInit_Aurora	Gets the VAD structure size.
VADDecision_Aurora	Takes the VAD decision.
VADFlush_Aurora	Takes VAD decision for zero input frame.
Ephraim-Malah Noise Suppressor Functions	
FilterUpdateEMNS	Calculates the noise suppression filter coefficients.
FilterUpdateWiener	Calculates the Wiener filter coefficients.
GetSizeMCRA	Calculates the size in bytes required for the state structure.
InitMCRA	Initializes the `IppMCRAState` state structure.
AltInitMCRA	Allocates memory and initializes the `IppMCRAState` state structure.
UpdateNoisePSDMCRA	Re-estimates the noise power spectrum.
Voice Activity Detector Functions	
FindPeaks	Identifies peaks in the input vector.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
PeriodicityLSPE	Computes the periodicity of the input speech frame.
Periodicity	Computes the periodicity of the input block.

NOTE. In the below descriptions of function arguments, when an argument refers to a vector or an array, the expression in square brackets given after the explanation of the argument specifies the dimension of that vector or array.
In this chapter, unlike other chapters of this manual, arguments that represent step values are measured in elements of the corresponding array, unless explicitly stated to the contrary.

The use of the Intel IPP speech recognition functions is demonstrated in Intel IPP Samples. See *Intel IPP Speech Recognition Samples* downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

Basic Arithmetics

The functions included in this section perform generic arithmetic operations on vectors and matrices.

AddAllRowSum

Calculates the sums of column vectors in a matrix and adds the sums to a vector.

Syntax

```
IppStatus ippsAddAllRowSum_32f_D2(const Ipp32f* pSrc, int step,
    int height, Ipp32f* pSrcDst, int width);
IppStatus ippsAddAllRowSum_32f_D2L(const Ipp32f** mSrc, int height,
    Ipp32f* pSrcDst, int width);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in *pSrc*.
height	Number of rows in the input matrix *mSrc*.
pSrcDst	Pointer to the output vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the output vector *pSrcDst*.

Description

The function ippsAddAllRowSum is declared in the ippsr.h file. This function calculates sums of column vectors in the input matrix, and adds these sums to the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} pSrc[i \cdot step + j], \quad 0 \le j < width .$$

For functions with the D2L suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} mSrc[i][j], \quad 0 \le j < width .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

SumColumn

Calculates sums of column vectors in a matrix.

Syntax

```
IppStatus ippsSumColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int step, int height,
    Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumColumn_16s32f_D2(const Ipp16s* pSrc, int step, int height,
    Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2(const Ipp32f* pSrc, int step, int height,
    Ipp32f* pDst, int width);
IppStatus ippsSumColumn_64f_D2(const Ipp64f* pSrc, int step, int height,
    Ipp64f* pDst, int width);
IppStatus ippsSumColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
    Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumColumn_16s32f_D2L(const Ipp16s** mSrc, int height,
    Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f* pDst,
    int width);
IppStatus ippsSumColumn_64f_D2L(const Ipp64f** mSrc, int height, Ipp64f* pDst,
    int width);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
height	Number of rows in the input matrix *mSrc*.
pDst	Pointer to the output vector [*width*].
width	Number of columns in the input matrix *mSrc* and the length of the output vector *pDst*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsSumColumn` is declared in the `ippsr.h` file. This function calculates sums of column vectors in the input matrix, and stores these sums in the output vector. The operations are as follows:

For functions with the `D2` suffix,

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i \cdot step + j], \quad 0 \le j < width .$$

For functions with the `D2L` suffix,

$$pDst[j] = \sum_{i=0}^{height-1} mSrc[i][j], \quad 0 \le j < width .$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *height* or *width* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

SumRow

Calculates sums of row vectors in a matrix.

Syntax

```
IppStatus ippsSumRow_16s32s_D2Sfs(const Ipp16s*  pSrc, int width, int step,
    Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2(const Ipp16s* pSrc, int width, int step, Ipp32f*
    pDst, int height);
IppStatus ippsSumRow_32f_D2(const Ipp32f* pSrc, int width, int step,
    Ipp32f* pDst, int height);
IppStatus ippsSumRow_64f_D2(const Ipp64f* pSrc, int width, int step,
    Ipp64f* pDst, int height);
```

```
IppStatus ippsSumRow_16s32s_D2LSfs(const Ipp16s** mSrc, int width, Ipp32s*
    pDst, int height, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2L(const Ipp16s** mSrc, int width, Ipp32f* pDst,
    int height);
IppStatus ippsSumRow_32f_D2L(const Ipp32f** mSrc, int width, Ipp32f* pDst,
    int height);
IppStatus ippsSumRow_64f_D2L(const Ipp64f** mSrc, int width, Ipp64f* pDst,
    int height);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*, and also the length of the output vector *pDst*.
step	Row step in the input vector *pSrc*.
pDst	Pointer to the output vector [*height*].
width	Number of columns in the input matrix *mSrc*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsSumRow is declared in the ippsr.h file. This function calculates sums of row vectors in the input matrix *mSrc* and stores these sums in the output vector *pDst*. The operations are as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[i \cdot step + j], \quad 0 \le i < height .$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mSrc[i][j], \quad 0 \le i < height .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pDst* pointer is NULL.

ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

SubRow

Subtracts a vector from all matrix rows.

Syntax

```
IppStatus ippsSubRow_16s_D2(const Ipp16s* pSrc, int width, Ipp16s* pSrcDst, int
    dstStep, int height);
IppStatus ippsSubRow_32f_D2(const Ipp32f* pSrc, int width, Ipp32f* pSrcDst,
    int dstStep, int height);
IppStatus ippsSubRow_16s_D2L(const Ipp16s* pSrc, Ipp16s** mSrcDst, int width,
    int height);
IppStatus ippsSubRow_32f_D2L(const Ipp32f* pSrc, Ipp32f** mSrcDst, int width,
    int height);
```

Parameters

pSrc	Pointer to the input vector [*width*].
pSrcDst	Pointer to the source and destination vector [*height*dstStep*].
mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
dstStep	Row step in the vector *pSrcDst*.
height	Number of rows in the matrix *mSrcDst*.

Description

The function ippsSubRow is declared in the ippsr.h file. This function subtracts the input vector *pSrc* from all matrix rows. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] - pSrc[j],$$

$$0 \le i < height,\ 0 \le j < width\ .$$

For functions with the D2L suffix,

$mSrcDst[i][j] = mSrcDst[i][j] - pSrc[j]$,

$0 \le i < height$, $0 \le j < width$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pSrcDst*, or *mSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *dstStep* is less than *width*.

CopyColumn_Indirect

Copies the input matrix with columns redirection.

Syntax

```
IppStatus ippsCopyColumn_Indirect_16s_D2(const Ipp16s* pSrc, int srcLen,
    int srcStep, Ipp16s* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
    height);
IppStatus ippsCopyColumn_Indirect_32f_D2(const Ipp32f* pSrc, int srcLen,
    int srcStep, Ipp32f* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
    height);
IppStatus ippsCopyColumn_Indirect_64f_D2 (const Ipp64f* pSrc, int srcLen, int
    srcStep, Ipp64f* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
    height);
IppStatus ippsCopyColumn_Indirect_16s_D2L(const Ipp16s** mSrc, int srcLen,
    Ipp16s** mDst, const Ipp32s* pIndx, int dstLen, int height);
IppStatus ippsCopyColumn_Indirect_32f_D2L(const Ipp32f** mSrc, int srcLen,
    Ipp32f** mDst, const Ipp32s* pIndx, int dstLen, int height);
IppStatus ippsCopyColumn_Indirect_64f_D2L (const Ipp64f** mSrc, int srcLen,
    Ipp64f** mDst, const Ipp32s* pIndx, int dstLen, int height);
```

Parameters

pSrc	Pointer to the input vector [*height***srcStep*].
mSrc	Pointer to the input matrix [*height*][*srcLen*].

srcLen	Number of columns in the input matrix *mSrc*.
srcStep	Row step in the vector *pSrc*.
pDst	Pointer to the output vector [*height***dstStep*].
mDst	Pointer to the output matrix [*height*][*dstLen*].
pIndx	Pointer to the redirection vector [*dstLen*].
dstLen	Number of columns in the output matrix *mDst*.
dstStep	Row step in the vector *pDst*.
height	Number of rows in both the input and output matrices.

Description

The function `ippsCopyColumn_Indirect` is declared in the `ippsr.h` file. This function copies the input matrix *mSrc* to the output matrix *mDst* with the columns redirected by *pIndx*. The operations are as follows:

For functions with the D2 suffix,

$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcStep + pIndx[j]], \quad 0 \leq i < height, \quad 0 \leq j < dstLen.$

For functions with the D2L suffix,

$mDst[i][j] = mSrc[i][pIndx[j]], \quad 0 \leq i < height, \quad 0 \leq j < dstLen.$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, *pDst*, *mDst*, or *pIndx* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *height*, *srcLen*, or *dstLen* is less than or equal to 0, or $pIndx[j] \geq srcLen$ or $pIndx[j] < 0$ for $0 \leq j < dstLen$.
`ippStsStrideErr`	Indicates an error when *srcStep* is less than *srcLen* or *dstStep* is less than *dstLen*.

BlockDMatrixInitAlloc

Initializes the structure that represents a symmetric block diagonal matrix.

Syntax

```
IppStatus ippsBlockDMatrixInitAlloc_16s(IppsBlockDMatrix_16s** pMatrix,
    const Ipp16s** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_32f(IppsBlockDMatrix_32f** pMatrix,
    const Ipp32f** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_64f(IppsBlockDMatrix_64f** pMatrix,
    const Ipp64f** mSrc, const int* bSize, int nBlocks);
```

Parameters

pMatrix	Pointer to the block diagonal matrix to be created.
mSrc	Pointer to the vector of pointers to matrix rows.
bSize	Pointer to vector of block sizes [*nBlocks*].
nBlocks	Number of blocks in the matrix.

Description

The function `ippsBlocDMatrixInitAlloc` is declared in the `ippsr.h` file. This function creates the structure that contains a symmetric block diagonal matrix. Matrix elements outside the blocks are assumed to be zero. The block diagonal matrix *A* is defined as follows:

$A[K_l + i][K_l + j] = A[K_l + j][K_l + i] = mSrc[K_l + i]\,[j]$,

for $i, j = 0 \ldots bSize[l] - 1$ and $l = 0 \ldots nBlocks - 1$,

where

$$K_l = \sum_{k<l} bSize[k]$$

and $mSrc[K_l + i]$ points to the non-zero part of the matrix *A* row.
The size of the matrix is equal to $K_{nBlocks}$ by $K_{nBlocks}$.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pMatrix* or *mSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *bSize* or *nBlocks* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

BlockDMatrixFree

Deallocates the block diagonal matrix structure.

Syntax

```
IppStatus ippsBlockDMatrixFree_16s(IppsBlockDMatrix_16s* pMatrix);
IppStatus ippsBlockDMatrixFree_32f(IppsBlockDMatrix_32f* pMatrix);
IppStatus ippsBlockDMatrixFree_64f(IppsBlockDMatrix_64f* pMatrix);
```

Parameters

pMatrix	Pointer to the block diagonal matrix.

Description

The function ippsBlockDMatrixFree is declared in the ippsr.h file. This function destroys the block diagonal matrix structure, and frees all memory associated with it.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMatrix* pointer is NULL.

NthMaxElement

Searches for the N-th maximal element of a vector.

Syntax

```
IppStatus ippsNthMaxElement_32s(const Ipp32s* pSrc, int len, int N,
    Ipp32s* pRes);
```

```
IppStatus ippsNthMaxElement_32f(const Ipp32f* pSrc, int len, int N,
    Ipp32f* pRes);
IppStatus ippsNthMaxElement_64f(const Ipp64f* pSrc, int len, int N,
    Ipp64f* pRes);
```

Parameters

pSrc	Pointer to the input vector [*len*].
len	Number of elements in the input vector *pSrc*.
N	Rank of element to find.
pRes	Pointer to the value of *N*-th maximal element.

Description

The function `ippsNthMaxElement` is declared in the `ippsr.h` file. This function finds the *N*-th maximal element of the input vector *pSrc*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pRes* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsBadArgErr`	Indicates an error when *N* is less then 0, or greater than or equal to *len*.

VecMatMul

Multiplies a vector by a matrix.

Syntax

```
IppStatus ippsVecMatMul_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pMatr, int
    step, int height, Ipp16s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_16s_D2LSfs(const Ipp16s* pSrc, const Ipp16s** mMatr,
    int height, Ipp16s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMatr, int
    step, int height, Ipp32f* pDst, int width);
IppStatus ippsVecMatMul_32f_D2L(const Ipp32f* pSrc, const Ipp32f** mMatr, int
    height, Ipp32f* pDst, int width);
```

```
IppStatus ippsVecMatMul_16s32s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pMatr,
    int step, int height, Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_16s32s_D2LSfs(const Ipp16s* pSrc, const Ipp16s* pMatr,
    int step, int height, Ipp32s* pDst,int width, int scaleFactor);
IppStatus ippsVecMatMul_32s_D2Sfs(const Ipp32s* pSrc, const Ipp32s* pMatr, int
    step, int height, Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_32s_D2LSfs(const Ipp32s* pSrc, const Ipp32s* pMatr, int
    step, int height, Ipp32s* pDst, int width, int scaleFactor);
```

Parameters

pSrc	Pointer to the input vector [*height*].
pMatr	Pointer to the input matrix vector [*height***step*].
mMatr	Pointer to the input matrix [*height*][*width*].
step	Row step in *pMatr* vector (in *pMatr* elements).
width	Length of the result vector and input matrix rows.
height	Number of rows in the input matrix.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsVecMatMul` is declared in the `ippsr.h` file. This function multiplies the input row vector by the given matrix as:

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot pMatr[i \cdot step + j]$$

for the function flavors with `D2` suffix,

and

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot mMatr[i][j]$$

for the function flavors with `D2L` suffix,

where $0 \le j < width$.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc*, *pMatr*, *mMatr* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

MatVecMul

Multiplies a matrix by a vector.

Syntax

```
IppStatus ippsMatVecMul_16s_D2Sfs(const Ipp16s* pMatr, int step, const Ipp16s*
    pSrc, int width, Ipp16s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_16s_D2LSfs(const Ipp16s** mMatr, const Ipp16s* pSrc,
    int width, Ipp16s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_32f_D2(const Ipp32f* pMatr, int step, const Ipp32f*
    pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMatVecMul_32f_D2L(const Ipp32f** mMatr, const Ipp32f* pSrc, int
    width, Ipp32f* pDst, int height);
IppStatus ippsMatVecMul_16s32s_D2Sfs(const Ipp16s* pMatr, int step, const
    Ipp16s* pSrc, int width, Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_16s32s_D2LSfs(const Ipp16s** mMatr, const Ipp16s* pSrc,
    int width, Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_32s_D2Sfs(const Ipp32s* pMatr, int step, const Ipp32s*
    pSrc, int width, Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_32s_D2LSfs(const Ipp32s** mMatr, const Ipp32s* pSrc,
    int width, Ipp32s* pDst, int height, int scaleFactor);
```

Parameters

pSrc	Pointer to the input vector [*width*].
pMatr	Pointer to the input matrix vector [*height***step*].
mMatr	Pointer to the input matrix [*height*][*width*].
step	Row step in *pMatr* vector (in *pMatr* elements).
width	Length of the input vector and input matrix rows.

height	Number of rows in the input matrix and the length of the result vector.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsVecMatMul` is declared in the `ippsr.h` file. This function multiplies the the given matrix by the input column vector as:

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot pMatr[i \cdot step + j]$$

for the function flavors with `D2` suffix,

and

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot mMatr[i][j]$$

for the function flavors with `D2L` suffix,

where $0 \le i <$ *height* .

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pMatr*, *mMatr* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

Feature Processing

This section describes functions that pre-process raw speech signals. Feature extraction is the first step in the recognition process. Speech features are a compressed form of the original speech signals. By extracting features, the problem dimension is reduced. To some extent, feature extraction normalizes speaker variations and environmental distortions. This section also includes functions needed to support both silence/speech detection and also some well-known analysis techniques on speech signals.

This functions are used in the Intel IPP Speech Recognition Samples. See *Aurora Encoder-Decoder Sample* downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

ZeroMean

Subtracts the mean value from the input vector.

Syntax

```
IppStatus ippsZeroMean_16s(Ipp16s* pSrcDst, int len);
```

Parameters

pSrcDst	Pointer to the source and destination vector [*len*].
len	The number of elements in the vector.

Description

The function `ippsZeroMean` is declared in the `ippsr.h` file. This function calculates the mean value of the *pSrcDst* vector and subtracts it from the vector *pSrcDst*. The resulting values are saturated if they exceed the range [-32768..32767] .
The operations are as follows:

$$pSrcDst[i] = \max(-32768, \min(32767, pSrcDst[i] - \frac{1}{len}\sum_{j=0}^{len-1} pSrcDst[j]))$$

for $0 \le i < len$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

CompensateOffset

Removes the DC offset of the input signals.

Syntax

```
IppStatus ippsCompensateOffset_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s* pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f* pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffset_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
    pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f_I(Ipp32f* pSrcDst, int len, Ipp32f*
    pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffsetQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
IppStatus ippsCompensateOffsetQ15_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
    pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
```

Parameters

pSrc	Pointer to the source vector [*len*].
pDst	Pointer to the destination vector [*len*].
pSrcDst	Pointer to the source and destination vector [*len*] for in-place operations.
pSrcDst0	Pointer to the previous source element.
len	Number of elements in the vector.
dst0	Previous destination element.
val	Constant for offset compensation.
valQ15	Real-valued Q15 format constant for offset compensation ($0.0_{Q31} \le valQ15 < 1.0_{Q31}$).
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsCompensateOffset` is declared in the `ippsr.h` file. This function removes the offset of the input signals. The destination vector is calculated as follows.

For the `ippsCompensateOffset` function:

$pDst[0] = pSrc[0] - pSrc[0] + val \cdot dst0$

$pDst[i] = pSrc[i] - pSrc[i-1] + val \cdot pDst[i-1]$, $1 \le i \le len-1$

$pSrcDst0[0] = pSrc[len-1]$,

and for `ippsCompensateOffset_I` function:

$y = pSrcDst[0] - pSrcDst0[0] + val \cdot dst0$, $x = pSrcDst[0]$, $pSrcDst[0] = y$,

$y = pSrcDst[i] - pSrcDst[i-1] + val \cdot x$, $x = pSrcDst[i]$, $pSrcDst[i] = y$,

$1 \le i \le len-1$,

$pSrcDst0[0] = x$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pDst*, *pSrcDst*, or *pSrcDst0* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0, or when *valQ15* is out of range.

SignChangeRate

Counts the zero-cross rate for the input signal.

Syntax

```
IppStatus ippsSignChangeRate_16s(const Ipp16s* pSrc, int len, Ipp32s* pRes);
IppStatus ippsSignChangeRate_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
IppStatus ippsSignChangeRate_Count0_16s(const Ipp16s* pSrc, int len, Ipp32s* pRes);
IppStatus ippsSignChangeRate_Count0_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
IppStatus ippsSignChangeRateXor_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
```

Parameters

pSrc	Pointer to the input signal [*len*].
len	Number of elements in the input signal *pSrc*.
pRes	Pointer to the result variable.

Description

The function `ippsSignChangeRate` is declared in the `ippsr.h` file. This function counts the number of sign changes in the input signal and can be used to detect speech in continuous speech input.
The operations are as follows:

For the `ippsSignChangeRate` function,

$$pRes[0] = \sum_{i=1}^{len-1} \begin{cases} 1, & if \;\; pSrc[i] \cdot pSrc[i-1] < 0 \\ 0, & otherwise \end{cases}$$

For the `ippsSignChangeRateCount0` function,

$$pRes[0] = \frac{1}{2} \sum_{i=1}^{len-1} \left| sign(pSrc[i]) - sign(pSrc[i-1]) \right| \;, \quad sign(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

For the `ippsSignChangeRateXor` function,

$$pRes[0] = \sum_{i=0}^{len-2} sign(pSrc[i]) \oplus sign(pSrc[i+1]) \text{, where } sign(x) = \begin{cases} 0, & x > 0, x = +0 \\ 1, & x < 0, x = -0 \end{cases}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pRes* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

LinearPrediction

Performs linear prediction analysis on the input vector.

Syntax

```
IppStatus ippsLinearPrediction_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDst, int lenDst, int scaleFactor);
```

```
IppStatus ippsLinearPrediction_Auto_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
    pDst, int lenDst);
IppStatus ippsLinearPredictionNeg_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPredictionNeg_Auto_32f(const Ipp32f* pSrc, int lenSrc,
    Ipp32f* pDst, int lenDst);
IppStatus ippsLinearPrediction_Cov_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPrediction_Cov_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
    pDst, int lenDst);
```

Parameters

pSrc	Pointer to the input vector [*lenSrc*].
lenSrc	Length of the input vector *pSrc*.
pDst	Pointer to the output LPC coefficients vector [*lenDst*].
lenDst	Length of the output vector *pDst*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsLinearPrediction` is declared in the `ippsr.h` file. This function performs linear prediction analysis on the input signal.
For functions with the `Auto` suffix, the LPC coefficients are calculated by solving the following equations that represent the autocorrelation approach:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = r[k] \ , k = 1,\ldots lenDst \ ,$$

where $r[k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k]$.

For functions with the `Neg_Auto` suffix, the LPC coefficients are calculated by solving the similar equations with inverted sign of the right-hand side:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = -r[k] \ , k = 1 \ldots lenDst \ ,$$

For functions with the Cov suffix, the LPC coefficients are calculated by solving the following equations that represent the covariance approach:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot c[i][k] = c[0][k] \ , k = 1, \ldots lenDst \ ,$$

where $c[i][k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k-i]$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *lenSrc* or *lenDst* is less than or equal to 0, or *lenDst* is greater or equal than *lenSrc*.
ippStsNoOperation	Indicates no solution to the LPC problem.

Durbin

Performs Durbin's recursion on an input vector of autocorrelations.

Syntax

```
IppStatus ippsDurbin_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp32f*
    pErr, int scaleFactor);
IppStatus ippsDurbin_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f* pErr);
```

Parameters

pSrc	Pointer to the input vector [*len+1*].
pDst	Pointer to the output LPC coefficients vector [*len*].

len	Length of the output vector.
pErr	Pointer to the result prediction error.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsDurbin is declared in the ippsr.h file. This function performs the Durbin's recursion on the input autocorrelation vector and calculates the linear prediction coefficients and the prediction error as follows:

1) Initialization:

$$m = len,\ R_j = pSrc[j],\ j = 0,...m,\ E^0 = R_0$$

2) Iterations for $i = 1,...m$:

$$k_i = \left(R_i - \sum_{j=1}^{i-1} y_j^{i-1} \cdot R_{i-j} \right) / E^{i-1},\ E^i = (1 - k_i^2) \cdot E^{i-1},$$

$$y_i^i = k_i,\quad y_j^i = y_j^{i-1} - k_i \cdot y_{i-j}^{i-1},\ j = 1,...\ i - 1$$

3) Result:

$$pDst[i-1] = y_i^m,\ i = 1,...m,\quad pErr[0] = E^m.$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pErr* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsNoOperation	Indicates no solution to the LPC problem ($E^i \le 0$).

Schur

Calculates reflection coefficients using Schur algorithm.

Syntax

```
IppStatus ippsSchur_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp32f*
    pErr, int scaleFactor);
IppStatus ippsSchur_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f*
    pErr);
```

Parameters

pSrc	Pointer to the input autocorrelations vector [*len+1*].
pDst	Pointer to the output reflection coefficients vector [*len*].
len	Length of the output vector.
pErr	Pointer to the result prediction error.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsSchur` is declared in the `ippsr.h` file. This function uses Schur algorithm to compute reflection coefficients in the following steps:

1) Initialization:

$$b_1^0 = pSrc[0],\quad a_j^0 = b_{j+1}^0 = pSrc[j],\quad \text{for } j = 1,\ldots len\text{-}1,\quad a_{len}^0 = pSrc[len]$$

2) Iterations for $i = 1,\ldots len$:

$$k_i = -a_i^{i-1} / b_i^{i-1}$$

$$a_j^i = a_j^{i-1} + k_i b_j^{i-1},\quad b_j^i = b_{j-1}^{i-1} + k_i a_{j-1}^{i-1},\quad j = i+1,\ldots len$$

3) Result:

$$pDst[i-1] = k_i,\quad i = 1,\ldots len$$

$$pErr[0] = b_{len}^{len-1} + k_{len} \cdot a_{len}^{len-1}.$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pErr* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsNoOperation	Indicates no solution to the LPC problem ($b_i^i = 0$).

LPToSpectrum

Calculates smoothed magnitude spectrum.

Syntax

```
IppStatus ippsLPToSpectrum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pDst,
    int order, Ipp32s val, int scaleFactor);
IppStatus ippsLPToSpectrum_32f(const Ipp32f* pSrc, int len, Ipp32f* pDst,
    int order, Ipp32f val);
```

Parameters

pSrc	Pointer to the input LPC coefficients vector [*len*].
pDst	Pointer to the output LP spectrum coefficients vector [2^{order}].
len	Number of LPC coefficients.
order	FFT order for spectrum calculation.
val	The value to add to spectrum.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsLPToSpectrum is declared in the ippsr.h file. This function computes the first half of a linear prediction magnitude spectrum as follows:

$$pDst[k] = \frac{1}{\left| val - \sum_{i=1}^{len} pSrc[i-1] \cdot e^{-\frac{jik\pi}{N}} \right|}, \quad k = 0,\ldots N-1, \quad N = 2^{order}$$

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or when it is greater or equal to $2^{order+1}$.
ippStsFftOrderErr	Indicates an error when the *order* value is incorrect.
ippStsDivByZero	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to +Inf and for integer operations is set to IPP_MAX_16S.

LPToCepstrum

Calculates cepstrum coefficients from linear prediction coefficients.

Syntax

```
IppStatus ippsLPToCepstrum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsLPToCepstrum_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the linear prediction coefficients [*len*].
pDst	Pointer to the cepstrum coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsLPToCepstrum is declared in the ippsr.h file. This function calculates the cepstrum coefficients from the linear prediction coefficients according to the following formula:

$$pDst[k] = -\left(pSrc[k] + \frac{1}{k+1} \sum_{i=1}^{k} (k-i+1) \cdot pSrc[i-1] \cdot pDst[k-i] \right), \; k = 0 \ldots len-1 .$$

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

CepstrumToLP

Calculates linear prediction coefficients from cepstrum coefficients.

Syntax

```
IppStatus ippsCepstrumToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsCepstrumToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the cepstrum coefficients [*len*].
pDst	Pointer to the linear prediction coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsCepstrumToLP is declared in the ippsr.h file. This function calculates the linear prediction coefficients from the cepstrum coefficients according to the following formula:

$$pDst[k] = -\left(pSrc[k] + \frac{1}{k+1} \sum_{i=1}^{k} (k-i+1) \cdot pSrc[k-i] \cdot pDst[i-1] \right), \; k = 0 \ldots len-1 .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

LPToReflection

Calculates the linear prediction reflection coefficients from the linear prediction coefficients.

Syntax

```
IppStatus ippsLPToReflection_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsLPToReflection_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the linear prediction coefficients [*len*].
pDst	Pointer to the linear prediction reflection coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsLPToReflection` is declared in the `ippsr.h` file. This function calculates the linear prediction reflection coefficients according to the following formulae:

1) Initialization:

$$n = len, \quad a_i^n = pSrc[i-1], \quad i = 1,\ldots n$$

2) Iterations for $i = m,\ldots 1$:

$$k_i = a_i^i, \quad a_j^{i-1} = \frac{a_j^i - a_i^i \cdot a_{i-j}^i}{1 - k_i^2}, \quad j = 1,\ldots i\text{-}1$$

3) Result:

$$pDst[i-1] = k_i, \quad i = 1,\ldots n.$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsNoOperation`	Indicates that reflection coefficients could not be calculated (that is, $\lvert k_i \rvert = 1$ for one of iterations).

ReflectionToLP

Calculates the linear prediction coefficients from the linear prediction reflection coefficients.

Syntax

```
IppStatus ippsReflectionToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsReflectionToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the linear prediction reflection coefficients [*len*].
pDst	Pointer to the linear prediction coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsReflectionToLP` is declared in the `ippsr.h` file. This function calculates the linear prediction reflection coefficients from the linear prediction coefficients according to the following formulae:

1) Initialization:

$$n = len, \quad k_i = pSrc[i-1], \quad i = 1,...n$$

2) Iterations for $i = 1,...m$:

$$a_i^i = k_i, \quad a_j^i = a_j^{i-1} - k_i \cdot a_{i-j}^{i-1}, \quad j = 1,...k\text{-}1$$

3) Result:

$$pDst[i-1] = a_i^n, \quad i = 1,...n.$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

ReflectionToAR

Converts reflection coefficients to area ratios.

Syntax

```
IppStatus ippsReflectionToAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsReflectionToLAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
    Ipp16s* pDst, int len, Ipp32f val, int scaleFactor);
IppStatus ippsReflectionToLAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f val);
IppStatus ippsReflectionToTrueAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToTrueAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.
val	Threshold value (1 > *val* > 0).
srcShiftVal	Scale factor used in `ippsReflectionToLAR` function only. Refer to "Integer Scaling" in Chapter 2.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

These functions are declared in the `ippsr.h` file.

The function `ippsReflectionToAR` calculates approximate area ratios using the following formula:

$$pDst[k] = \frac{1 - pSrc[k]}{1 + pSrc[k]}, \quad k = 0,\ldots len-1$$

The function `ippsReflectionToLAR` calculates logarithm of area ratios as given by:

$$pDst[k] = \begin{cases} \ln\frac{1-val}{1+val}, & if\ pSrc[k] \geq val \\ \ln\frac{1-pSrc[k]}{1+pSrc[k]}, & if\ val > pSrc[k] > -val,\ k = 0, \ldots len-1 \\ \ln\frac{1+val}{1-val}, & if\ -val \geq pSrc[k] \end{cases}$$

The function `ippsReflectionToTrueAR` calculates area ratios using the formulae:

$$pDst[0] = \frac{1-pSrc[0]}{1+pSrc[0]},\ pDst[k] = pDst[k-1] \cdot \frac{1-pSrc[k]}{1+pSrc[k]},\ k = 1, \ldots len-1.$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDivByZero`	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to: `NaN`, if the dividend vector element has zero value; `+Inf`, if the dividend vector element has positive value; `-Inf`, if the dividend vector element is negative. The value of the destination vector element for integer operations is set to: `IPP_MAX_16S`, if the dividend vector element is positive; `IPP_MIN_16S`, if the dividend vector element is negative.
`ippStsRangeErr`	Indicates an error when the condition $1 > val > 0$ is not met.

ReflectionToTilt

Calculates tilt for rise/fall/connection parameters.

Syntax

```
IppStatus ippsReflectionToAbsTilt_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToAbsTilt_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
```

```
IppStatus ippsReflectionToTilt_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToTilt_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
```

Parameters

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

These functions are declared in the `ippsr.h` file.

The function `ippsReflectionToAbsTilt` converts rise and fall coefficients to absolute tilt as given by:

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{|pSrc1[k]| + |pSrc2[k]|}, \quad k = 0,\ldots len-1 .$$

The function `ippsReflectionToTilt` converts rise and fall coefficients to tilt:

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{pSrc1[k] + pSrc2[k]}, \quad k = 0,\ldots len-1 .$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDivByZero`	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to: `NaN`, if the dividend vector element has zero value; `+Inf`, if the dividend vector element has positive value; `-Inf`, if the dividend vector element is negative. The value of the destination vector element for integer operations is set to: `IPP_MAX_16S`, if the dividend vector element is positive; `IPP_MIN_16S`, if the dividend vector element is negative.

PitchmarkToF0

Calculates rise and fall amplitude and duration for tilt.

Syntax

```
IppStatus ippsPitchmarkToF0Cand_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, int scaleFactor);
IppStatus ippsPitchmarkToF0Cand_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsPitchmarkToF0Cand` is declared in the `ippsr.h` file. This function calculates `F0` candidate values from pitchmarks using the following formula:

$$pDst[0] = \frac{1}{pSrc[0]}, \quad pDst[k] = \frac{1}{pSrc[k] - pSrc[k-1]}, \quad k = 1,\ldots len-1 .$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDivByZero`	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to `+Inf`. The value of the destination vector element for integer operations is set to `IPP_MAX_16S`.

UnitCurve

Calculates tilt for rise and fall coefficicents.

Syntax

```
IppStatus ippsUnitCurve_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsUnitCurve_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsUnitCurve_16s_ISfs(Ipp16s* pSrcDst, int srcShiftVal, int len, int
    scaleFactor);
IppStatus ippsUnitCurve_32f_I(Ipp32f* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pSrcDst	Pointer to the input and destination vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.
srcShiftVal	Refer to "Integer Scaling" in Chapter 2.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsUnitCurve` is declared in the `ippsr.h` file. This function calculates the destination vector according to the following formula:

$$pDst[k] = \begin{cases} 0, & if\ pSrc[k] < 0 \\ pSrc[k]^2, & if\ 0 \le pSrc[k] < 1 \\ 2-(2-pSrc[k])^2, & if\ 1 \le pSrc[k] \le 2 \\ 2, & if\ 2 < pSrc[k] \end{cases}, \text{ for } k = 0,\ldots len-1.$$

In-place function flavors ovewrite source vector with destination data.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pSrcDst* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

LPToLSP

Calculates line spectrum pairs vector from linear prediction coefficients.

Syntax

```
IppStatus ippsLPToLSP_16s_Sfs(const Ipp16s* pSrcLP, int srcShiftVal, Ipp16s*
    pDstLSP, int len, int* nRoots, int nInt, int nDiv, int scaleFactor);
IppStatus ippsLPToLSP_32f(const Ipp32f* pSrcLP, Ipp32f* pDstLSP, int len, int*
    nRoots, int nInt, int nDiv);
```

Parameters

pSrcLP	Pointer to the input LP coefficients vector [*len*].
pDstLSP	Pointer to the output LSP coefficients vector [*len*].
len	Number of LP coefficients.
nRoots	Pointer to the number of LSP values found.
nInt	Number of intervals in zero crossing search.
nDiv	Number of interval bisections during search for roots.
srcShiftVal	Scale factor for *pSrcLP* values.
scaleFactor	Scale factor for *pDstLSP* values, refer to "Integer Scaling" in Chapter 2.

Description

The function ippsLPToLSP is declared in the ippsr.h file. This function converts linear prediction (LP) coefficients to the line spectrum pair (LSP) representation and implements the algorithm generally used in speech coding.

Let $a_k = pSrcLP[k]$, $k = 0,\ldots len-1$ (or $a_k = pSrcLP[k] \cdot 2^{-srcShiftVal}$ in case of the ippsLPToLSP_16s_Sfs function) be the LP coefficients (see also the formula used in the ippsLinearPredictionNeg_Auto function to compute LP coefficients), and

$$A(z) = 1 - \sum_{k=0}^{len-1} a_k z^{-k} .$$

LSP coefficients are defined as the roots of symmetric and anti-symmetric polynomials of the form

$$F_1(z) = A(z) + z^{-len-1} \cdot A(z^{-1}), \quad F_2(z) = A(z) - z^{-len-1} \cdot A(z^{-1}),$$

with the exception of the roots equal to 1 and -1, which are excluded through the definition of the following polynomials $G_1(z)$ and $G_2(z)$:

$G_1(z) = F_1(z)/(1+z^{-1})$, $G_2(z) = F_2(z)/(1-z^{-1})$, if len is even;

$G_1(z) = F_1(z)$, $G_2(z) = F_2(z)/(1-z^{-2})$, if len is odd.

These polynomials are symmetric, so that

$$G_i(z) = g_{i,m_i} \cdot z^{-m_i} + \sum_{k=0}^{m_i-1} g_{i,k} \cdot \left(z^{-k} + z^{-(2m_i-k)}\right),$$

and they can be represented in the form

$$G_i(e^{jw}) = e^{-jwm_i} \cdot G_i'(w) = g_{i,m_i} + 2 \cdot \sum_{k=0}^{m_i-1} g_{i,k} \cdot T_{m_i-k}(x), \text{ for } i = 1,2,$$

where $x = \cos w$, and $T_k(x) = \cos kw$ are the m-th order Chebyshev polynomials (see [Kab86]).

When len is even, the polynomial degrees are given by $m_1 = m_2 = len/2$, and

$g_{1,0} = g_{2,0} = 1$, $g_{1,k} = a_{k-1} + a_{len-k} - g_{1,k-1}$, $g_{2,k} = a_{k-1} - a_{len-k} + g_{2,k-1}$, for $k = 1,..m_1$.

When len is odd, $m_1 = (len+1)/2$, $m_2 = (len-1)/2$ and

$g_{1,0} = 1$, $g_{1,k} = -a_{k-1} - a_{len-k}$ for $k = 1,..m_1$,

$g_{2,-1} = 0$, $g_{2,0} = 1$, and if $m_2 > 0$ then $g_{2,k} = g_{2,k-2} + a_{len-k} - a_{k-1}$ for $k = 1,..m_2$.

Polynomials $G_1(z)$ and $G_2(z)$ have distinct interlaced roots on the unit circle that are found in cosine domain by checking for polynomials sign change at $nInt$ points equally spaced between 0 and π (zero crossing search). The sign change interval is then bisected $nDiv$ times and after that the root is located by linear interpolation. The Chebyshev polynomials are used to evaluate $G_1(z)$ and $G_2(z)$. The roots are written to $pDstLSP$ vector in ascending order.

The number of roots found is written to *nRoots*[0]. If all *len* roots were not found, the error status is returned.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcLP*, *pDstLSP* or *nRoots* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or *nInt* is less than *len*, or *nDiv* is less than 0.
ippStsNoRootFoundErr	Indicates that less than *len* roots were found.

LSPToLP

Converts line spectrum pairs vector to linear prediction coefficients.

Syntax

```
IppStatus ippsLSPToLP_16s_Sfs(const Ipp16s* pSrcLSP, int srcShiftVal, Ipp16s*
    pDstLP, int len, int scaleFactor);
IppStatus ippsLSPToLP_32f(const Ipp32f* pSrcLSP, Ipp32f* pDstLP, int len);
```

Parameters

pSrcLSP	Pointer to the input LSP coefficients vector [*len*].
pDstLP	Pointer to the output LP coefficients vector [*len*].
len	Number of LSP and LP coefficients.
srcShiftVal	Scale factor for *pSrcLSP* values.
scaleFactor	Scale factor for *pDstLP* values, refer to "Integer Scaling" in Chapter 2.

Description

The function ippsLSPToLP is declared in the ippsr.h file. This function converts line spectrum pair (LSP) values to linear prediction (LP) coefficients using the Kabal's method (see [Kab86]).

The polynomials

$$G_1(e^{-jw}) = e^{-jwm_1} \cdot G_1'(w) \quad \text{and} \quad G_2(e^{-jw}) = e^{-jwm_2} \cdot G_2'(w)$$

are reconstructed from their roots *pSrcLSP*[k], $k = 0,\ldots len-1$, using Chebyshev representation.

Next the polynomials $F_1(z)$ and $F_2(z)$ and are reconstructed as:

$F_1(z) = G_1(z) \cdot (1 + z^{-1})$, $F_2(z) = G_2(z) \cdot (1 - z^{-1})$, if *len* is even;

$F_1(z) = G_1(z)$, $F_2(z) = G_2(z) \cdot (1 - z^{-2})$, if *len* is odd.

Finally, the linear prediction coefficients are determined from

$$A(z) = \frac{F_1(z) + F_2(z)}{2} = 1 - \sum_{k=0}^{len-1} a_k z^{-k}$$

and

$pDstLP[k] = a_k$, $k = 0,\ldots len-1$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcLSP* or *pDstLP* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsIncorrectLSP	Indicates a warning that *pSrcLSP* elements are not valid LSP values (so that condition $-1 < \ldots < pSrcLSP[k+1] < pSrcLSP[k] < \ldots < 1$ is not met).

MelToLinear

Converts Mel-scaled values to linear scale values.

Syntax

```
IppStatus ippsMelToLinear_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    melMul, Ipp32f melDiv);
```

Parameters

pSrc	Pointer to the input vector [*len*].

pDst	Pointer to the output vector [*len*].
len	Length of input and output vectors.
melMul	Multiply factor in the Mel-scale equation.
melDiv	Divide factor in the Mel-scale equation.

Description

The function ippsMelToLinear is declared in the ippsr.h file. This function converts the Mel-frequency scale to the linear frequency scale:

$$pDst[i] = melDiv \cdot \left[\exp\left(\frac{pSrc[i]}{melMul}\right) - 1\right], \quad i = 0,\ldots,len-1 .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or *melMul* or *melDiv* is equal to 0.

LinearToMel

Converts linear-scale values to Mel-scale values.

Syntax

```
IppStatus ippsLinearToMel_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    melMul, Ipp32f melDiv);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the input and output vectors.
melMul	Multiply factor in the Mel-scale equation.
melDiv	Divide factor in the Mel-scale equation.

Description

The function ippsLinearToMel is declared in the ippsr.h file. This function converts the linear frequency scale to the Mel-frequency scale:

$$pDst[i] = melMul \cdot \ln\left(1 + \frac{pSrc[i]}{melDiv}\right), \quad i = 0,\ldots,len-1 .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or *melMul* or *melDiv* is equal to 0.

CopyWithPadding

Copies the input signal to the output with zero-padding.

Syntax

```
IppStatus ippsCopyWithPadding_16s(const Ipp16s* pSrc, int lenSrc, Ipp16s* pDst,
    int lenDst);
IppStatus ippsCopyWithPadding_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f* pDst,
    int lenDst);
```

Parameters

pSrc	Pointer to the input vector [*lenSrc*].
pDst	Pointer to the output vector [*lenDst*].
lenSrc	Length of the input vector *pSrc*.
lenDst	Length of the output vector *pDst*.

Description

The function ippsCopyWithPadding is declared in the ippsr.h file. This function copies the input vector to the output vector. If the length of the output vector is bigger, the trailing elements are padded with zeroes.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *lenSrc* or *lenDst* is less than or equal to 0, or when *lenDst* is less than *lenSrc*.

MelFBankGetSize

Gets the size of the Mel-frequency filter bank structure.

Syntax

```
IppStatus ippsMelFBankGetSize_32s(int winSize, int nFilter, IppMelMode mode,
    int* pSize);
```

Parameters

winSize	Frame length in samples (32 ≤*winSize* ≤8192).
nFilter	Number of Mel-scale filter banks K (0 < *nFilter* ≤*winSize*).
mode	Flag that determines the execution mode. Currently only `IPP_FBANK_FREQWGT` is supported.
pSize	Pointer to the variable to contain the size of the filter bank structure.

Description

The function `ippsMelFBankGetSize` is declared in the `ippsr.h` file. This function determines the size required for the Mel-frequency filter bank structure and associated storage. It should be called before memory allocation and before the call to `ippsMelFBankInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsFBankFlagErr`	Indicates an error when the *mode* value is incorrect.
`ippStsNullPtrErr`	Indicates an error when *pSize* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *nFilter* is out of range or when *winSize* is less than or equal to 0.

MelFBankInit

Initializes the structure for performing the Mel-frequency filter bank analysis.

Syntax

```
IppStatus ippsMelFBankInit_32s(IppsFBankState_32s* pFBank, int* pFFTLen, int
    winSize, Ipp32s sampFreq, Ipp32s lowFreq, Ipp32s highFreq, int nFilter,
    Ipp32s melMulQ15, Ipp32s melDivQ15, IppMelMode mode);
```

Parameters

pFBank	Pointer to the Mel-scale filter bank structure to be initialixed.
pFFTLen	Pointer to the length of FFT (N = *pFFTLen* [0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
sampFreq	Input signal sampling frequency f_i (in Hz).
lowFreq	Start frequency f_{low} of the first band-pass filter (in Hz).
highFreq	End frequency f_{high} of the last band-pass filter (in Hz).
nFilter	Number of Mel-scale filter banks K.
melMulQ15	Real-valued mel-scale formula multiply factor in Q15.
melDivQ15	Real-valued mel-scale formula divisor in Q15.
mode	Flags that determine execution mode; can have the following values: IPP_FBANK_MELWGT – the function calculates filter bank weights in Mel-scale; IPP_FBANK_FREQWGT – the function calculates filter bank weights in the frequency space. One of the above two flags should necessarily be set. IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis.

Description

The function `ippsMelFBankInit` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis function. Mel filter coefficients are calculated in the same way as for the `ippsMelFBankInitAlloc` functions. Memory for the structure should be previously allocated, the size of this memory can be determined by the `ippsMelFBankGetSize` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* or *pFFTLen* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *winSize, nFilter, sampFreq*, or *lowFreq* is less than or equal to 0.
`ippStsFBankFreqErr`	Indicates an error when *highFreq* is less than *lowFreq* or *highFreq* is greater than *sampFreq/2*.
`ippStsFBankFlagErr`	Indicates an error when the *mode* value is incorrect.

MelFBankInitAlloc

Allocates memory and initializes the structure for performing the Mel-frequency filter bank analysis.

Syntax

```
IppStatus ippsMelFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f highFreq,
    int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
IppStatus ippsMelFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f highFreq,
    int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

Parameters

pFBank	Pointer to the Mel-scale filter bank structure to be created.
pFFTLen	Pointer to the length of FFT (N = *pFFTLen*[0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
sampFreq	Input signal sampling frequency f_i (in Hz).

lowFreq	Start frequency f_{low} of the first band-pass filter (in Hz).
highFreq	End frequency f_{high} of the last band-pass filter (in Hz).
nFilter	Number of Mel-scale filter banks K.
melMul	Mel-scale formula multiply factor.
melDiv	Mel-scale formula divisor.
mode	Flags that determine execution mode; can have the following values: IPP_FBANK_MELWGT – the function calculates filter bank weights in Mel-scale; IPP_FBANK_FREQWGT – the function calculates filter bank weights in the frequency space. One of the above two flags should necessarily be set. IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis.

Description

The function ippsMelFBankInitAlloc is declared in the ippsr.h file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis. The filter bank analysis is one of the major steps in the Mel-Frequency Cepstrum Coefficients (MFCC) feature calculation.

The Mel-frequency translation is accomplished according to the following equation:

$$mel(f) = melMul \cdot \ln\left(1 + \frac{f}{melDiv}\right)$$

The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

$$c_k = mel(f_{low}) + \frac{k}{K+1} \cdot [mel(f_{high}) - mel(f_{low})] ,$$

$$y_k = round\left\{\frac{N}{f_s} mel^{-1}(c_k)\right\},$$

for $k = 0 \ldots K+1$,

where $round(x)$ rounds the value of x to the nearest integer (when *mode* is equal to IPP_FBANK_FREQWGT) or to the largest integer that is less than or equal to x (when *mode* is equal to IPP_FBANK_MELWGT), and N is the length of FFT.

If *mode* is IPP_FBANK_FREQWGT, then the filter outputs will be calculated (by the function ippsEvalFBank) according to the following formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}}^{y_k} \frac{i - y_{k-1} + 1}{y_k - y_{k-1} + 1} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{y_{k+1} - i + 1}{y_{k+1} - y_k + 1} \cdot x_i \quad , \text{ for } k = 1 \ldots K , \qquad (8.1)$$

where x_i is the magnitude of the corresponding FFT spectrums.

If *mode* is IPP_FBANK_MELWGT, the filter outputs will be calculated (by the function ippsEvalFBank) according to the formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}+1}^{y_k} \frac{mel\left(i\frac{f_s}{N}\right) - c_{k-1}}{c_k - c_{k-1}} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{c_{k+1} - mel\left(i\frac{f_s}{N}\right)}{c_{k+1} - c_k} \cdot x_i , \; k = 1 \ldots K \qquad (8.2)$$

For the function flavor ippsMelFBankInitAlloc_32s, currently only IPP_FBANK_MELWGT mode is supported.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pFFTLen* pointer is NULL.
ippStsSizeErr	Indicates an error when *winSize*, *nFilter*, *sampFreq*, or *lowFreq* is less than or equal to 0.
ippStsFBankFreqErr	Indicates an error when *highFreq* is less than *lowFreq* or *highFreq* is greater than *sampFreq/2*.
ippStsFBankFlagErr	Indicates an error when the *mode* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

MelLinFBankInitAlloc

Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.

Syntax

```
IppStatus ippsMelLinFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
    Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
    Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

```
IppStatus ippsMelLinFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
    Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
    Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

Parameters

pFBank	Pointer to the filter bank structure to be created.
pFFTLen	Pointer to the length of FFT ($N = pFFTLen[0]$) used for the filter bank evaluation.
winSize	Frame length (in samples).
sampFreq	Input signal sampling frequency f_i (in Hz).
lowFreq	Start frequency f_{low} of the first band-pass filter (in Hz).
highLinFreq	End frequency f_{lin} of the last band-pass filter in linear scale (in Hz).
highFreq	End frequency f_{high} of the last band-pass filter (in Hz).
nFilter	Number of Mel-scale filter banks K.
nLinFilter	Number of linear-scale filter banks L.
melMul	Mel-scale formula multiply factor.
melDiv	Mel-scale formula divisor.
mode	Flags determining the function's execution mode; can have the following values:

`IPP_FBANK_MELWGT` – the function calculates filter bank weights in Mel-scale;

`IPP_FBANK_FREQWGT` – the function calculates filter bank weights in the frequency space.

One of the above two flags should necessarily be set.

`IPP_POWER_SPECTRUM` – indicates that the FFT power spectrum is used during the filter bank analysis.

Description

The function `ippsMelLinFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the linear and Mel-frequency triangular filter banks. The first L filters are placed linearly on the frequency band from f_{low} to f_{lin}. The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

If $L = K$, then

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L+1} \cdot k, \quad y_k = round\left\{\frac{N}{f_s} \cdot z_k\right\}, \quad c_k = mel(z_k) \quad , k = 0,\ldots L+1$$

If $L < K$, then

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L} \cdot k \ , \quad y_k = round\left\{\frac{N}{f_s} \cdot z_k\right\} \ , \quad c_k = mel(z_k) \quad , k = 0,\ldots L$$

and

$$c_k = mel(f_{lin}) + \frac{k-L}{K-L+1} \cdot [mel(f_{high}) - mel(f_{lin})] \ ,$$

$$y_k = round\left\{\frac{N}{f_s} \cdot mel^{-1}(c_k)\right\}, \text{ for } k = L,\ldots K+1 \ ,$$

where *round*(*x*) rounds the value of *x* to the nearest integer (when *mode* is equal to IPP_FBANK_FREQWGT) or to the largest integer that is less than or equal to *x* (when *mode* is IPP_FBANK_MELWGT), and *N* is the length of FFT.

If *mode* is IPP_FBANK_FREQWGT, the filter outputs are calculated (by the function ippsEvalFBank) according to the formula (8.1) given for the ippsMelFBankInitAlloc function.

If *mode* is IPP_FBANK_MELWGT, the filter output is calculated (by the function ippsEvalFBank) according to the formula (8.2).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pFFTLen* pointer is NULL.
ippStsSizeErr	Indicates an error when *winSize*, *nFilter*, *sampFreq*, *lowFreq* is less than or equal to 0, or when *nLinFilter* is less than or equal to 1, or *nLinFilter* is greater than *nFilter*.
ippStsFBankFreqErr	Indicates an error when *highLinFreq* is less than *lowFreq*, or *highFreq* is less than *highLinFreq* , or *highFreq* is greater than *sampFreq/2*, or *(highLinFreq > lowFreq)&&(nLinFilter==0)*, or *(highLinFreq < highFreq)&&(nLinFilter==nFilter)* .
ippStsFBankFlagErr	Indicates an error when the *mode* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

EmptyFBankInitAlloc

Initializes an empty filter bank structure.

Syntax

```
IppStatus ippsEmptyFBankInitAlloc_16s(IppsFBankState_16s** pFBank, int*
    pFFTLen, int winSize, int nFilter, IppMelMode mode);
IppStatus ippsEmptyFBankInitAlloc_32f(IppsFBankState_32f** pFBank, int*
    pFFTLen, int winSize, int nFilter, IppMelMode mode);
```

Parameters

pFBank	Pointer to the filter bank structure to be created.
pFFTLen	Pointer to the length of FFT (N = *pFFTLen*[0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
nFilter	Number of filters banks K.
mode	Flag determining the function's execution mode; can have the following value: `IPP_POWER_SPECTRUM` – indicates that the FFT power spectrum is used during the filter bank analysis.

Description

The function `ippsEmptyFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the filter bank structure for the *nFilter* filters. The filter bank center frequencies can be set by the function `ippsFBankSetCenters` and the filter weights can be set by the function `ippsFBankSetCoeffs`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* or *pFFTLen* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *winSize* or *nFilter* or *pFBank* is less than or equal to 0.
`ippStsMemAllocErr`	Indicates an error when no memory was allocated.

FBankFree

Destroys the structure for the filter bank analysis.

Syntax

```
IppStatus ippsFBankFree_16s(IppsFBankState_16s* pFBank);
IppStatus ippsFBankFree_32f(IppsFBankState_32f* pFBank);
```

Parameters

pFBank	Pointer to the filter bank structure.

Description

The function `ippsFBankFree` is declared in the `ippsr.h` file. This function destroys the filter bank structure and frees all memory associated with it.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* pointer is `NULL`.

FBankGetCenters

Retrieves the center frequencies of the triangular filter banks.

Syntax

```
IppStatus ippsFBankGetCenters_16s(const IppsFBankState_16s* pFBank, int*
    pCenters);
IppStatus ippsFBankGetCenters_32f(const IppsFBankState_32f* pFBank, int*
    pCenters);
```

Parameters

pFBank	Pointer to the filter bank structure.
pCenters	Pointer to the output vector that contains the center frequencies.

Description

The function `ippsGetCenters` is declared in the `ippsr.h` file. This function retrieves the indexes y_k (in FFT domain points) of the filter bank centers. The resulting array *pCenters* is of length *nFilter*+2. The filter bank structure *pFBank* must be initialized by either `ippsMelFBankInitAlloc` or `ippsMelLinFBankInitAlloc` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* or *pCenters* pointer is `NULL`.
`ippStsFBankErr`	Indicates an error when filter centers are not valid after filter bank initialization by `ippsEmptyFBankInitAlloc` function.

FBankSetCenters

Sets the center frequencies of the triangular filter banks.

Syntax

```
IppStatus ippsFBankSetCenters_16s(IppsFBankState_16s* pFBank, const int*
    pCenters);
IppStatus ippsFBankSetCenters_32f(IppsFBankState_32f* pFBank, const int*
    pCenters);
```

Parameters

pFBank	Pointer to the filter bank structure.
pCenters	Pointer to the vector that contains center frequencies.

Description

The function `ippsSetCenters` is declared in the `ippsr.h` file. This function sets the filter center indexes y_k in *pFBank*.
The indexes must meet the following conditions:

$$0 \le \ldots \le y_k \le y_{k+1} \le \ldots \le N/2, \quad i = 0,\ldots K.$$

If the *k*-th center is modified, the filter weight coefficients in the (*k*-1)-th, *k*-th and (*k*+1)-th filter banks must be also modified using the `ippsFBankSetCoeffs` function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pCenters* pointer is NULL.

FBankGetCoeffs

Retrieves the filter bank weight coefficients.

Syntax

```
IppStatus ippsFBankGetCoeffs_16s(const IppsFBankState_16s* pFBank, int fIdx,
    Ipp32f* pCoeffs);
IppStatus ippsFBankGetCoeffs_32f(const IppsFBankState_32f* pFBank, int fIdx,
    Ipp32f* pCoeffs);
```

Parameters

pFBank	Pointer to the filter bank structure
fIdx	Filter index.
pCoeffs	Pointer to the output vector that contains the filter coefficients.

Description

The function ippsFBankGetCoeffs is declared in the ippsr.h file. This function copies the weight coefficients of the filter bank *fIdx* to the array *pCoeffs* which is of length $(y_{k+1} - y_{k-1} + 1)$.

Note that for ippsFBankGetCoeffs_16s function flavor, the filter output weights are represented in floating-point format.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pCoeffs* pointer is NULL.
ippStsSizeErr	Indicates an error when *fIdx* is less then 1 or greater then *nFilter*.
ippStsFBankErr	Indicates an error when *fIdx* filter coefficients are not available or valid.

FBankSetCoeffs

Sets the filter bank weight coefficients.

Syntax

```
IppStatus ippsFBankSetCoeffs_16s(IppsFBankState_16s* pFBank, int fIdx, const
    Ipp32f* pCoeffs);
IppStatus ippsFBankSetCoeffs_32f(IppsFBankState_32f* pFBank, int fIdx, const
    Ipp32f* pCoeffs);
```

Parameters

pFBank	Pointer to the filter bank structure.
fIdx	Filter index.
pCoeffs	Pointer to the output coefficients vector.

Description

The function `ippsFBankGetCoeffs` is declared in the `ippsr.h` file. This function sets the weight coefficients of the filter bank *fIdx*. The vector *pCoeffs* contains the weight coefficients from y_{k-1} to y_{k+1}.

Note that for `ippsFBankSetCoeffs_16s` function flavor, the weight coefficients are represented in floating-point format and are saturated to the [-1,1] interval.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* or *pCoeffs* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *fIdx* is less then 1 or greater then *nFilter*.
`ippStsFBankErr`	Indicates an error when the weight coefficients are not available or valid.

EvalFBank

Performs the filter bank analysis.

Syntax

```
IppStatus ippsEvalFBank_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst, const
    IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFBankState_32s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFBankState_32f* pFBank);
```

Parameters

pSrc	Pointer to the source vector $[2^{pFFTOrder[0]}]$.
pDst	Pointer to the filter bank coefficients vector [*nFilter*].
pFBank	Pointer to the filter bank structure.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsEvalFBank` is declared in the `ippsr.h` file. This function performs the filter bank analysis for the input vector *pSrc*.

The execution mode set during the filter bank structure initialization determines the use of the input vector as follows:

If the `IPP_POWER_SPECTRUM` flag is set, the source vector is the wave signals. The magnitude of input signal spectrum is calculated for the filter bank analysis as follows:

$$x_k = \left| \left(\sum_{i=0}^{N-1} pSrc[i] \cdot \exp\left(-j \cdot 2\pi \frac{ik}{N}\right) \right) \right| , \quad 0 \le k \le N/2.$$

Otherwise, the input vector is used directly for filter bank analysis:

$$x_k = pSrc[i] , \quad 0 \le k \le N/2.$$

Depending on the mode, either formula (8.1) or (8.2) is used to obtain the filter bank coefficients. The input vector *pSrc* is destroyed after the analysis.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsFBankErr	Indicates an error when *pFBank* structure is not ready for calculation.

DCTLifterGetSize_MulC0

Gets the size of the DCT structure.

Syntax

```
IppStatus ippsDCTLifterGetSize_MulC0_16s(int lenDCT, int lenCeps, int* pSize);
```

Parameters

lenDCT	Length of the DCT (0 < *lenDCT* ≤8192).
lenCeps	Number of the output coefficients not including C0 (0 < *lenCeps* ≤*lenDCT*).
pSize	Pointer to the variable to contain the size in bytes.

Description

The function ippsDCTLifterGetSize_MulC0 is declared in the ippsr.h file.
This function computes the size in bytes required for the DCT structure and associated storage.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSize* pointer is NULL.
ippStsSizeErr	Indicates an error if *lenDCT* or *lenCeps* are out of bounds.

DCTLifterInit_MulC0

Initializes the structure to perform DCT and lift the DCT coefficients.

Syntax

```
IppStatus ippsDCTLifterInit_MulC0_16s(IppsDCTLifterState_16s* pDCTLifter, int
    lenDCT, const Ipp32s* pLifterQ15, int lenCeps);
```

Parameters

pDCTLifter	Pointer to the structure to be initialized for the DCT calculation and liftering.
lenDCT	Length of the DCT (0 < *lenDCT* ≤ 8192).
pLifterQ15	Pointer to the Q15 format liftering coefficients vector ($0.0_{Q15} \le$ *pLifterQ15*[k] $< 512.0_{Q15}$).
lenCeps	Number of the output coefficients not including C_0 (0 < *lenCeps* ≤ *lenDCT*).

Description

The function `ippsDCTLifterInit_MulC0` is declared in the `ippsr.h` file. This function initializes the structure for the DCT calculation and liftering similar to what is done by the `ippsDCTLifterInitAlloc_MulC0` function. However, memory for the structure should be previously allocated, and the size of memory could be determined by the `ippsDCTLifterGetSize_MulC0` function.

C_0 is stored as the last element of the output vector.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pLifterQ15* or *pDCTLifter* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *lenDCT*, *lenCeps*, or *pLifterQ15*[k] is out of range.

DCTLifterInitAlloc

Initializes the structure and allocates memory to perform DCT and lift the DCT coefficients.

Syntax

```
IppStatus ippsDCTLifterInitAlloc_16s(IppsDCTLifterState_16s** pDCTLifter,
    int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_16s(IppsDCTLifterState_16s** pDCTLifter,
    int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_16s(IppsDCTLifterState_16s** pDCTLifter,
    int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_16s(IppsDCTLifterState_16s** pDCTLifter,
    int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_32f(IppsDCTLifterState_32f** pDCTLifter,
    int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_32f(IppsDCTLifterState_32f** pDCTLifter,
    int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_32f(IppsDCTLifterState_32f** pDCTLifter,
    int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_32f(IppsDCTLifterState_32f** pDCTLifter,
    int lenDCT, const Ipp32f* pLifter, int lenCeps);
```

Parameters

pDCTLifter	Pointer to the structure to be created for the DCT calculation and liftering.
lenDCT	Length of the DCT.
lenCeps	Number of the output coefficients (not including C_0).
nLifter	Liftering factor.
pLifter	Pointer to the liftering coefficients vector.
val	The scale factor for the output coefficients (except C_0).
val0	The scale factor for C_0.

Description

The function `ippsDCTLifterInitAlloc` is declared in the `ippsr.h` file. This function initializes the structure for the DCT calculation and liftering.

The first DCT coefficient C_0 is usually ignored. Therefore, the output of the `ippsDCTLifter` function contains only C_1 to $C_{lenCeps}$ coefficients. However, if the `C0` suffix is specified, C_0 is stored as the last element of the output vector.

The liftering coefficients are calculated according to the below formulae:

For functions without both the `Mul` and `C0` suffixes,

$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val, \quad i = 1,\ldots lenCeps$$

For functions with the `C0` suffix,

$$l_0 = val0$$

$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val, \quad i = 1,\ldots lenCeps$$

For functions with the `Mul` suffix,

$$l_i = pLifter[i-1], \quad i = 1,\ldots lenCeps$$

For functions that have both the `Mul` and `C0` suffixes:

$$l_0 = pLifter[lenCeps-1]$$

$$l_i = pLifter[i-1], \quad i = 1,\ldots lenCeps$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pLifter`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`lenDCT`*, *`lenCeps`*, or *`nLifter`* is less than or equal to 0, or when *`lenDCT`* is less than *`lenCeps`*.
`ippStsMemAllocErr`	Indicates an error when no memory was allocated.

DCTLifterFree

Destroys the structure used for the DCT and liftering.

Syntax

```
IppStatus ippsDCTLifterFree_16s(IppsDCTLifterState_16s* pDCTLifter);
IppStatus ippsDCTLifterFree_32f(IppsDCTLifterState_32f* pDCTLifter);
```

Parameters

pDCTLifter	Pointer to the DCT and liftering structure.

Description

The function `ippsDCTLifterFree` is declared in the `ippsr.h` file. This function closes the DCT and liftering structure by freeing all memory associated with it.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pDCTLifter* pointer is `NULL`.

DCTLifter

Performs the DCT and lifts the DCT coefficients.

Syntax

```
IppStatus ippsDCTLifter_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
IppStatus ippsDCTLifter_32s16s_Sfs (const Ipp32s* pSrc, Ipp16s* pDst, const
    IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
IppStatus ippsDCTLifter_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDCTLifterState_32f* pDCTLifter);
```

Parameters

pSrc	Pointer to the source vector [*lenDCT*].

pDst	Pointer to the output vector [*lenCeps*] or [*lenCeps*+1].
pDCTLifter	Pointer to the DCT and liftering structure.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsDCTLifter is declared in the ippsr.h file. This function first performs the DCT and then lifts the DCT coefficients.

The DCT coefficients are calculated according to the formula:

$$y_i = \sum_{j=1}^{lenDCT} pSrc[j-1] \cdot \cos\left(\frac{\pi \cdot i \cdot (j-0.5)}{lenDCT}\right), \quad i = 0,\ldots lenCeps$$

The output coefficients are weighted as follows:

$$pDst[i-1] = l_i \cdot y_i, \quad i = 1,\ldots lenCeps$$

If the C_0 coefficient is required (*pDCTLifter* is initialized by functions with the C0 suffix), it is stored as the last element of the output vector:

$$pDst[lenCeps] = l_0 \cdot y_0$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pFBank* pointer is NULL.

The following example shows how the feature processing functions can be used for MFCC feature calculation.

Example 8-1 MFCC feature calculation

```
/* Input: samples[] Input samples
sample_number Number of samples
Output: mfccs[12*(sample_num-240)/160] The resulting MFCC coefficients
*/

void Calc_MFCC (Ipp32f *samples, int sample_num, Ipp32f *mfccs) {
Ipp32f* frame_buffer,fbank_buffer,mfcc_cur;
IppsFBankState_32f *fbank;
```

```
IppsDCTLifterState_32f *dctl;
int fft_len,fft_order;
int i,j;
float LogEnergy;

/* Initialize the structures */
ippsMelFBankInitAlloc_32f(&fbank, /* return the structure pointer */
                  &fft_order, /* return the FFT length */
                  400, /* 25ms window/512 point FFT */
                  16000, /* sample rate */
                  64, /* lowest frequency of interest */
                  8000, /* highest frequency of interest */
                  24, /* number of filter banks */
                  1127.0, /* mel-scale factor 1 */
                  700.0, /* mel-scale factor 2 */
                  IPP_FBANK_MELWGT | IPP_POWER_SPECTRUM);
ippsDCTLifterInitAlloc_32f(&dctl, /* return the structure pointer */
                  24, /* filter bank channels */
                  12, /* number of MFCC coefficients */
                  22, /* liftering */
                  1.0); /* no scaling */
fft_len=1<<fft_order;
frame_buffer=ippsMalloc_32f(fft_len);
fbank_buffer=ippsMalloc_32f(24);
mfcc_cur=mfccs;

/* Calculate MFCC features */
for (i=j=0; i+400<sample_num; i+=160,mfcc_cur+=12,j++) {
       /* Organize the input wave data into a frame */
       ippsCopyWithPadding_32f(&samples[i],400,frame_buffer,fft_len);
       ippsDotProd_32f(frame_buffer,frame_buffer,fft_len,&LogEnergy);
       /* Pre-emphasize the input signal with factor 0.97 */
       ippsPreemphasize_32f(frame_buffer,400,0.97);
       frame_buffer[0]*=(1.0-0.97);
```

```
        /* Add the hamming window to the input signal */
        ippsWinHamming_32f_I(frame_buffer,400);
        /* Perform the filter bank analysis */
        ippsEvalFBank_32f(frame_buffer,fbank_buffer,fbank);
        ippsThreshold_LTVal_32f_I(fbank_buffer, 24, 1.0, 1.0);
        ippsLn_32f_I(fbank_buffer, 24);
        /* Perform the DCT analysis and liftering */
        ippsDCTLifter_32f(fbank_buffer,mfcc_cur,dctl);
        mfcc_cur[12] = (float) log ((double) LogEnergy);

}
/* Normalize log energy */
ippsNormEnergy_32f(mfccs+11,12,(sample_num-240)/160,50.0,1.0);
/* Destroy the structures after calculation */
ippsFree(fbank_buffer);
ippsFree(frame_buffer);
ippsFBankFree_32f(fbank);
ippsDCTLifterFree_32f(dctl);
    }
```

NormEnergy

Normalizes a vector of energy values.

Syntax

```
IppStatus ippsNormEnergy_32f(Ipp32f* pSrcDst, int step, int height,
    Ipp32f silFloor, Ipp32f enScale);
IppStatus ippsNormEnergy_16s(Ipp16s* pSrcDst, int step, int height,
    Ipp16s silFloor, Ipp16s val, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_32f(Ipp32f* pSrcDst, int step, int height,
    Ipp32f silFloor, Ipp32f maxE, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_16s(Ipp16s* pSrcDst, int step, int height,
    Ipp16s silFloor, Ipp16s maxE, Ipp16s val, Ipp32f enScale);
```

Parameters

pSrcDst	Pointer to the input and output vector [*height***step*].
step	Sample step in the vector *pSrcDst*.
height	Number of samples for the normalization.
silFloor	Silence floor value.
val	Coefficient value.
maxE	Maximum energy value.
enScale	Energy scale.

Description

The function `ippsNormEnergy` is declared in the `ippsr.h` file. This function normalizes the input vector that contains energy values.
The normalization is performed as follows:

For functions without the RT suffix, the maximum energy value is calculated as

$$maxE = \max_{0 < i < height-1} pSrcDst[i \cdot step]$$

For all functions, assuming *val* = 1 if not specified,

$$minE = maxE - (silFloor \cdot \ln 10)/10$$

$$pSrcDst[i \cdot step] = val - (maxE - \max(pSrcDst[i \cdot step], minE)) \cdot enScale,$$

for $0 \le i < height$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *step* or *height* is less than or equal to 0.

SumMeanVar

Calculates both the sum of a the vector and its square sum.

Syntax

```
IppStatus ippsSumMeanVar_32f(const Ipp32f* pSrc, int srcStep, int height,
    Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_32f_I(const Ipp32f* pSrc, int srcStep, int height,
    Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32f(const Ipp16s* pSrc, int srcStep, int height,
    Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_16s32f_I(const Ipp16s* pSrc, int srcStep, int height,
    Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32s_Sfs(const Ipp16s* pSrc, int srcStep, int
    height, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsSumMeanVar_16s32s_ISfs(const Ipp16s* pSrc, int srcStep, int
    height, Ipp32s* pSrcDstMean, Ipp32s* pSrcDstVar, int width, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the source vector [*height***srcStep*].
srcStep	Row step in the vector *pSrc* .
height	Number of rows in *pSrc*
pDstMean	Pointer to the destination vector that contains the sums [*width*].
pDstVar	Pointer to the destination vector that contains the square sums [*width*].
pSrcDstMean	Pointer to the source and destination vector that contains the sums [*width*].
pSrcDstVar	Pointer to the source and destination vector that contains the square sums [*width*].
width	Number of columns in the source vector *pSrc*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsSumMeanVar` is declared in the `ippsr.h` file. This function calculates both the sums and the square sums of the source vectors as follows:

$$pDstVar[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pDstMean[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] ,$$

for $j = 0,\ldots width - 1$.

The function `ippsSumMeanVar_I` performs the in-place calculation as given by:

$$pSrcDstVar[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pSrcDstMean[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] ,$$

for $j = 0,\ldots width - 1$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pDstMean*, *pDstVar*, *pSrcDstMean*, or *pSrcDstVar* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *srcStep*, *width*, or *height* is less than or equal to 0, or when *width* is greater than *srcStep*.

NewVar

Calculates the variances given the sum and square sum accumulators.

Syntax

```
IppStatus ippsNewVar_32f(const Ipp32f* pSrcMean, const Ipp32f* pSrcVar, Ipp32f*
    pDstVar, int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32f_I(const Ipp32f* pSrcMean, Ipp32f* pSrcDstVar,
    int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32s_Sfs(const Ipp32s* pSrcMean, const Ipp32s* pSrcVar,
    Ipp32s* pDstVar, int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
IppStatus ippsNewVar_32s_ISfs(const Ipp32s* pSrcMean, Ipp32s* pSrcDstVar,
    int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
```

Parameters

pSrcMean	Pointer to the vector that accumulates sums [*width*].
pSrcVar	Pointer to the vector that accumulates square sums [*width*].
pSrcDstVar	Pointer to the square sum accumulators and the resulting variance vector [*width*].
pDstVar	Pointer to the result variance vector [*width*].
width	Length of the variance vector *pDstVar*.
val1, *val2*	Constant coefficients.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsNewVar is declared in the ippsr.h file. This function calculates the variance values given the sum and
square sum accumulators as follows:

$$pDstVar[i] = (pSrcVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2 \quad , i = 0 \ldots width-1$$

whereas the in-place function ippsNewVar_I uses the following formula:

$$pSrcDstVar[i] = (pSrcDstVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2 \ ,$$

for $i = 0, \ldots width - 1$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcMean*, *pSrcVar*, *pDstVar*, or *pSrcDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* is less than or equal to 0.

RecSqrt

Calculates square roots of a vector and their reciprocals.

Syntax

```
IppStatus ippsRecSqrt_32s_Sfs(Ipp32s* pSrcDst, int len, Ipp32s val, int
    scaleFactor);
IppStatus ippsRecSqrt_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
IppStatus ippsRecSqrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
    Ipp32s val, int scaleFactor);
```

Parameters

pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the vector *pSrcDst*.
val	Threshold for processed source values.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsRecSqrt is declared in the ippsr.h file. This function calculates the square root of a vector and then takes the reciprocals. The operations are as follows:

$$pSrcDst[j] = \begin{cases} val & , \ if\ pSrcDst[j] < val \\ \dfrac{1}{\sqrt{pSrcDst[j]}} & , \ otherwise \end{cases}$$

for $j = 0,\ldots,len - 1$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0 or *val* is less than or equal to 0.
ippStsInvZero	Indicates a warning that all *pSrcDst*[i] are less than *val*.

AccCovarianceMatrix

Accumulates covariance matrix.

Syntax

```
IppStatus ippsAccCovarianceMatrix_16s64f_D2L(const Ipp16s** mSrc, int height,
    const Ipp16s* pMean, Ipp64f** mSrcDst, int width, Ipp64f val);
IppStatus ippsAccCovarianceMatrix_32f64f_D2L(const Ipp32f** mSrc, int height,
    const Ipp32f* pMean, Ipp64f** mSrcDst, int width, Ipp64f val);
IppStatus ippsAccCovarianceMatrix_16s64f_D2(const Ipp16s* pSrc, int srcStep,
    int height, const Ipp16s* pMean, Ipp64f* pSrcDst, int width, int dstStep,
    Ipp64f val);
IppStatus ippsAccCovarianceMatrix_32f64f_D2(const Ipp32f* pSrc, int srcStep,
    int height, const Ipp32f* pMean, Ipp64f* pSrcDst, int width, int dstStep,
    Ipp64f val);
```

Parameters

pSrc	Pointer to the input vector [*height***srcStep*].
mSrc	Pointer to the input matrix [*height*] [*width*].
srcStep	Row step in *pSrc*.
pMean	Pointer to the mean vector [*width*].
width	Length of the input matrix row, also length of the mean and variance vectors.
pSrcDst	Pointer to the result vector [*width***dstStep*].
mSrcDst	Pointer to the result matrix [*width*] [*width*].
dstStep	Row step in *pSrcDst*.
height	Number of rows in the input matrix.

val	Value to multiply to each distance.

Description

The function `ippsAccCovarianceMatrix` is declared in the `ippsr.h` file. This function accumulates destination covariance matrix elements according to formulas below.

For functions with the `D2` suffix,

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] + val \cdot \sum_{k=0}^{height-1} (pSrc[k \cdot srcStep + i] - pMean[j]) \cdot (pSrc[k \cdot srcStep + j] - pMean[j])$$

$$pSrcDst[j \cdot dstStep + i] = pSrcDst[i \cdot dstStep + j],$$

$$\text{for } i = 0, \ldots width - 1,\ j = i, \ldots width - 1.$$

For functions with the `D2L` suffix,

$$mSrcDst[i][j] = mSrcDst[i][j] + val \cdot \sum_{k=0}^{height-1} (mSrc[k][i] - pMean[j]) \cdot (mSrc[k][j] - pMean[j])$$

$$mSrcDst[j][i] = mSrcDst[i][j],$$

$$\text{for } i = 0, \ldots width - 1,\ j = i, \ldots width - 1.$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *mSrc*, *pMean*, *pSrcDst*, or *mSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *srcStep* or *dstStep* is less than *width*.

Derivative Functions

Functions described in this section are used to calculate first and second derivatives of feature vectors. These functions process the sequence of input feature vectors and generate the corresponding output feature vectors that contain derivatives.

The input sequence $a_0, \ldots, a_{N-1}$ is stored as a one-dimensional array of length $N \cdot M$, where N is the number of feature vectors and M is the dimension of each feature a_i. Similarly, the output sequence $b_0, \ldots, b_{N-1}$ is also stored as a one-dimensional array of length $N \cdot K$, where K is the dimension of each feature b_j.
Certain constraints on values of M and K must be observed, specifically,

$K \geq M$;

$K \geq 2M$, for generating first derivatives;

$K \geq 3M$, for generating both first and second derivatives.

The `ippsCopyColumn` function copies the input sequence to the output sequence, that is, places the base features into the output sequence. The base features are placed in the first M elements of each output vector. The function `ippsEvalDelta` is then used to calculate

the derivatives. While these two functions are used for general derivative calculation, the functions `ippsDelta` and `ippsDeltaDelta` provide combined but specific functionalities.

As the derivative operation accesses feature vectors both in the history and in the future, special treatment is required for the first `winSize` (delta window size) features, as well as for the last `winSize` features. For the first `winSize` features, the first feature vector is usually repeated to provide the history. For the last `winSize` features, the last feature vector is repeated to provide the future information.

CopyColumn

Copies the input sequence into the output sequence.

Syntax

```
IppStatus ippsCopyColumn_16s_D2(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstWidth, int height);
IppStatus ippsCopyColumn_32f_D2(const Ipp32f* pSrc, int srcWidth,
    Ipp32f* pDst, int dstWidth, int height);
```

Parameters

pSrc	Pointer to the input feature sequence [*height*srcWidth*].
srcWidth	Length of each input feature vector.
pDst	Pointer to the output feature sequence [*height*dstWidth*].
dstWidth	Length of each output feature vector.
height	Number of features in the sequence.

Description

The function `ippsCopyColumn` is declared in the `ippsr.h` file. This function copies the input feature sequence into the output sequence as follows:

$$pDst[j \cdot dstWidth + i] = pSrc[j \cdot srcWidth + i], \quad 0 \le i < srcWidth, \quad 0 \le j < height$$

The unspecified elements in the output sequence remain unchanged.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *height* or *srcWidth* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *dstWidth* is less than *srcWidth*.

EvalDelta

Calculates the derivatives of feature vectors.

Syntax

```
IppStatus ippsEvalDelta_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step, int
    width, int offset, int winSize, Ipp16s val, int scaleFactor);
IppStatus ippsEvalDelta_32f_D2(Ipp32f* pSrcDst, int height, int step, int
    width, int offset, int winSize, Ipp32f val);
IppStatus ippsEvalDeltaMul_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step,
    const Ipp16s* pVal, int width, int offset, int winSize, int scaleFactor);
IppStatus ippsEvalDeltaMul_32f_D2(Ipp32f* pSrcDst, int height, int step, const
    Ipp32f* pVal, int width, int offset, int winSize);
```

Parameters

pSrcDst	Pointer to the input and output sequence [*height***step*].
height	Number of features in *pSrcDst*.
step	Length of each feature in *pSrcDst*.
width	The number of derivatives to be calculated for each feature.
offset	Offset to place the derivative values.
winSize	The delta window size
val	The delta coefficient.
pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsEvalDelta` is declared in the `ippsr.h` file. This function calculates the derivatives for the input feature sequence. The base feature ranges from *offset* to (*offset* + *width* - 1) in each feature vector. The output derivatives are stored in each feature vector next to the base features ranging from (*offset* + *width*) to (*offset* + 2**width* - 1). The operations are detailed as follows:

For `ippsEvalDelta` functions,

$$pSrcDst[i \cdot step + width + j] = val \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$0 \le i < height, \quad offset \le j < offset + width$.

For `ippsEvalDeltaMul` functions,

$$pSrcDst[i \cdot step + width + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$0 \le i < height, \quad offset \le j < offset + width$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, or *winSize* is less than or equal to 0; or *offset* is less than 0; or *height* is less than 2**winSize*.
ippStsStrideErr	Indicates an error when *step* is less than *offset*+2**width*.

Delta

Copies the base features and calculates the derivatives of feature vectors.

Syntax

```
IppStatus ippsDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
    pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);
IppStatus ippsDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
    pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);
IppStatus ippsDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f* pDst,
    int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippsDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f* pDst,
    int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippsDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
    int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
    scaleFactor);
IppStatus ippsDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
    int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
    scaleFactor);
IppStatus ippsDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal, int
    srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
IppStatus ippsDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal, int
    srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Parameters

pSrc	Pointer to the input feature sequence [*height**srcWidth*].
srcWidth	Length of the feature vector in the input sequence *pSrc*.

pDst	Pointer to the output feature sequence
dstStep	Length of the feature vector in the output sequence *pDst*.
height	Number of feature vectors.
val	Delta coefficient.
deltaMode	Execution mode.
pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The functions ippsDelta and ippsDeltaMul are declared in the ippsr.h file. These functions provide a combined functionality of ippsCopyColumn and ippsEvalDelta. First, the input feature vectors are copied to the output sequence. Then the derivatives are calculated.

In the subsequent discussion, the following conditions hold:

The function suffix Win1 or Win2 specifies the delta window size, *winSize* = 1 or 2, respectively;

The function ippsDelta implies the following constraints on the delta coefficients:

$pVal[j] \equiv val, \quad \forall j.$

The execution mode *deltaMode* provides additional controls along the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to IPP_DELTA_BEGIN|IPP_DELTA_END

   Perform the offline delta feature calculation. All base features are assumed available at the time of the calculation. The base features are copied from the input stream *pSrc* to the output stream *pDst* as follows:

$$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \quad 0 \le i < height, \quad 0 \le j < srcWidth. \tag{8.4}$$

   Then the derivatives are calculated as given by:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[\min(i+k, height-1) \cdot dstStep + j] - pDst[\max(i-k, 0) \cdot dstStep + j]\}, \tag{8.5}$$

   for $0 \le i < height, \quad 0 \le j < srcWidth$.

2. *deltaMode* is equal to 0

Perform the online delta feature calculation. The input features are the current segment of a continuous stream. The function `ippsDelta` calculates the partial delta features accordingly. First, the base features are copied as follows:

$$pDst[(i+2\cdot winSize)\cdot dstStep+j] = pSrc[i\cdot srcWidth+j], \tag{8.6}$$

for $0 \le i < height$, $0 \le j < srcWidth$.

Then the derivatives are calculated as given by:

$$pDst[i\cdot dstStep+srcWidth+j] = pVal[j]\cdot \sum_{k=1}^{winSize} k\cdot \{pDst[(i+k)\cdot dstStep+j] - pDst[(i-k)\cdot dstStep+j]\}, \tag{8.7}$$

for $0 \le i - winSize < height$, $0 \le j < srcWidth$.

In this execution mode, the base features in $pDst[i\cdot dstStep+j]$ and the derivatives in $pDst[k\cdot dstStep+srcWidth+j]$ are assumed available through a previous derivative calculation for $0 \le j < srcWidth$, $0 \le i < 2\cdot winSize$, and $0 \le k < winSize$.

3. *deltaMode* is equal to `IPP_DELTA_BEGIN`

   Perform the partial online delta feature calculation, where the beginning of the input stream is known. First, the base features are copied as in equation (8.4). Then, the derivatives are calculated as follows:

$$pDst[i\cdot dstStep+srcWidth+j] = pVal[j]\cdot \sum_{k=1}^{winSize} k\cdot \{pDst[(i+k)\cdot dstStep+j] - pDst[\max(i-k,0)\cdot dstStep+j]\}, \tag{8.8}$$

   for $0 \le i < height - winSize$, $0 \le j < srcWidth$.

4. *deltaMode* is equal to `IPP_DELTA_END`

   Perform the partial online delta feature calculation, where the ending of the input stream is known. First, the base features are copied as in equation (8.6). Then, the derivatives are calculated as follows:

$$pDst[i\cdot dstStep+srcWidth+j] = pVal[j]\cdot \sum_{k=1}^{winSize} k\cdot \{ pDst[\min(i+k, height+2\cdot winSize-1)\cdot dstStep+j] - pDst[(i-k)\cdot dstStep+j]\}$$

for $0 \le i - winSize < height + winSize$, $0 \le j < srcWidth$.

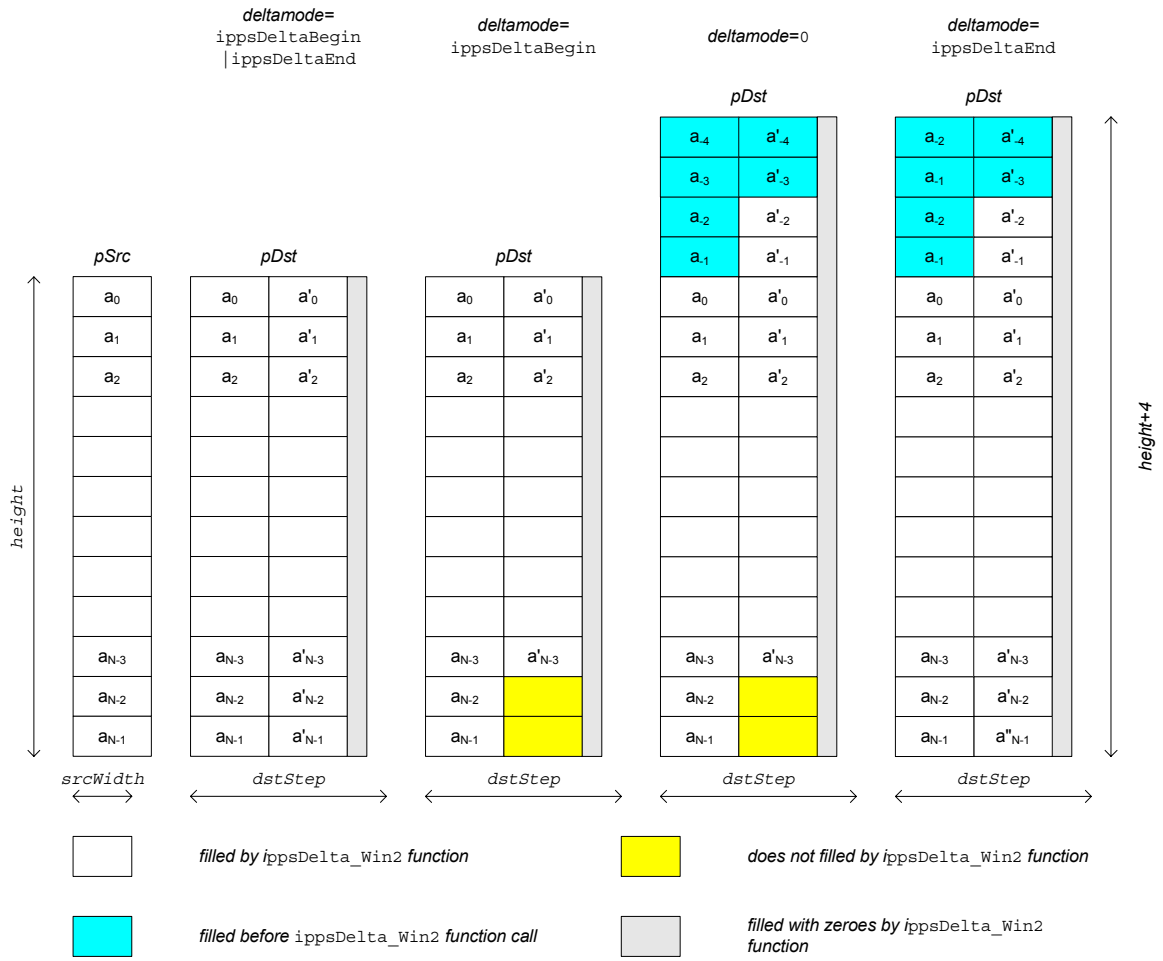
In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the derivatives in $pDst[k \cdot dstStep + srcWidth + j]$ are assumed available through a previous derivative calculation for $0 \le j < srcWidth$, $0 \le i < 2 \cdot winSize$, and $0 \le k < winSize$.

The Figure 8-1 illustrates the above four delta calculation modes.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pVal* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcWidth* is less than or equal to 0; or *height* is less than 0; or *height* is less than *2*winSize* when *deltaMode* is equal to IPP_DELTA_BEGIN ; or *height* is equal to 0 when *deltaMode* is not equal to IPP_DELTA_END.
ippStsStrideErr	Indicates an error when *dstStep* is less than 2**srcWidth*.

Figure 8-1 Execution Modes of ippsDelta_Win2 function



DeltaDelta

Copies the base features and calculates their first and second derivatives.

Syntax

```
IppStatus ippsDeltaDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
    deltaMode, int scaleFactor);
IppStatus ippsDeltaDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
    deltaMode, int scaleFactor);
IppStatus ippsDeltaDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
    pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);
IppStatus ippsDeltaDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
    pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);
IppStatus ippsDeltaDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
    int scaleFactor);
IppStatus ippsDeltaDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
    int scaleFactor);
IppStatus ippsDeltaDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal,
    int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
IppStatus ippsDeltaDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal,
    int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Parameters

pSrc	Pointer to the input feature sequence [*height*srcWidth*].
srcWidth	Length of the input feature in the input sequence *pSrc*.
pDst	Pointer to the output feature sequence.
dstStep	Length of the output feature in the output sequence *pDst*.
height	Number of feature vectors.
val1, val2	The first and second delta coefficients.
deltaMode	Execution mode.

pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The functions `ippsDeltaDelta` and `ippsDeltaDeltaMul` are declared in the `ippsr.h` file. These functions provide a combined functionality of `ippsCopyColumn` and double operations of `ippsEvalDelta`. First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated.

In the subsequent discussion, the following conditions hold:

The function suffix Win1 or Win2 specifies the delta window size, winSize = 1 or 2, respectively;

The function `ippsDeltaDelta` implies the following constraints on the delta coefficients:

$pVal[j] \equiv val1, \ \forall j$, for the first derivative calculation and

$pVal[j] \equiv val2, \ \forall j$, for the second derivative calculation.

The execution mode *deltaMode* provides additional controls along the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to `IPP_DELTA_BEGIN|IPP_DELTA_END`

   Perform the offline delta-delta feature calculation. All base features are assumed available at the time of the calculation. First, the base features are copied from the input stream pSrc to the output stream pDst according to (8.4). Then the first derivatives are calculated as given by (8.5) for $0 \le i < height$, $0 \le j < srcWidth$. Finally, the second derivatives are calculated also by formula (8.5) for $0 \le i < height$, $0 \le j - srcWidth < srcWidth$.

2. *deltaMode* is equal to `0`

   Perform the online delta-delta feature calculation. The input features are the current segment of a continuous stream. The function `ippsDeltaDelta` calculates the partial delta-delta features accordingly.

First, the base features are copied as follows:

$$pDst[(i+3\cdot winSize)\cdot dstStep+j] = pSrc[i\cdot srcWidth+j], \tag{8.9}$$

for $0 \le i < height$, $0 \le j < srcWidth$.

Then the first derivatives are calculated as given by (8.7), for $0 \le i - 2\cdot winSize < height$ and $0 \le j < srcWidth$.

Finally, the second derivatives are calculated also by formula (8.7) for $0 \le i - winSize < height$, $0 \le j - srcWidth < srcWidth$.

In this execution mode, the base features in $pDst[i\cdot dstStep+j]$, the first derivatives in $pDst[k\cdot dstStep+srcWidth+j]$, and the second derivatives in $pDst[l\cdot dstStep+2\cdot srcWidth+j]$ are assumed available through a previous delta-delta calculation for $0 \le j < srcWidth$, $0 \le i < 3\cdot winSize$, $0 \le k < 2\cdot winSize$, and $0 \le l < winSize$.

3. *deltaMode* is equal to IPP_DELTA_BEGIN

   Perform the partial online delta-delta feature calculation, where the beginning of the input stream is known. Initially, the base features are copied as in equation (8.4). Then, the first derivatives are calculated as given by (8.8), for $0 \le i < height - winSize$ and $0 \le j < srcWidth$.

   Finally, the second derivatives are calculated also by formula (8.8) for $0 \le i < height - 2\cdot winSize$ and $0 \le j - srcWidth < srcWidth$.

4. *deltaMode* is equal to IPP_DELTA_END

   Perform the partial online delta-delta feature calculation, where the ending of the input stream is known. Initially, the base features are copied as in equation (8.9). Then, the first derivatives are calculated as follows:

$$pDst[i\cdot dstStep+srcWidth+j] = pVal[j]\cdot \sum_{k=1}^{winSize} k\cdot \{ \tag{8.10}$$

$$pDst[\min(i+k, height+3\cdot winSize-1)\cdot dstStep+j] - pDst[(i-k)\cdot dstStep+j]\}$$

for $0 \le i - 2\cdot winSize < height + winSize$, $0 \le j < srcWidth$.

Finally, the second derivatives are calculated also according to (8.10) where the indexes *i* and *j* vary in the following ranges:

$0 \le i - winSize < height + 2 \cdot winSize,\ 0 \le j - srcWidth < srcWidth$.

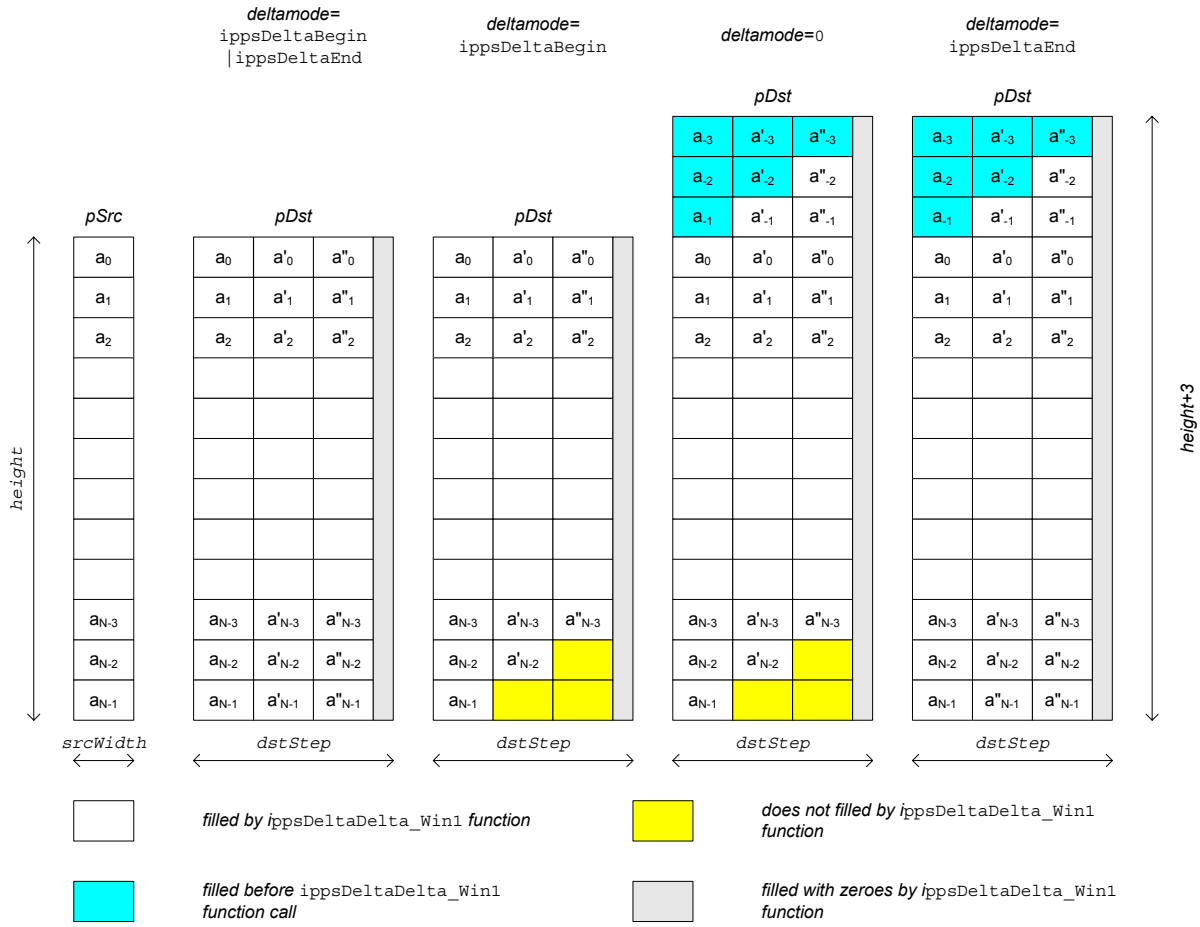
In this execution mode, the base features in $pDst[i \cdot dstStep + j]$, the first derivatives in $pDst[k \cdot dstStep + srcWidth + j]$, and the second derivatives in $pDst[l \cdot dstStep + 2 \cdot srcWidth + j]$ are assumed available through a previous delta-delta calculation for $0 \le j < srcWidth$, $0 \le i < 3 \cdot winSize$, $0 \le k < 2 \cdot winSize$, and $0 \le l < winSize$.

The Figure 8-2 illustrates the above four delta-delta calculation modes.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst* or *pVal* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcWidth* is less than or equal to 0; or *height* is less than 0; or *height* is less than *3*winSize* when *deltaMode* is equal to IPP_DELTA_BEGIN; or *height* is equal to 0 when *deltaMode* is not equal to IPP_DELTA_END.
ippStsStrideErr	Indicates an error when *dstStep* is less than *3*srcWidth*.

Figure 8-2 Execution Modes of `ippsDeltaDelta_Win1` function



Pitch Super Resolution

This section describes functions that can be used for implementing the pitch resolution algorithm (see [Med91]).

Some of these functions are used in the Intel IPP Speech Recognition Sample. See *Speech Processing* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

CrossCorrCoeffDecim

Calculates vector of cross correlation coefficients with decimation.

Syntax

```
IppStatus ippsCrossCorrCoeffDecim_16s32f(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int maxLen, int minLen, Ipp32f* pDst, int dec);
```

Parameters

pSrc1	Pointer to the first input vector [*maxLen*].
pSrc2	Pointer to the second input vector [*maxLen*].
pDst	Pointer to the output vector [(*maxLen*-*minLen*)/*dec*+1].
maxLen	Maximal length of cross correlation.
minLen	Minimal length of cross correlation.
dec	Decimation step.

Description

The function `ippsCrossCorrCoeffDecim` is declared in the `ippsr.h` file. This function calculates the vector of cross correlation coefficients with lengths from *minLen* to *maxLen* using the decimation step *dec* .

Computations are performed as given by the following formula:

$$pDst[k] = \begin{cases} 0, & if\ (\mathbf{x}^k, \mathbf{x}^k)_{dec} \cdot (\mathbf{y}^k, \mathbf{y}^k)_{dec} = 0 \\ \dfrac{(\mathbf{x}^k, \mathbf{y}^k)_{dec}}{\sqrt{(\mathbf{x}^k, \mathbf{x}^k)_{dec}} \cdot \sqrt{(\mathbf{y}^k, \mathbf{y}^k)_{dec}}}, & otherwise \end{cases}$$

where

$$\mathbf{x}_i^k = pSrc1[maxLen - minLen - dec \cdot k + i], \quad \mathbf{y}_i^k = pSrc2[i],$$

$$i = 0, \ldots minLen + dec \cdot k - 1, \quad k = 0, \ldots (maxLen - minLen)/dec,$$

and $(\mathbf{a}, \mathbf{b})_{dec}$ denotes the decimated dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$:

$$(\mathbf{a}, \mathbf{b})_{dec} = \sum_{i=0}^{(len-1)/dec} a[i \cdot dec] \times b[i \cdot dec]$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *minLen* or *dec* is less than or equal to 0, or if *maxLen* is less than *minLen*.

CrossCorrCoeff

Calculates the cross correlation coefficient.

Syntax

```
IppStatus ippsCrossCorrCoeff_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32f* pResult);
IppStatus ippsCrossCorrCoeffPartial_16s32f(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32f val, Ipp32f* pResult);
```

Parameters

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*].
len	Length of the input vectors.
pResult	Pointer to the result cross correlation coefficient value.

val	Value used as the squared magnitude of the first vector.

Description

The functions `ippsCrossCorrCoeff` and `ippsCrossCorrCoeffPartial` are declared in the `ippsr.h` file. These functions calculate the cross correlation coefficient of two vectors according to the formulas given below.

For the `ippsCrossCorrCoeff` function:

$$pResult[0] = \begin{cases} 0, & if\ (\mathbf{x}, \mathbf{x}) \cdot (\mathbf{y}, \mathbf{y}) = 0 \\ \dfrac{(\mathbf{x}, \mathbf{y})}{\sqrt{(\mathbf{x}, \mathbf{x})} \cdot \sqrt{(\mathbf{y}, \mathbf{y})}}, & otherwise \end{cases}$$

For the `ippsCrossCorrCoeffPartial` function:

$$pResult[0] = \begin{cases} 0, & if\ val \cdot (\mathbf{y}, \mathbf{y}) = 0 \\ \dfrac{(\mathbf{x}, \mathbf{y})}{\sqrt{val} \cdot \sqrt{(\mathbf{y}, \mathbf{y})}}, & otherwise \end{cases}$$

where

$\mathbf{x}_i = pSrc1[i-1]$, $\mathbf{y}_i = pSrc2[i-1]$, for $i = 1, \ldots len$,

and $(\mathbf{a}, \mathbf{b})$ denotes the dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, or *pResult* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsBadArgErr`	Indicates an error when *val* is less than 0.

CrossCorrCoeffInterpolation

Calculates interpolated cross correlation coefficient.

Syntax

```
IppStatus ippsCrossCorrCoeffInterpolation_16s32f(const Ipp16s* pSrc1, const
    Ipp16s* pSrc2, int len, Ipp32f* pBeta, Ipp32f* pResult);
```

Parameters

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*+1].
len	Length of the input vectors.
pResult	Pointer to the result cross correlation coefficient value.
pBeta	Pointer to the result value of fractional part of the pitch period.

Description

The function `ippsCrossCorrCoeffInterpolation` is declared in the `ippsr.h` file. This function calculates interpolated cross correlation coefficient *pResult* of two vectors and the fractional part *pBeta* of the pitch period according to the following formulas:

$$pBeta[0] = \beta = \frac{(\boldsymbol{x}, \boldsymbol{y}') \cdot (\boldsymbol{y}, \boldsymbol{y}) - (\boldsymbol{x}, \boldsymbol{y}) \cdot (\boldsymbol{y}, \boldsymbol{y}')}{(\boldsymbol{x}, \boldsymbol{y}') \cdot [(\boldsymbol{y}, \boldsymbol{y}) - (\boldsymbol{y}, \boldsymbol{y}')] + (\boldsymbol{x}, \boldsymbol{y}) \cdot [(\boldsymbol{y}', \boldsymbol{y}') - (\boldsymbol{y}, \boldsymbol{y}')]},$$

and if $0 \le \beta < 1$, then

$$pResult[0] = \frac{(1-\beta) \cdot (\boldsymbol{x}, \boldsymbol{y}) + \beta \cdot (\boldsymbol{x}, \boldsymbol{y}')}{\sqrt{(\boldsymbol{x}, \boldsymbol{x}) \cdot ((1-\beta)^2 \cdot (\boldsymbol{y}, \boldsymbol{y}) + 2\beta \cdot (1-\beta) \cdot (\boldsymbol{y}, \boldsymbol{y}') + \beta^2 \cdot (\boldsymbol{y}', \boldsymbol{y}'))}},$$

where

$$\boldsymbol{x}_i = pSrc1[i-1], \quad \boldsymbol{y}_i = pSrc2[i-1], \quad \boldsymbol{y}_i' = pSrc2[i], \quad i = 1, \ldots len,$$

and $(\boldsymbol{a}, \boldsymbol{b})$ denotes the dot product of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, *pBeta* or *pResult* pointer is `NULL`.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *srcStep* or *dstStep* is less than *width*.
ippStsDivByZero	Indicates a warning that the denominator in formula for *pBeta*[0] has zero value. Operation execution is not aborted.

Model Evaluation

This section describes functions that evaluate the acoustic and language models.

Some of these functions are used in the Intel IPP Speech Recognition Samples. See *Gaussian Calculations* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

AddNRows

Adds N vectors from a vector array.

Syntax

```
IppStatus ippsAddNRows_32f_D2(Ipp32f* pSrc, int height, int offset,
    int step, Ipp32s* pInd, Ipp16u* pAddInd, int rows, Ipp32f* pDst,
    int width, Ipp32f weight);
```

Parameters

pSrc	Pointer to the vector array [*height*step*].
height	Number of rows in the vector array *pSrc*.
offset	Offset to the vector of interest in the vector array *pSrc*.
step	Row step in the vector array *pSrc*.
pInd	Pointer to the index vector [*rows*].
pAddInd	Pointer to the additional index vector [*rows*].
rows	Number of vectors to be added.
pDst	Pointer to the output vector [*width*].
width	Length of the output vector *pDst*.
weight	Weight value to be added to the output.

Description

The function `ippsAddNRows` is declared in the `ippsr.h` file. This function calculates the sum of the *rows* number of vectors, according to the sum of the indexing vectors *pInd* and *pAddInd*, from the vector array *pSrc* as follows:

$$pDst[k] = weight + \sum_{i=0}^{rows} pSrc[k + offset + step \cdot (pInd[i] + pAddInd[i])] \quad ,$$

$0 \le k < width$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pInd*, *pAddInd*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, or *rows* is less than or equal to 0; or (*pInd*[*i*]+*pAddInd*[*i*])or *offset* is less than 0; or (*pInd*[*i*]+*pAddInd*[*i*]) is greater than or equal to *height*.
ippStsStrideErr	Indicates an error when *step* is less than *width* + *offset* .

ScaleLM

Scales vector elements with thresholding.

Syntax

```
IppStatus ippsScaleLM_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len, Ipp16s
    floor, Ipp16s scale, Ipp32s base);
IppStatus ippsScaleLM_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
    floor, Ipp32f scale, Ipp32f base);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the vector *pSrc* or *pDst*.
floor	Threshold value.
scale	Scaling factor.
base	Additive factor.

Description

The function `ippsScaleLM` is declared in the `ippsr.h` file. This function sets threshold on the input vector *pSrc* and scales the vector elements as follows:

$$pDst[n] = scale \cdot \max(pSrc[n], floor) + base, \quad 0 \le n < len$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

LogAdd

Adds two vectors in the logarithmic representation.

Syntax

```
IppStatus ippsLogAdd_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogAdd_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogAdd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
    scaleFactor, IppHintAlgorithm hint);
IppStatus ippsLogAdd_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
    scaleFactor, IppHintAlgorithm hint);
```

Parameters

pSrc	Pointer to the first input vector [*len*].
pSrcDst	Pointer to the second input vector and also the output vector [*len*].
len	Number of elements in the input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.
hint	Suggestion for using specific code for logarithmic addition.

Description

The function `ippsLogAdd` is declared in the `ippsr.h` file. This function adds the *pSrc* and *pSrcDst* vectors, whose elements are in the logarithmic representation.
For functions with floating point arguments, the output vector *pSrcDst*, also in the logarithmic representation, is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} + e^{pSrcDst[i]}), \quad 0 \le i < len.$$

For functions with integer arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} + e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$$0 \le i < len.$$

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The values you can enter for the *hint* argument are listed in Table 7-3, "Flag and Hint Arguments".

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

LogSub

Subtracts a vector from another vector, in the logarithmic representation.

Syntax

```
IppStatus ippsLogSub_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsLogSub_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsLogSub_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsLogSub_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len,
    int scaleFactor);
```

Parameters

pSrc	Pointer to the first input vector [*len*].
pSrcDst	Pointer to the second input and the output vector [*len*].
len	Number of elements in the input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLogSub is declared in the ippsr.h file. This function subtracts the vector *pSrcDst* from the vector *pSrc*, both of them in the logarithmic representation. For functions with floating point arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} - e^{pSrcDst[i]}), \quad 0 \le i < len$$

For functions with integer arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} - e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$0 \le i < len$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsRangeErr	Indicates an error when *pSrc*[*i*] is less than or equal to *pSrcDst*[*i*].

LogSum

Sums vector elements in the logarithmic representation.

Syntax

```
IppStatus ippsLogSum_32f(const Ipp32f* pSrc, Ipp32f* pResult, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogSum_64f(const Ipp64f* pSrc, Ipp64f* pResult, int len,
    IppHintAlgorithm hint);
```

```
IppStatus ippsLogSum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pResult, int len,
    int scaleFactor, IppHintAlgorithm hint);
IppStatus ippsLogSum_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pResult, int len,
    int scaleFactor, IppHintAlgorithm hint);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pResult	Pointer to the result value.
len	Number of elements in the input vector.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.
hint	Suggestion for using specific code for logarithmic addition.

Description

The function `ippsLogSum` is declared in the `ippsr.h` file. This function sums the source vector elements that are taken in logarithmic representation. For functions with floating point arguments, the output value, also in logarithmic representation, is calculated as follows:

$$pResult[0] = \ln \sum_{i=0}^{len-1} e^{pSrc[i]}$$

For functions with integer arguments, the output value is calculated as follows:

$$pResult[0] = 2^{scaleFactor} \cdot \ln\left(\sum_{i=0}^{len-1} e^{pSrc[i] \cdot 2^{-scaleFactor}}\right)$$

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The values you can enter for the *hint* argument are listed in Table 7-3, "Hint Arguments for Fourier Transform Functions".

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pResult* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MahDistSingle

Calculates the Mahalanobis distance for a single observation vector.

Syntax

```
IppStatus ippsMahDistSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s* pMean,
    const Ipp16s* pVar, int len, Ipp32s* pResult, int scaleFactor);
IppStatus ippsMahDistSingle_16s32f(const Ipp16s* pSrc, const Ipp16s* pMean,
    const Ipp16s* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean, const
    Ipp32f* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean, const
    Ipp64f* pVar, int len, Ipp64f* pResult);
IppStatus ippsMahDistSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int len, Ipp64f* pResult);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pMean	Pointer to the mean vector [*len*].
pVar	Pointer to the variance vector [*len*].
len	Number of elements in the input, mean, and variance vectors.
pResult	Pointer to the result.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsMahDistSingle` is declared in the `ippsr.h` file. This function calculates the Mahalanobis distance for the input vector *pSrc*, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows:

$$pResult[0] = \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc*, *pMean*, *pVar*, or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

MahDist

Calculates the Mahalanobis distances for multiple observation vectors.

Syntax

```
IppStatus ippsMahDist_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f* pMean,
    const Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
    Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f* pMean,
    const Ipp64f* pVar, int width, Ipp64f* pDst, int height);
IppStatus ippsMahDist_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, const
    Ipp64f* pVar, int width, Ipp64f* pDst, int height);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pDst	Pointer to the result vector [*height*].
height	Number of rows in the input matrix *mSrc*.

Description

The function ippsMahDist is declared in the ippsr.h file. This function calculates the Mahalanobis distances for multiple input vectors, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows:
For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, \quad 0 \le i < height.$$

For functions with the `D2L` suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, \quad 0 \le i < height.$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

MahDistMultiMix

Calculates the Mahalanobis distances for multiple means and variances.

Syntax

```
IppStatus ippsMahDistMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
    int step, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDistMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
    mVar, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDistMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
    int step, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
IppStatus ippsMahDistMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
    mVar, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
```

Parameters

pMean	Pointer to the mean vector [*height**step*].
pVar	Pointer to the variance vector [*height**step*].
mMean	Pointer to the mean matrix [*height*][*width*].

mVar	Pointer to the variance matrix [*height*] [*width*].
step	Row step in mean and variance vectors.
pSrc	Pointer to the input vector [*width*].
width	Length of the input vector *pSrc*.
pDst	Pointer to the result vector [*height*].
height	Length of the result vector *pDst*.

Description

The function ippsMahDistMultiMix is declared in the ippsr.h file. This function calculates the Mahalanobis distances for a single observation vector but multiple mean and variance pairs as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2, 0 \le i < height.$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2, \quad 0 \le i < height.$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussSingle

Calculates the observation probability for a single Gaussian with an observation vector.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val,
    int scaleFactor);
IppStatus ippsLogGaussSingle_Scaled_16s32f(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val,
    int scaleFactor);
IppStatus ippsLogGaussSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean, const
    Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean, const
    Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_Low_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val, int
    scaleFactor);
IppStatus ippsLogGaussSingle_LowScaled_16s32f(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val, int
    scaleFactor);
```

Case 2: Operation for the diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_DirectVar_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val,
    int scaleFactor);
IppStatus ippsLogGaussSingle_DirectVarScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val,
    int scaleFactor);
IppStatus ippsLogGaussSingle_DirectVar_32f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_DirectVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```

```
IppStatus ippsLogGaussSingle_DirectVar_64f(const Ipp64f* pSrc, const Ipp64f*
    pMean, const Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```

Case 3: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussSingle_IdVar_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
    pMean, int len, Ipp32s* pResult, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussSingle_IdVarScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussSingle_IdVar_32f(const Ipp32f* pSrc, const Ipp32f* pMean,
    int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_IdVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
    pMean, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_IdVar_64f(const Ipp64f* pSrc, const Ipp64f* pMean,
    int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_IdVarLow_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, int len, Ipp32s* pResult, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussSingle_IdVarLowScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int scaleFactor);
```

Case 4: Operation for the block diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_BlockDVar_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, const IppsBlockDMatrix_16s*  pBlockVar, int len, Ipp32s*
    pResult, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussSingle_BlockDVarScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, const IppsBlockDMatrix_16s* pBlockVar, int len, Ipp32f*
    pResult, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussSingle_BlockDVar_32f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const IppsBlockDMatrix_32f* pBlockVar, int len, Ipp32f* pResult,
    Ipp32f val);
IppStatus ippsLogGaussSingle_BlockDVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const IppsBlockDMatrix_32f*  pBlockVar, int len, Ipp64f* pResult,
    Ipp64f val);
IppStatus ippsLogGaussSingle_BlockDVar_64f(const Ipp64f* pSrc, const Ipp64f*
    pMean, const IppsBlockDMatrix_64f*  pBlockVar, int len, Ipp64f* pResult,
    Ipp64f val);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pMean	Pointer to the mean vector [*len*].

pVar	Pointer to the variance vector [*len*].
pBlockVar	Pointer to the block diagonal variance matrix.
len	Number of elements in the input, mean, and variance vectors.
pResult	Pointer to the result.
val	Gaussian constant.
scaleFactor	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussSingle` is declared in the `ippsr.h` file. This function calculates the observation probability for a single observation vector and a Gaussian mixture component. The result *pResult* is in the logarithmic representation.

For functions without the `DirectVar`, `IdVar`, or `BlockDVar` suffix, the covariance matrix is assumed to be inverse diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

For functions with the `DirectVar` suffix, the covariance matrix is assumed to be diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2 / pVar[i]$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2$$

For functions with the `BlockDVar` suffix, the covariance matrix is assumed to be block diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} \sum_{j=0}^{len-1} (pSrc[i] - pMean[i]) \cdot V[i][j] \cdot (pSrc[j] - pMean[j])$$

where $V[i][j]$ is the element of the block diagonal matrix *pVar*.

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulas, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pMean*, *pVar*, *pBlockVar*, or *pResult* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDivByZero`	Indicates a warning that a divisor vector element has zero value (for `DirectVar` functions only).

LogGauss

Calculates the observation probability for a single Gaussian with multiple observation vectors.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGauss_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val,
    int scaleFactor);
IppStatus ippsLogGauss_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s* pMean,
    const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);
IppStatus ippsLogGauss_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
    Ipp32f val, int scaleFactor);
IppStatus ippsLogGauss_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val,
    int scaleFactor);
IppStatus ippsLogGauss_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);
```

```
IppStatus ippsLogGauss_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
    Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);
IppStatus ippsLogGauss_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int height, Ipp64f val);
IppStatus ippsLogGauss_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, const
    Ipp64f* pVar, int width, Ipp64f* pDst, int height, Ipp64f val);
IppStatus ippsLogGauss_Low_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGauss_Low_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val,
    int scaleFactor);
IppStatus ippsLogGauss_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
    Ipp32f val, int scaleFactor);
IppStatus ippsLogGauss_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val,
    int scaleFactor);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGauss_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);
IppStatus ippsLogGauss_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGauss_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val,
    int scaleFactor);
IppStatus ippsLogGauss_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val,
    int scaleFactor);
IppStatus ippsLogGauss_IdVar_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
    pMean, int width, Ipp32f* pDst, int height, Ipp32f val);
IppStatus ippsLogGauss_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
    int width, Ipp32f* pDst, int height, Ipp32f val);
IppStatus ippsLogGauss_IdVar_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
    pMean, int width, Ipp64f* pDst, int height, Ipp64f val);
IppStatus ippsLogGauss_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
    int width, Ipp64f* pDst, int height, Ipp64f val);
```

```
IppStatus ippsLogGauss_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);
IppStatus ippsLogGauss_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);
IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int
    scaleFactor);
IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pDst	Pointer to the result vector [*height*].
height	Number of rows in the input matrix, also the length of the result vector *pDst*.
val	Gaussian constant.
scaleFactor	Scaling factor for intermediate sums.

Description

The function `ippsLogGauss` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors for a Gaussian mixture component. The result *pResult* is in the logarithmic representation.

For functions without the `IdVar` suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional `D2` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, 0 \le i < height$$

For functions with the additional D2L suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, \ 0 \le i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional D2 suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2, \ 0 \le i < height$$

For functions with the additional D2L suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2, \ 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *mSrc*, *pMean*, *pVar*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

LogGaussMultiMix

Calculates the observation probability for multiple Gaussian mixture components.

Syntax

```
IppStatus ippsLogGaussMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const Ipp16s*
    pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
    int scaleFactor);
IppStatus ippsLogGaussMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
    int scaleFactor);
IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst,
    int height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst, int height,
    int scaleFactor);
IppStatus ippsLogGaussMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
    int step, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
    mVar, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
    int step, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
    mVar, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int
    height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
    int scaleFactor);
IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int
    height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, int
    scaleFactor);
```

Parameters

pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mMean	Pointer to the mean matrix [*height*] [*width*].
mVar	Pointer to the variance matrix [*height*] [*width*].
step	Row step in the mean and variance vectors.
pSrc	Pointer to the input vector [*width*].
pSrcDst	Pointer to the input and destination vector [*height*].
width	Number of columns in the mean matrix *mMean*.
height	Number of rows in the mean matrix *mMean*.
scaleFactor	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components. The results are in the logarithmic representation.
The calculations are as follows:

For functions with the `D2` suffix,

$$pSrcDst[i] = pSrcDst[i] - 0.5 \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2$$

$0 \le i < height$.

For functions with the `D2L` suffix,

$$pSrcDst[i] = pSrcDst[i] - 0.5 \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \quad ,$$

$0 \le i < height$.

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussMax

Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMax_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
    Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val);
IppStatus ippsLogGaussMax_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height,
    Ipp64f val);
```

```
IppStatus ippsLogGaussMax_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
    const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussMax_Low_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_Low_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height, Ipp32s
    val, int scaleFactor);
IppStatus ippsLogGaussMax_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMax_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s val,
    int scaleFactor);
IppStatus ippsLogGaussMax_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s val,
    int scaleFactor);
IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
    int scaleFactor);
IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
    scaleFactor);
IppStatus ippsLogGaussMax_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussMax_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int
    height, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst,
    int height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
```

Parameters

pSrc	Pointer to the observation vector [*height***step*].
mSrc	Pointer to the observation matrix [*height*][*width*].
step	Row step in the observation vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pSrcDst	Pointer to the likelihood vector [*height*].
height	Length of the vector *pSrcDst*.
val	Gaussian constant.
scaleFactor	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussMax` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector *pSrcDst* using the "maximum" operation.

For functions without the `IdVar` suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional D2 suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the additional D2L suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional D2 suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the additional D2L suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, *pMean*, *pVar*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

LogGaussMaxMultiMix

Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.

Syntax

```
IppStatus ippsLogGaussMaxMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
    Ipp32s* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal, Ipp32s*
    pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
    Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal, Ipp32f*
    pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
    pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
    pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
    mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
    int height);
```

```
IppStatus ippsLogGaussMaxMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
    pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
    pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
    mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
    int height);
IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
    Ipp32s* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal, Ipp32s*
    pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2(const Ipp16s* pMean,
    const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f*
    pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mMean,
    const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
    Ipp32f* pSrcDst, int height, int scaleFactor);
```

Parameters

pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in the mean and variance vectors.
pSrc	Pointer to the input vector [*width*].
width	Number of columns in the mean and variance matrices.
pVal	Pointer to the weight constant vector [*height*].
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of rows in the mean and variance matrices.
scaleFactor	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussMaxMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst* using the "maximum" operation. The covariance matrix is assumed to be inverse diagonal.

For functions with the D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2 ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) , \quad 0 \le i < height.$$

For functions with the D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) , \quad 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with Low suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, *pVal*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussAdd

Calculates the likelihood probability for multiple observation vectors.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussAdd_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussAdd_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f
    val, int scaleFactor);
IppStatus ippsLogGaussAdd_32f_D2(const Ipp32f** pSrc, int step, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f
    val);
IppStatus ippsLogGaussAdd_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
    const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
    int scaleFactor);
IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
    int scaleFactor);
IppStatus ippsLogGaussAdd_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
```

```
IppStatus ippsLogGaussAdd_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f
    val, int scaleFactor);
IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the observation vector [*height***step*].
mSrc	Pointer to the observation matrix [*height*][*width*].
step	Row step in the observation vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of observation vectors.
val	Weight constant.
scaleFactor	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussAdd` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector *pSrcDst*.

For functions without the `IdVar` suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional D2 suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the additional D2L suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional D2 suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the additional D2L suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pMean*, *pVar*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussAddMultiMix

Calculates the likelihood probability for multiple Gaussian mixture components.

Syntax

```
IppStatus ippsLogGaussAddMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f pVal,
    Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean, const
    Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal, Ipp32f*
    pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
    pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
    pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
    mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
    int height);
IppStatus ippsLogGaussAddMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
    pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
    pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
    mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
    int height);
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2(const Ipp16s* pMean,
    const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f*
    pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mMean,
    const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
    Ipp32f* pSrcDst, int height, int scaleFactor);
```

Parameters

pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in mean and variance vectors (in *pMean* elements).
pSrc	Pointer to the input vector [*width*].
width	Length of the mean and variance matrices.
pVal	Pointer to the weight constant vector [*height*].
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of Gaussian mixture components.
scaleFactor	Scaling factor for intermediate sums.

Description

The function `ippsLogGaussAddMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst*. The covariance matrix is assumed to be inverse diagonal.
For functions with the `D2` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the `D2L` suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, *pVal*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

LogGaussMixture

Calculates the likelihood probability for the Gaussian mixture.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMixture_Scaled_16s32f_D2(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int width, const
    Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_Scaled_16s32f_D2L(const Ipp16s* pSrc, const
    Ipp16s** mMean, const Ipp16s** mVar, int height, int width, const Ipp32f*
    pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int width, const
    Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2L(const Ipp16s* pSrc, const
    Ipp16s** mMean, const Ipp16s** mVar, int height, int width, const Ipp32f*
    pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int height, int step, int width, const Ipp32f* pVal,
    Ipp32f* pResult);
```

```
IppStatus ippsLogGaussMixture_32f_D2L(const Ipp32f* pSrc, const Ipp32f** mMean,
    const Ipp32f** mVar, int height, int width, const Ipp32f* pVal, Ipp32f*
    pResult);
IppStatus ippsLogGaussMixture_64f_D2(const Ipp64f* pSrc, const Ipp64f* pMean,
    const Ipp64f* pVar, int height, int step, int width, const Ipp64f* pVal,
    Ipp64f* pResult);
IppStatus ippsLogGaussMixture_64f_D2L(const Ipp64f* pSrc, const Ipp64f** mMean,
    const Ipp64f** mVar, int height, int width, const Ipp64f* pVal, Ipp64f*
    pResult);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, const
    Ipp16s* pMean, int height, int step, int width, const Ipp32f* pVal, Ipp32f*
    pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2L(const Ipp16s* pSrc, const
    Ipp16s** mMean, int height, int width, const Ipp32f* pVal, Ipp32f* pResult,
    int scaleFactor);
IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc,
    const Ipp16s* pMean, int height, int step, int width, const Ipp32f* pVal,
    Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2L(const Ipp16s* pSrc,
    const Ipp16s** mMean, int height, int width, const Ipp32f* pVal, Ipp32f*
    pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVar_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pMean, int height, int step, int width, const Ipp32f* pVal, Ipp32f*
    pResult);
IppStatus ippsLogGaussMixture_IdVar_32f_D2L(const Ipp32f* pSrc, const Ipp32f**
    mMean, int height, int width, const Ipp32f* pVal, Ipp32f* pResult);
IppStatus ippsLogGaussMixture_IdVar_64f_D2(const Ipp64f* pSrc, const Ipp64f*
    pMean,  int height, int step, int width, const Ipp64f* pVal, Ipp64f*
    pResult);
IppStatus ippsLogGaussMixture_IdVar_64f_D2L(const Ipp64f* pSrc, const Ipp64f**
    mMean, int height, int width, const Ipp64f* pVal, Ipp64f* pResult);
```

Parameters

pSrc	Pointer to the input vector [*width*].
pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mMean	Pointer to the mean matrix [*height*] [*width*].

mVar	Pointer to the variance matrix [*height*] [*width*].
height	Number of Gaussian mixture components.
step	Row step in mean and variance vectors (in *pMean* elements).
width	Length of the mean and variance matrix rows; also length of the *pSrc* vector.
pVal	Pointer to the vector of weight constants [*height*].
pResult	Pointer to the output mixture value.
scaleFactor	Scaling factor for intermediate sums.

Description

The function ippsLogGaussMixture is declared in the ippsr.h file. This function calculates observation probability for the Gaussian mixture and returns it in *pResult*.

For functions that do not have the IdVar suffix in their names, the covariance matrix is assumed to be inverse diagonal.

For these functions that also have the additional D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2,$$

for $i = 0, \ldots height - 1$, and

$$pResult[0] = \ln \sum_{i=0}^{height-1} e^{V[i]} .$$

For these functions with the D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 ,$$

$i = 0, \ldots height - 1$.

For functions that have the `IdVar` suffix in their names, the covariance matrix is assumed to be identity.

For these functions that have the additional D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[j] - pMean[i \cdot step + j])^2 \quad ,$$

for $i = 0,\ldots height - 1$, and

$$pResult[0] = \ln \sum_{i=0}^{height-1} e^{V[i]}$$

For these functions with the additional D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[j] - mMean[i][j])^2 \quad , \quad i = 0,\ldots height - 1.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, *pMean*, *mMean*, *pVar*, *mVar*, *pVal*, or *pResult* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

LogGaussMixtureSelect

Calculates the likelihood probability for the Gaussian mixture using Gaussian selection.

Syntax

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int step, int width, const Ipp32s* pVal,
    const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f none,
    int scaleFactor);
IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s** mMean, const Ipp16s** mVar, int width, const Ipp32s* pVal,
    const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f none,
    int scaleFactor);
IppStatus ippsLogGaussMixture_Select_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int step, int width, const Ipp32f* pVal, const
    Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);
IppStatus ippsLogGaussMixture_Select_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f** mMean, const Ipp32f** mVar, int width, const Ipp32f* pVal, const
    Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2(const Ipp16s* pSrc,
    const Ipp16s* pMean, int step, int width, const Ipp32s* pVal, const Ipp8u*
    pSign, int height, Ipp32s* pResult, int frames, Ipp32f none, int
    scaleFactor);
IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s** mMean, int width, const Ipp32s* pVal, const Ipp8u* pSign, int
    height, Ipp32s* pResult, int frames, Ipp32f none, int scaleFactor);
IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2(const Ipp32f* pSrc, const
    Ipp32f* pMean, int step, int width, const Ipp32f* pVal, const Ipp8u* pSign,
    int height, Ipp32f* pResult, int frames, Ipp32f none);
IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f** mMean, int width, const Ipp32f* pVal, const Ipp8u* pSign, int
    height, Ipp32f* pResult, int frames, Ipp32f none);
```

Parameters

pSrc	Pointer to the input vector [*frames***step*].
pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mSrc	Pointer to the input matrix [*frames*][*width*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
pVal	Pointer to the vector of weight constants [*height*].
pSign	Pointer to the vector of Gaussian calculation signs [*frames**(*height*+7)/8] (in bytes).
step	Row step in mean, variance and input vectors (in *pMean* elements).
width	Length of the mean, variance and input matrix rows.
height	Number of Gaussian mixture components.
pResult	Pointer to the vector of output mixture values [*frames*].
frames	Number of input vectors; also the length of *pSign* row.
none	Result value if no Gaussian is calculated.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function `ippsLogGaussMixture_Select` is declared in the `ippsr.h` file. This function calculates observation probabilities for the Gaussian mixture and *frames* input vectors and returns them in *pResult* vector. Gaussians with corresponding signs equal to zero are not calculated. If no Gaussian is calculated for the input vector, *none* value is returned for this vector.

Let $s(i,k)$ be the k-th bit of the i-th row of *pSign*..

For functions that do not have the `IdVar` suffix in their names, the covariance matrix is assumed to be inverse diagonal.

For these functions that also have the additional `D2` suffix,

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot \left(pSrc[k \cdot step + j] - pMean[i \cdot step + j]\right)^2$$

for k = 0,... *frames* - 1, i = 0,... *height* - 1 and $s(i,k) \neq 0$,

and

$$pResult[k] = \begin{cases} none, & \text{if } s(i,k) = 0 \quad \text{for} \quad i = 0,\ldots height-1 \\ \ln \sum_{\substack{i=0 \\ s(i,k)\neq 0}}^{height-1} e^{V[i,k]}, & \text{otherwise} \end{cases}$$

For these functions with the `D2L` suffix,

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot \left(mSrc[k][j] - mMean[i][j]\right)^2$$

for k = 0,... *frames* - 1, i = 0,... *height* - 1 and $s(i,k) \neq 0$,
and the *pResult* vector is computed by the same formula as given above.

For functions that have the `IdVar` suffix in their names, the covariance matrix is assumed to be identity.

For these functions that also have the additional `D2` suffix,

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} \left(pSrc[k \cdot step + j] - pMean[i \cdot step + j]\right)^2$$

for k = 0,... *frames* - 1, i = 0,... *height* - 1 and $s(i,k) \neq 0$,
and the *pResult* vector is computed by the same formula as given above.

For these functions with the `D2L` suffix,

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} \left(mSrc[k][j] - mMean[i][j]\right)^2$$

for k = 0,... *frames* - 1, i = 0,... *height* - 1 and $s(i,k) \neq 0$,
and the *pResult* vector is computed by the same formula as given above.

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc*, *pMean*, *pVar*, *mSrc*, *mMean*, *mVar*, *pSign*, *pVal* or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *width*, *height*, or *frames* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.
ippStsNoGaussian	Indicates a warning when no Gaussian is calculated for one of the input vectors.

BuildSignTable

Fills sign table for Gaussian mixture calculation.

Syntax

```
IppStatus ippsBuildSignTable_8u1u(const Ipp32s* pIndx, int num, const Ipp8u**
    mShortlist, int clust, int width, int shift, Ipp8u* pSign, int frames, int
    comps);
IppStatus ippsBuildSignTable_Var_8u1u(const Ipp32s* pIndx, const int* pNum,
    const Ipp8u** mShortlist, int clust, int width, int shift, Ipp8u* pSign, int
    frames, int comps);
```

Parameters

pIndx	Pointer to the cluster indexes vector ([*frames***num*] or sum of *pNum* elements).
num	Number of clusters for each input vector.
pNum	Number of clusters vector [*frames*].
mShortlist	Pointer to the shortlist matrix [*clust***width*] (in bytes).
clust	Number of rows in shortlist vector (equal to the codebook size).
width	Row length in shortlist matrix in bytes.
shift	First element displacement in shortlist vector rows (in bits).
pSign	Pointer to the output vector of Gaussian calculation signs [*frames**(*comps*+7)/8] (in bytes).
frames	Number of input vectors (rows in *pSign* vector).
comps	Number of Gaussian mixture components (bit columns in *pSign* vector).

Discussion

The functions `ippsBuildSignTable` are declared in the `ippsr.h` file. These functions build sign table for Gaussian mixture calculation using Gaussian selection technique. The sign table is used as the input argument for `ippsLogGaussMixture_Select` functions. The input vectors could activate one or several codebook clusters. Shortlist table rows correspond to clusters, and its columns – to Gaussians. Row substring of length *clust* indicates if the cluster is active for a mixture component.

Let $l(i) = i \cdot num$ for `ippsBuildSignTable` function, and

$$l(i) = \sum_{j=0}^{i-1} pNum[j]$$ for `ippsBuildSignTable_Var` function,

for $i = 0,\ldots frames - 1$.

Let also c_k be the bit substring of the k-th row of *mShortlist*, consisting of bits *shift*,...*shift+comps-1* , for $k = 0,\ldots clust - 1$.

Then elements $l(i),\ldots l(i+1)-1$ of the *pIndx* vector are active cluster numbers for i-th input vector. The i-th row of *pSign* is set to the logical sum of substrings

$$c_{l(i)},\ldots c_{l(i+1)-1}, \quad i = 0,\ldots frames - 1.$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pIndx*, *pSign*, *mShortlist* or *pNum* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *clust*, *width*, *frames*, *comps*, *num*, or one of *pNum* vector elements is less than or equal to 0, or when *shift* is less than 0.
`ippStsStrideErr`	Indicates an error when *width* is less than (*shift*+*comps*+7)/8.
`ippStsBadArgErr`	Indicates an error when one of *pIndx* vector elements is less than 0, or greater than or equal to *clust*.

FillShortlist_Row

Fills row-wise shortlist table for Gaussian selection.

Syntax

```
IppStatus ippsFillShortlist_Row_1u(const Ipp32s* pIndx, int height, int num,
    Ipp8u** mShortlist, int clust, int width, int shift);
IppStatus ippsFillShortlist_RowVar_1u(const Ipp32s* pIndx, const int* pNum, int
    height, Ipp8u** mShortlist, int clust, int width, int shift);
```

Parameters

pIndx	Pointer to the cluster indexes vector ([*num***clust*] or sum of *pNum* elements).
height	Number of Gaussian mixture components.
num	Number of Gaussian means for each cluster.
pNum	Number of Gaussians vector [*clust*].
mShortlist	Pointer to the shortlist matrix [*clust***width*] (in bytes).
clust	Number of rows in shortlist matrix (equal to the codebook size).
width	Row length in shortlist matrix (in bytes).
shift	First element displacement in shortlist matrix rows (in bits).

Discussion

The function `ippsFillShortlist_Row` is declared in the `ippsr.h` file. The function fills the shortlist table and provides that this table contains at least one Gaussian of the mixture for each cluster. This is done by quantizing of cluster centroids on codebook made of Gaussian mixture means. Quantization results are placed to *pIndx* vector.

Bits in the shortlist table *mShortlist* that correspond to one or more cluster centroids nearest to Gaussian mixture means are set to 1. The shortlist is the bit table with elements indicating whether the *i*-th Gaussian should be calculated if the input vector activates the *k*-th cluster of the codebook. The number of rows is equal to the codebook size, the number of columns is equal to the overall number of Gaussians in the model. Row step is chosen so as to start each row from the byte boundary.

The shortlist table should be zeroed after definition. Components of mixtures have sequential indexes. The *shift* argument is the index of the first mixture component in the model, and the *height* argument is the number of the components in the mixture.

Let $l(k) = k \cdot num$ for `ippsFillShortlist_Row` function,

$$l(k) = \sum_{j=0}^{k-1} pNum[j]$$ for `ippsFillShortlist_RowVar` function,

and $c(k,i)$ be equal to $(shift + i)$-th bit in the k-th row of *mShortlist*, for $i = 0,\ldots height - 1$, $k = 0,\ldots clust - 1$.

Then *height* columns of the shortlist table are filled as:

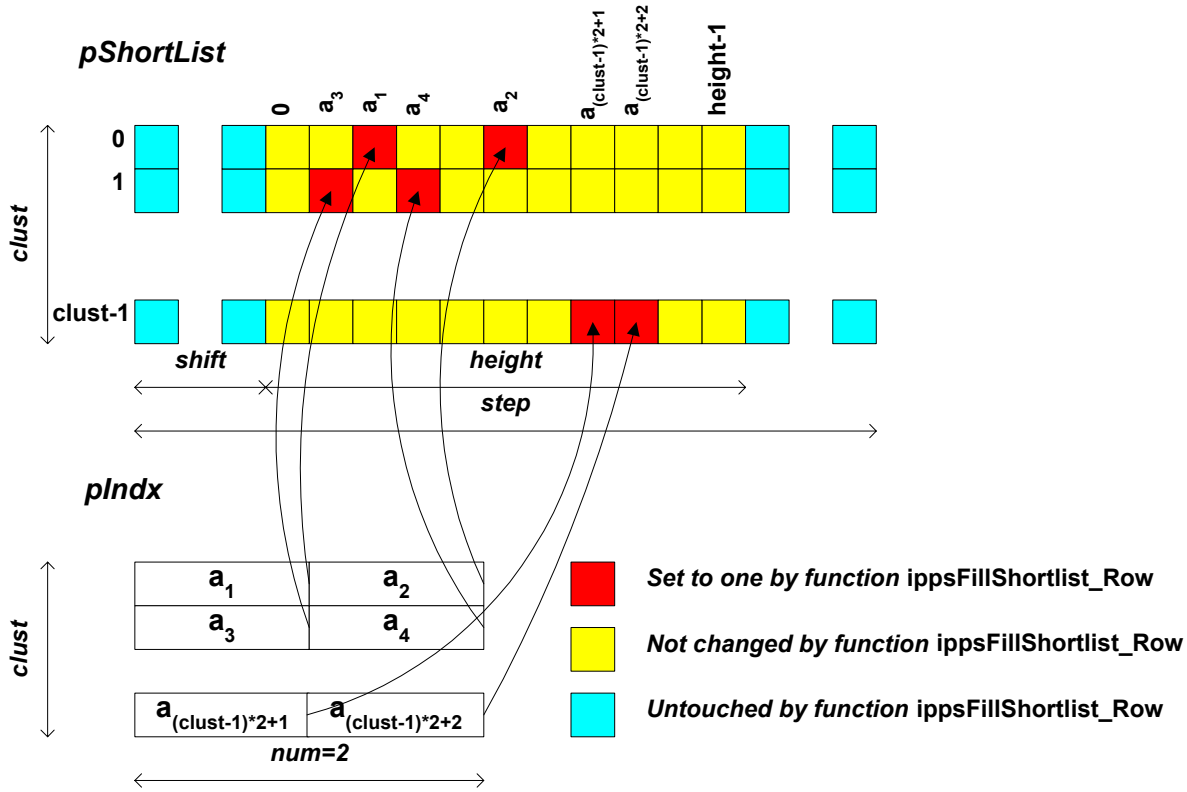
$c(k, pIndx[m]) = 1$, $m = l(k),\ldots l(k+1)-1$, $k = 0,\ldots clust - 1$.

The Figure 8-3 illustrates the execution of this functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pIndx* or `pShortlist` pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *clust*, *height*, *width*, *num* or one of *pNum* vector elements is less than or equal to 0, or when *shift* is less than 0.
`ippStsStrideErr`	Indicates an error when *width* is less than `(shift+height+7)/8`.
`ippStsBadArgErr`	Indicates an error when one of *pIndx* vector elements is less than 0, or greater than or equal to *height*.

Figure 8-3 Execution of ippsFillShortlist_Row function for num=2



FillShortlist_Column

Fills column-wise shortlist table for Gaussian selection.

Syntax

```
IppStatus ippsFillShortlist_Column_1u(const Ipp32s* pIndx, int num, Ipp8u**
    mShortlist, int clust, int width, int shift, int height);
IppStatus ippsFillShortlist_ColumnVar_1u(const Ipp32s* pIndx, const int* pNum,
    Ipp8u** mShortlist, int clust, int width, int shift, int height);
```

Parameters

pIndx	Pointer to the cluster indexes vector ([*num***height*] or sum of *pNum* elements).
height	Number of Gaussian mixture components.
num	Number of clusters for each Gaussian mean.
pNum	Number of clusters vector [*height*].
mShortlist	Pointer to the shortlist matrix [*clust***width*] (in bytes).
clust	Number of rows in shortlist matrix (equal to the codebook size).
width	Row length in shortlist matrix (in bytes).
shift	First element displacement in shortlist matrix rows (in bits).

Discussion

The function ippsFillShortlist_Column is declared in the ippsr.h file. This function fills the part of the shortlist table corresponding to the Gaussian mixture.

The shortlist is the bit table with elements indicating whether the *i*-th Gaussian should be calculated if the input vector activates the *k*-th cluster of the codebook. The number of rows is equal to the codebook size, the number of columns is equal to the overall number of Gaussians in the model. Row step is chosen so as to start each row from the byte boundary.
The shortlist table should be zeroed after definition. Components of mixtures have sequential indexes. The *shift* argument is the index of the first mixture component in the model, and the *height* argument is the number of the components in the mixture.

After means of Gaussian mixture are quantized by one of ippsVQSingle_Sort, VQSingle_Thresh functions, quantization results can be used to fill the shortlist table. Bits corresponding to active clusters for mixture components are set to 1.

Let $l(i) = i \cdot num$ for ippsFillShortlist_Column function,

$$l(i) = \sum_{j=0}^{i-1} pNum[j]$$ for ippsFillShortlist_ColumnVar function,

and $c(k,i)$ be equal to ($shift + i$)-th bit in the k-th row of *mShortlist*,
for $i = 0,\ldots height - 1$, $k = 0,\ldots clust - 1$.

Then *height* columns of the shortlist table are filled as:

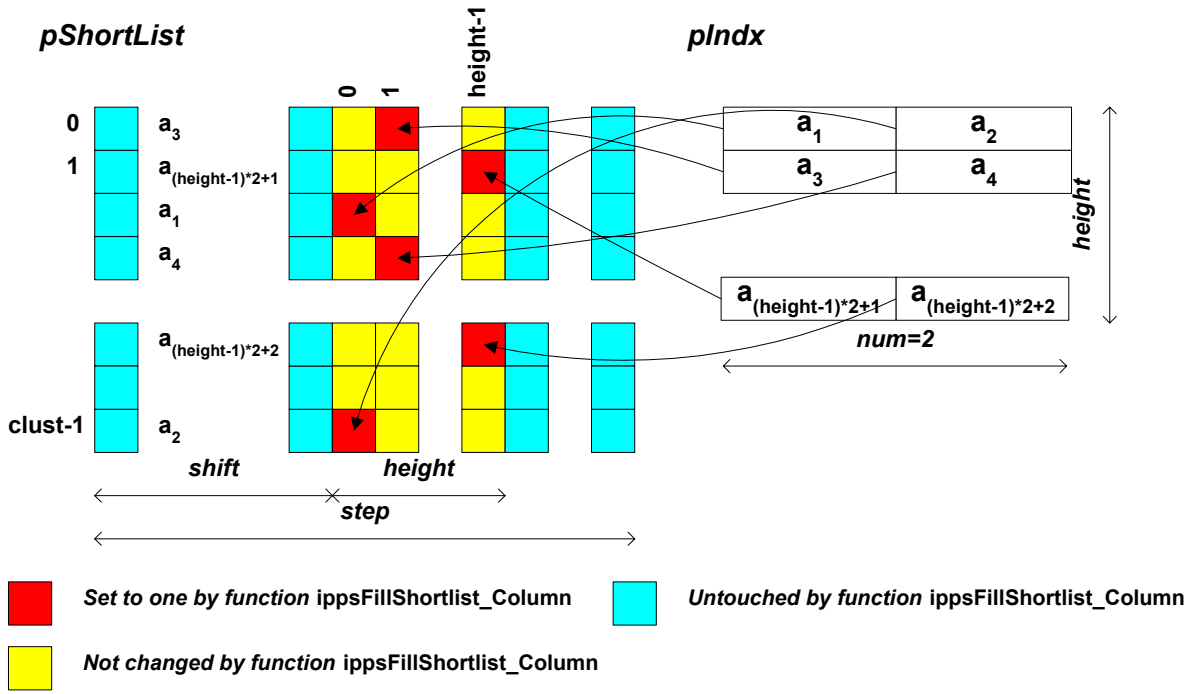
$c(pIndx[m], i) = 1$, $m = l(i),\ldots l(i+1)-1$, $i = 0,\ldots height - 1$.

The Figure 8-4 illustrates the execution of this functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pIndx* or pShortlist pointer is NULL.
ippStsSizeErr	Indicates an error when *clust*, *height*, *width*, *num* or one of *pNum* vector elements is less than or equal to 0, or when *shift* is less than 0.
ippStsStrideErr	Indicates an error when width is less than (shift+*height*+7)/8.
ippStsBadArgErr	Indicates an error when one of *pIndx* vector elements is less than 0, or greater than or equal to *clust*.

Figure 8-4 Execution of ippsFillShortlist_Column function for num=2



DTW

Computes the distance between observation and reference vector sequences using Dynamic Time Warping algorithm.

Syntax

```
IppStatus ippsDTW_L2_8u32s_D2Sfs(const Ipp8u* pSrc1, int height1, const Ipp8u*
    pSrc2, int height2, int width, int step, Ipp32s* pDist, int delta, Ipp32s
    beam, int scaleFactor);
IppStatus ippsDTW_L2_8u32s_D2LSfs(const Ipp8u** mSrc1, int height1, const
    Ipp8u** mSrc2, int height2, int width, Ipp32s* pDist, int delta, Ipp32s
    beam, int scaleFactor);
IppStatus ippsDTW_L2Low_16s32s_D2Sfs(const Ipp16s* pSrc1, int height1, const
    Ipp16s* pSrc2, int height2, int width, int step, Ipp32s* pDist, int delta,
    Ipp32s beam, int scaleFactor);
IppStatus ippsDTW_L2Low_16s32s_D2LSfs(const Ipp16s** mSrc1, int height1, const
    Ipp16s** mSrc2, int height2, int width, Ipp32s* pDist, int delta, Ipp32s
    beam, int scaleFactor);
IppStatus ippsDTW_L2_32f_D2(const Ipp32f* pSrc1, int height1, const Ipp32f*
    pSrc2, int height2, int width, int step, Ipp32f* pDist, int delta, Ipp32f
    beam);
IppStatus ippsDTW_L2_32f_D2L(const Ipp32f** mSrc1, int height1, const Ipp32f**
    mSrc2, int height2, int width, Ipp32f* pDist, int delta, Ipp32f beam);
```

Parameters

pSrc1	Pointer to the first input (observation) vector **x** [*height1***step*].
pSrc2	Pointer to the second input (reference) vector **y** [*height2***step*].
mSrc1	Pointer to the first input (observation) matrix **x** [*height1*] [*width*].
mSrc2	Pointer to the second input (reference) matrix **y** [*height2*] [*width*].
height1	Number of rows in the first input matrix (*N1*).
height2	Number of rows in the second input matrix (*N2*).
width	Vector dimension. Equal to the length of the input matrices row (*M*).

step	Row step in *pSrc1* and *pSrc2*.
pDist	Pointer to the distance value.
beam	Beam value, used if positive.
delta	Endpoint constraint relaxation value.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

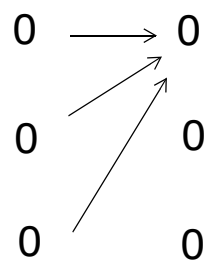
Description

The function `ippsDTW` is declared in the `ippsr.h` file. This function calculates the distance between two vector sequences using the Dynamic Time Warping (DTW) principle. Computations are performed as follows:

$$pDist[0] = \min_{W} \sum_{i=0}^{N1-1} \left| \mathbf{x}_i - \mathbf{y}_{W(i)} \right| ,$$

when $0 \le W(0) \le \delta$, $0 \le W(i) - W(i-1) \le 2$, $i = 1, ..N1-1$, $N2-1-\delta \le W(N1-1) \le N2-1$.

Local constraint on DTW path search looks as follows:



To compute distance between vectors *a* and *b*, the function uses L2-norm as given by:

$$|a-b| = \sqrt{\sum_{j=0}^{M-1} (a(j) - b(j))^2}$$

If the *beam* value is positive, paths *V* that are outside the beam, i.e., satisfy the condition

$$\sum_{i=0}^{k} \left| \mathbf{x}_i - \mathbf{y}_{V(i)} \right| > \min_{W} \sum_{i=0}^{k} \left| \mathbf{x}_i - \mathbf{y}_{W(i)} \right| - beam \ , \ k = 0, \ ..N1-1$$

are pruned away.

The global constraints on path search are described by the following formulas.

The upper boundaries are:

$J=2i-1+\delta$; $J=i/2-(N1/2-N2)+\delta$;

The lower boundaries are:

$J=i/2+1/2-\delta$; $J=2i-(2*N1-N2)-\delta$.

Note that if *height1<height2* , then *pSrc1* interchanges with *pSrc2*, that is, *pSrc1* becomes the reference vector and *pSrc2* is assumed to be the test vector.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, *mSrc1*, *mSrc2*, or *pDist* pointer is NULL.
ippStsSizeErr	Indicates an error when *height1*, *height2*, or *width* is less than or equal to 0, or *delta* is less than 0 or greater than *height2*.
ippStsStrideErr	Indicates an error when *step* is less than *width*.
ippStsNoOperation	Indicates that there are now admissible paths for *height1*, *height2* and *delta* values.

Model Estimation

This section describes functions that are needed to estimate the parameters of the acoustic and language models.

Some of these functions are used in the Intel IPP Speech recognition Sample. See *Gaussian Calculations* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

MeanColumn

Computes the mean values for the column elements.

Syntax

```
IppStatus ippsMeanColumn_16s_D2(const Ipp16s* pSrc, int height, int step,
    Ipp16s* pDstMean, int width);
IppStatus ippsMeanColumn_16s_D2L(const Ipp16s** mSrc, int height, Ipp16s*
    pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
    Ipp32f* pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
    pDstMean, int width);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*.
step	Row step in the input vector (measured in *pSrc* elements).
pDstMean	Pointer to the output mean vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the output mean vector *pDstMean*.

Description

The function `ippsMeanColumn` is declared in the `ippsr.h` file. This function calculates the mean values for the column elements of the input matrix as follows:

For functions with the `D2` suffix,

$$pDstMean[j] = \frac{1}{height} \sum_{i=1}^{height-1} pSrc[i \cdot step + j], \quad 0 \le j < width$$

For functions with the `D2L` suffix,

$$pDstMean[j] = \frac{1}{height} \sum_{i=1}^{height-1} mSrc[i][j], \quad 0 \le j < width$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pDstMean* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

VarColumn

Calculates the variances for the column elements.

Syntax

```
IppStatus ippsVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int step,
    Ipp16s* pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height, Ipp16s*
    pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
    Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);
IppStatus ippsVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
    pSrcMean, Ipp32f* pDstVar, int width);
```

Parameters

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*.
step	Row step in the input vector *pSrc*.
pSrcMean	Pointer to the input mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the input mean vector *pSrcMean* and the output variance vector *pDstVar*.
scaleFactor	Scale factor, refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsVarColumn` is declared in the `ippsr.h` file. This function calculates the variances for the column elements of the matrix as follows:
For functions with the `D2` suffix,

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} pSrc[i \cdot step + j]^2}{height - 1},$$

$0 \le j < width$.

For functions with the `D2L` suffix

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} mSrc[i][j]^2}{height - 1},$$

$0 \le j < width$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, *pSrcMean*, or *pDstVar* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* is less than or equal to 0 or *height* is less than or equal to 1.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

MeanVarColumn

Calculates the means and variances for the column elements of a matrix.

Syntax

```
IppStatus ippsMeanVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int step,
    Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height, Ipp16s*
    pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
```

```
IppStatus ippsMeanVarColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int height,
    int step, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
    Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
    Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsMeanVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
    pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsMeanVarColumn_16s16s32s_D2(const Ipp16s* pSrc, int height, int
    step, Ipp16s* pDstMean, Ipp32s* pDstVar, int width);
IppStatus ippsMeanVarColumn_16s16s32s_D2L(const Ipp16s** mSrc, int height,
    Ipp16s* pDstMean, Ipp32s* pDstVar, int width);
```

Parameters

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc* .
step	Row step in the input vector *pSrc* .
pDstMean	Pointer to the output mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the output mean vector *pDstMean* and variance vector *pDstVar*.
scaleFactor	Scale factor, refer to "Integer Scaling" in Chapter 2. *scaleFactor* argument (and integer scaling) is used for *pDstVar* only, while *pDstMean* elements are not scaled.

Discussion

The function ippsMeanVarColumn is declared in the ippsr.h file. This function calculates both the means and the variances for the column elements of the input matrix. See functions ippsMeanColumn and ippsVarColumn for calculation details.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *mSrc*, *pDstMean*, or *pDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* is less than or equal to 0 or *height* is less than or equal to 1.

ippStsStrideErr	Indicates an error when *step* is less than *width*.

WeightedMeanColumn

Computes the weighted mean values for the column elements.

Syntax

```
IppStatus ippsWeightedMeanColumn_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pWgt, int height, Ipp32f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pWgt, int height, Ipp32f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pWgt, int height, Ipp64f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pWgt, int height, Ipp64f* pDstMean, int width);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*] [*width*].
pWgt	Pointer to the weight vector [*height*].
height	Number of rows in the input matrix.
step	Row step in the input vector (measured in *pSrc* elements).
pDstMean	Pointer to the output mean vector [*width*].
width	Number of columns in the input matrix, and also the length of the output mean vector *pDstMean*.

Description

The function ippsWeightedMeanColumn is declared in the ippsr.h file. This function calculates the weighted mean values for the column elements of the input matrix as follows:

For functions with the D2 suffix,

$$pDstMean[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot pSrc[i \cdot step + j] \ , \ 0 \le j < width \ .$$

For functions with the D2L suffix,

$$pDstMean[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot mSrc[i][j] \ , \ 0 \le j < width \ .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pWgt*, or *pDstMean* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

WeightedVarColumn

Computes the weighted variance values for the column elements.

Syntax

```
IppStatus ippsWeightedVarColumn_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar, int
    width);
IppStatus ippsWeightedVarColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);
IppStatus ippsWeightedVarColumn_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar, int
    width);
IppStatus ippsWeightedVarColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar, int width);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*] [*width*].
pWgt	Pointer to the weight vector [*height*].
height	Number of rows in the input matrix.

step	Row step in the input vector (measured in *pSrc* elements).
pSrcMean	Pointer to the input mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix, and also the length of the input mean vector *pSrcMean* and the output variance vector *pDstVar*.

Description

The function `ippsWeightedVarColumn` is declared in the `ippsr.h` file. This function calculates the weighted variance values for the column elements of the input matrix as follows:

For functions with the `D2` suffix,

$$pDstVar[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot pSrc[i \cdot step + j]^2 - pSrcMean[j]^2 \ , \ 0 \le j < width \ .$$

For functions with the `D2L` suffix,

$$pDstVar[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot mSrc[i][j]^2 - pSrcMean[j]^2 \ , \ 0 \le j < width \ .$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mSrc*, *pWgt*, *pSrcMean*, or *pDstVar* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *height* or *width* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

WeightedMeanVarColumn

Computes weighted mean and variance values for the column elements.

Syntax

```
IppStatus ippsWeightedMeanVarColumn_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int width);
```

Parameters

pSrc	Pointer to the input vector [*height***step*].
mSrc	Pointer to the input matrix [*height*][*width*].
pWgt	Pointer to the weight vector [*height*].
height	Number of rows in the input matrix.
step	Row step in the input vector (measured in *pSrc* elements).
pDstMean	Pointer to the output mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix, and also the length of the output mean vector *pSrcMean* and the output variance vector *pDstVar*.

Description

The function `ippsWeightedMeanVarColumn` is declared in the `ippsr.h` file. This function calculates both weighted means and weighted variances for the column elements of the input matrix. See functions `ippsWeightedMeanColumn` and `ippsWeightedVarColumn` for calculation details.

The code example below shows how you can use the function `ippsWeightedMeanVarColumn`.

Example 8-2 Weights, Means and Variances EM Re-Estimation

```
/* Input:  int height;              // mixture components number
           int width;               // observation space dimension
          int step;               // row step for mean, var and obs (step>=width)
           int num;                 // observations number
           int step1;               // row step for gamma (step1>=num)
           float obs [num*step}     // observation vectors
   Update: float weight[height]     // Gaussian weights
           float mean[height*step]  // Gaussian mean vectors
           float var [height*step]  // Gaussian variance vectors
   Output: float result;            // probability logarithms sum */
{
   float gamma [height*step1] // gamma matrix
   float gammaT [height]      // gamma sums vector
   int k; float sum, sumGamma;
   /* adjust determinants for probability calculation */
   ippsLn_32f_I(weight,height);
   for (k=0; k<height; k++) {
      ippsSumLn_32f(var+k*step,width,&sum);
      weight[k]+=0.5f*(sum-width*log(2.0*3.1415926));
   }
```

Example 8-2 Weights, Means and Variances EM Re-Estimation

```
   /* invert variances for probability calculation */
      ippsDivCRev_32f_I(var,height*step);
      /* logarithm of weighted Gaussian probabilities */
      ippsLogGauss_32f_D2(obs,step,mean,var,width,gamma,num,weight[0]);
      ippsCopy_32f(gamma,probs,num);
      for (k=1; k<height; k++) {
         ippsLogGauss_32f_D2(obs,step,mean+k*step,var+k*step,width,
                        gamma+k*step1,num,weight[k]);
         ippsLogAdd_32f(gamma+k*step1,probs,num,ippAlgHintNone);
      }
     ippsSum_32f(probs,num,&result,ippAlgHintNone);
      /* gamma matrix and sum calculation */
      ippsExp_32f_I(gamma,height*step1);

       ippsSumRow_32f_D2(gamma,height,step1,gammaT,num);
      ippsSum_32f(gammaT,height,&sumGamma,ippAlgHintNone);
      /* weights update */
      ippsDivC_32f(gammaT,sumGamma,weight,height);
      /* means and variances update */
      for (k=0; k<height; k++) {
         ippsDivC_32f_I(gammaT[k],gamma+k*step1,num);
         ippsWeightedMeanVarColumn_32f_D2(obs,step,gamma+k*step1,num,
                           mean+k*step,var+k*step,width);
      }
}
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pWgt*, *pDstMean*, or *pDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

NormalizeColumn

Normalizes the matrix columns given the column means and variances.

Syntax

```
IppStatus ippsNormalizeColumn_16s_D2Sfs(Ipp16s* pSrcDst, int step, int height,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);
IppStatus ippsNormalizeColumn_16s_D2LSfs(Ipp16s** mSrcDst, int height, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);
IppStatus ippsNormalizeColumn_32f_D2(Ipp32f* pSrcDst, int step, int height,
    const Ipp32f* pMean, const Ipp32f* pVar, int width);
IppStatus ippsNormalizeColumn_32f_D2L(Ipp32f** mSrcDst, int height, const
    Ipp32f* pMean, const Ipp32f* pVar, int width);
```

Parameters

pSrcDst	Pointer to the input and output vector [*height***step*].
mSrcDst	Pointer to the input and output matrix [*height*][*width*].
pMean	Pointer to the column mean vector [*width*].
pVar	Pointer to the column variance vector [*width*].
width	Length of the mean and variance vectors.
step	Row step in the input and output vector *pSrcDst*.
height	Number of rows in the input and output matrix *mSrcDst*.

Description

The function ippsNormalizeColumn is declared in the ippsr.h file. This function normalizes the columns of the matrix *mSrcDst* as follows:

For functions with the D2 suffix,

$$pSrcDst[i \cdot step + j] = (pSrcDst[i \cdot step + j] - pMean[j]) \cdot pVar[j] ,$$

$$0 \le i < height, \ 0 \le j < width.$$

For functions with the D2L suffix,

$$mSrcDst[i][j] = (mSrcDst[i][j] - pMean[j]) \cdot pVar[j] ,$$

$$0 \le i < height, \ 0 \le j < width.$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst*, *mSrcDst*, *pMean*, or *pVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

NormalizeInRange

Normalizes and scales input vector elements.

Syntax

```
IppStatus ippsNormalizeInRange_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len,
    Ipp32f lowCut, Ipp32f highCut, Ipp8u range);
IppStatus ippsNormalizeInRange_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_16s_I(Ipp16s* pSrcDst, int len, Ipp32f lowCut,
    Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len,
    Ipp32f lowCut, Ipp32f highCut, Ipp8u range);
IppStatus ippsNormalizeInRange_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int
    len, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f lowCut, Ipp32f highCut, Ipp32f range);
IppStatus ippsNormalizeInRange_32f_I(Ipp32f* pSrcDst, int len, Ipp32f lowCut,
    Ipp32f highCut, Ipp32f range);
IppStatus ippsNormalizeInRangeMinMax_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int
    len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp8u
    range);
IppStatus ippsNormalizeInRangeMinMax_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s
    range);
IppStatus ippsNormalizeInRangeMinMax_16s_I(Ipp16s* pSrcDst, int len, Ipp16s
    valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
```

```
IppStatus ippsNormalizeInRangeMinMax_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int
    len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp8u
    range);
IppStatus ippsNormalizeInRangeMinMax_32f16s(const Ipp32f* pSrc, Ipp16s* pDst,
    int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s
    range);
IppStatus ippsNormalizeInRangeMinMax_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f
    range);
IppStatus ippsNormalizeInRangeMinMax_32f_I(Ipp32f* pSrcDst, int len, Ipp32f
    valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f range);
```

Parameters

pSrc	Pointer to the input array [*len*].
pSrcDst	Pointer to the input and output array (for the in-place operation) [*len*].
pDst	Pointer to the output array [*len*].
len	Number of elements in the input and output array.
lowCut	Lower cutoff value.
highCut	Higher cutoff value.
range	Upper bound of output data values (lower bound is 0).
valMin	Minimum value of input data.
valMax	Maximum value of input data.

Description

The function `ippsNormalizeInRange` is declared in the `ippsr.h` file. This function first normalizes the input vector elements to the range from 0 to 1 imposing lower and higher cutoffs ($0 \le lowCut < highCut \le 1$), and then scales them by the *range* value.

If we denote by x_k the input values (which are equal to $pSrc[k]$ for not-in-place functions and to $pSrcDst[k]$ for in-place functions), then the output values y_k (written to $pDst[k]$ for not-in-place functions and to $pSrcDst[k]$ for in-place functions) are calculated according to the following formula:

$$y_k = \max\left(0, \min\left(range, \frac{\frac{x_k - x_{min}}{x_{max} - x_{min}} - lowCut}{highCut - lowCut} \cdot range\right)\right), \quad k = 0, \ldots len - 1,$$

where

$x_{min} = \min_{i=0,..len-1} x_i$, $x_{max} = \max_{i=0,..len-1} x_i$ for `ippsNormalizeInRange` function

and

$x_{min} = valMin$, $x_{max} = valMax$ for `ippsNormalizeInRangeMinMax` function.

This function can be used to form spectrogram data.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsBadArgErr`	Indicates an error when the condition $0 \le lowCut < highCut \le 1$ is not met, or *range* is less than 0, or *valMin* is greater than *valMax*.
`ippStsInvZero`	Indicates a warning that $x_{min} = x_{max}$. All elements of the output vector are set to 0.

MeanVarAcc

Accumulates the estimates for the mean and variance re-estimation.

Syntax

```
IppStatus ippsMeanVarAcc_32f(Ipp32f const* pSrc, Ipp32f const* pSrcMean,
    Ipp32f* pDstMeanAcc, Ipp32f* pDstVarAcc, int len, Ipp32f val);
IppStatus ippsMeanVarAcc_64f(Ipp64f const* pSrc, Ipp64f const* pSrcMean,
    Ipp64f* pDstMeanAcc, Ipp64f* pDstVarAcc, int len, Ipp64f val);
```

Parameters

pSrc	Pointer to the observation vector [*len*].
pSrcMean	Pointer to the old mean vector [*len*].
pDstMeanAcc	Pointer to the mean accumulator [*len*].
pDstVarAcc	Pointer to the variance accumulator [*len*].
len	Length of the observation vector.

val	Constant value in the re-estimation.

Description

The function `ippsMeanVarAcc` is declared in the `ippsr.h` file. This function accumulates the estimates for the mean and variance re-estimation in the forward-backward algorithm. The calculation is as follows:

$$pDstMeanAcc[i] = pDstMeanAcc[i] + val \cdot (pSrc[i] - pSrcMean[i]) ,$$

$$pDstVarAcc[i] = pDstVarAcc[i] + val \cdot (pSrc[i] - pSrcMean[i])^2 ,$$

$$0 \le i < len .$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pSrcMean*, *pDstMeanAcc*, or *pDstVarAcc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

GaussianDist

Calculates the distance between two Gaussians.

Syntax

```
IppStatus ippsGaussianDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
    Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f wgt1,
    Ipp32f det1, Ipp32f wgt2, Ipp32f det2);
IppStatus ippsGaussianDist_64f(const Ipp64f* pMean1, const Ipp64f* pVar1, const
    Ipp64f* pMean2, const Ipp64f* pVar2, int len, Ipp64f* pResult, Ipp64f wgt1,
    Ipp64f det1, Ipp64f wgt2, Ipp64f det2);
```

Parameters

pMean1	Pointer to the mean vector of the first Gaussian [*len*].
pVar1	Pointer to the variance vector of the first Gaussian [*len*].
pMean2	Pointer to the mean vector of the second Gaussian [*len*].
pVar2	Pointer to the variance vector of the second Gaussian [*len*].
len	Length of the mean and variance vectors.

pResult	Pointer to the distance value.
wgt1	Weight of the first Gaussian.
det1	Determinant of the first Gaussian (in the logarithmic representation).
wgt2	Weight of the second Gaussian.
det2	Determinant of the second Gaussian (in the logarithmic representation).

Description

The function `ippsGaussianDist` is declared in the `ippsr.h` file. This function calculates the distance between two Gaussians as follows:

$$pResult[0] = (wgt1 \cdot det1) + (wgt2 \cdot det2) - (wgt1 + wgt2) \cdot (len \cdot \ln(2\pi) - \ln(V)) \quad ,$$

where

$$V = \prod_{i=0}^{len-1} \frac{W_1[i] + W_2[i] - (wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i])^2}{wgt1 + wgt2} \quad ,$$

and

$$W_1[i] = wgt1 \cdot (pVar1[i] + pMean1[i]^2) \quad ,$$

$$W_2[i] = wgt2 \cdot (pVar2[i] + pMean2[i]^2)$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, *pVar2*, or *pResult* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsLnZeroArg`	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to `–Inf` .
`ippStsLnNegArg`	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to `NaN`.

GaussianSplit

Splits a single Gaussian component into two with the same variance.

Syntax

```
IppStatus ippsGaussianSplit_32f(Ipp32f* pMean1, Ipp32f* pVar1, Ipp32f* pMean2,
    Ipp32f* pVar2, int len, Ipp32f val);
IppStatus ippsGaussianSplit_64f(Ipp64f* pMean1, Ipp64f* pVar1, Ipp64f* pMean2,
    Ipp64f* pVar2, int len, Ipp64f val);
```

Parameters

pMean1	Pointer to the input Gaussian mean vector, also the mean vector of the first Gaussian after splitting [*len*].
pVar1	Pointer to the input Gaussian variance vector, also the variance vector of the first Gaussian after splitting [*len*].
pMean2	Pointer to the mean vector of the second Gaussian after splitting [*len*].
pVar2	Pointer to the variance vector of the second Gaussian after splitting [*len*].
len	Length of the mean and variance vectors.
val	Variance perturbation value.

Description

The function `ippsGaussianSplit` is declared in the `ippsr.h` file. This function splits the Gaussian component into two Gaussian mixture components as follows:

$$pMean1[i] = pMean1[i] + val \cdot \sqrt{pVar1[i]} \ ,$$

$$pMean2[i] = pMean1[i] - val \cdot \sqrt{pVar1[i]} \ ,$$

$$pVar2[i] = pVar1[i] \ ,$$

$$0 \le i < len \ .$$

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, or *pVar2* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

GaussianMerge

Merges two Gaussian probability distribution functions.

Syntax

```
IppStatus ippsGaussianMerge_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
    const Ipp32f* pMean2, const Ipp32f* pVar2, Ipp32f* pDstMean, Ipp32f*
    pDstVar, int len, Ipp32f* pDstDet, Ipp32f wgt1, Ipp32f wgt2);
IppStatus ippsGaussianMerge_64f(const Ipp64f* pMean1, const Ipp64f* pVar1,
    const Ipp64f* pMean2, const Ipp64f* pVar2, Ipp64f* pDstMean, Ipp64f*
    pDstVar, int len, Ipp64f* pDstDet, Ipp64f wgt1, Ipp64f wgt2);
```

Parameters

pMean1	Pointer to the mean vector of the first Gaussian [*len*].
pVar1	Pointer to the variance vector of the first Gaussian [*len*].
pMean2	Pointer to the mean vector of the second Gaussian [*len*].
pVar2	Pointer to the variance vector of the second Gaussian [*len*].
len	Length of the mean and variance vectors.
pDstMean	Pointer to the mean vector of the merged Gaussian [*len*].
pDstVar	Pointer to the variance vector of the merged Gaussian [*len*].
pDstDet	Pointer to the determinant of the merged Gaussian.
wgt1	Weight of the first Gaussian.
wgt2	Weight of the second Gaussian.

Description

The function ippsGaussianMerge is declared in the ippsr.h file. This function merges two Gaussian probability distribution functions as follows:

$$pDstMean[i] = \frac{wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i]}{wgt1 + wgt2},$$

$$pDstVar[i] = \frac{wgt1^2 \cdot (pVar1[i] + pMean1[i]^2) + wgt2^2 \cdot (pVar2[i] + pMean2[i]^2)}{wgt1 + wgt2},$$

$$pDstDet[0] = len \cdot \ln(2\pi) - \sum_{i=0}^{len-1} \ln pDstVar[i],$$

$0 \le i < len$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, *pVar2*, *pDstMean*, *pDstVar*, or *pDstDet* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to –Inf .
ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to NaN.

Entropy

Calculates entropy of the input vector.

Syntax

```
IppStatus ippsEntropy_32f(const Ipp32f* pSrc, int len, Ipp32f* pResult);
IppStatus ippsEntropy_16s32s_Sfs(const Ipp16s* pSrc, int srcShiftVal, int len,
    Ipp32s* pResult, int scaleFactor);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pResult	Pointer to the destination entropy value.
len	Length of the input vector.
srcShiftVal	Refer to "Integer Scaling" in Chapter 2.

scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsEntropy` is declared in the `ippsr.h` file. This function calculates entropy of the input vector as given by:

$$pResult[0] = \sum_{i=0}^{len-1} pSrc[i] \cdot \log_2 pSrc[i]$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pResult* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsLnNegArg`	Indicates a warning that some input vector elements are less than 0. Operation execution is not aborted. For floating-point operations the destination value is set to `NaN`, and for integer operations it is set to 0.

SinC

Calculates sine divided by its argument.

Syntax

```
IppStatus ippsSinC_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinC_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinC_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinC_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSinC_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pSrcDst	Pointer to the input and destination vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.

Description

The function `ippsSinC` is declared in the `ippsr.h` file. This function calculates destination vector elements according to the formula:

$$pDst[k] = \frac{\sin(pSrc[k])}{pSrc[k]}, \quad k = 0,\ldots len - 1 .$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pSrcDst* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

ExpNegSqr

Calculates exponential of the squared argument taken with the inverted sign.

Syntax

```
IppStatus ippsExpNegSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExpNegSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNegSqr_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNegSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExpNegSqr_64f_I(Ipp64f* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the input vector [*len*].
pSrcDst	Pointer to the input and destination vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.

Description

The function `ippsExpNegSqr` is declared in the `ippsr.h` file. This function calculates destination vector elements according to the following formula:

$$pDst[k] = \exp(-pSrc[k]^2) \ , \ k = 0,\ldots len - 1 .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pSrcDst* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

BhatDist

Calculates the Bhattacharia distance between two Gaussians.

Syntax

```
IppStatus ippsBhatDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
    Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult);
IppStatus ippsBhatDist_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
    Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult);
IppStatus ippsBhatDistSLog_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
    Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f
    sumLog1, Ipp32f sumLog2);
IppStatus ippsBhatDistSLog_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1,
    const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult, Ipp32f
    sumLog1, Ipp32f sumLog2);
```

Parameters

pMean1	Pointer to the first mean vector [*len*].
pVar1	Pointer to the first variance vector [*len*].
pMean2	Pointer to the second mean vector [*len*].
pVar2	Pointer to the second variance vector [*len*].
pResult	Pointer to the result.
len	Length of the input mean and variance vectors.
sumLog1	Sum of the first Gaussian variance in the logarithmic representation.
sumLog2	Sum of the second Gaussian variance in the logarithmic representation.

Description

The functions `ippsBhatDist` and `ippsBhatDistSLog` are declared in the `ippsr.h` file.

The function `ippsBhatDist` calculates the Bhattacharia distance between two Gaussians as follows:

$$pResult[0] = \frac{1}{4}\sum_{i=0}^{len-1}\frac{(pMean1[i]-pMean2[i])^2}{pVar1[i]+pVar2[i]} + \\ + \frac{1}{2}\sum_{i=0}^{len-1}\left(\ln\left(\frac{pVar1[i]+pVar2[i]}{2}\right)-\frac{\ln(pVar1[i])+\ln(pVar2[i])}{2}\right)$$

The function `ippsBhatDistSLog` calculates the Bhattacharia distance between two Gaussians as follows:

$$pResult[0] = \frac{1}{4}\sum_{i=0}^{len-1}\frac{(pMean1[i]-pMean2[i])^2}{pVar1[i]+pVar2[i]} + \\ + \frac{1}{2}\sum_{i=0}^{len-1}\ln\left(\frac{pVar1[i]+pVar2[i]}{2}\right)-\frac{sumLog1+sumLog2}{4}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pMean1*, *pVar1*, *pMean2*, *pVar2*, or *pResult* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsLnZeroArg`	Indicates a warning that a zero value was detected in the input vector. The execution is not aborted. The result value is set to `–Inf` if there is no negative element in the vector.
`ippStsLnNegArg`	Indicates a warning that negative values were detected in the input vector. The execution is not aborted. The result value is set to `NaN`.

UpdateMean

Updates the mean vector in the EM training algorithm.

Syntax

```
IppStatus ippsUpdateMean_32f(const Ipp32f* pMeanAcc, Ipp32f* pMean, int len,
    Ipp32f meanOcc);
IppStatus ippsUpdateMean_64f(const Ipp64f* pMeanAcc, Ipp64f* pMean, int len,
    Ipp64f meanOcc);
```

Parameters

pMeanAcc	Pointer to the mean accumulator [*len*].
pMean	Pointer to the mean vector [*len*].
len	Length of the mean vector.
meanOcc	Occupation sum of the Gaussian mixture.

Description

The function `ippsUpdateMean` is declared in the `ippsr.h` file. This function calculates the updated mean vector in the EM (Expectation-Maximization) training algorithm as follows:

$$pMean[i] = pMean[i] + \frac{pMeanAcc[i]}{meanOcc}, \quad 0 \le i < len.$$

Note that if $meanOcc \le 0$, the mean vector *pMean* is not updated.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pMean* or *pMeanAcc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsZeroOcc`	Indicates a warning that *meanOcc* is equal to 0.
`ippStsNegOccErr`	Indicates an error when *meanOcc* is less than 0.

UpdateVar

Updates the variance vector in the EM training algorithm.

Syntax

```
IppStatus ippsUpdateVar_32f(const Ipp32f* pMeanAcc, const Ipp32f* pVarAcc,
    const Ipp32f* pVarFloor, Ipp32f* pVar, int len, Ipp32f meanOcc, Ipp32f
    varOcc);
IppStatus ippsUpdateVar_64f(const Ipp64f* pMeanAcc, const Ipp64f* pVarAcc,
    const Ipp64f* pVarFloor, Ipp64f* pVar, int len, Ipp64f meanOcc, Ipp64f
    varOcc);
```

Parameters

pMeanAcc	Pointer to the mean accumulator [*len*].
pVarAcc	Pointer to the variance accumulator [*len*].
pVarFloor	Pointer to the variance floor vector [*len*].
pVar	Pointer to the variance vector [*len*].
len	Length of the variance.
meanOcc	Occupation sum of the Gaussian mixture.
varOcc	Square occupation sum of the variance mixture.

Description

The function `ippsUpdateVar` is declared in the `ippsr.h` file. This function calculates the updated variance vector in the EM algorithm. The covariance matrix is assumed to be diagonal. The accumulators are calculated from the training data. The update equation is as follows:

$$pVar[i] = \max\left[varFloor[i],\ \frac{varAcc[i]}{varOcc} - \left(\frac{meanAcc[i]}{meanOcc}\right)^2\right],\ 0 \le i < len.$$

Note that if $meanOcc \le 0$ or $varOcc \le 0$, the variance vector $pVar$ is not updated.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pMeanAcc*, *pVarAcc*, *pVarFloor*, or *pVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning that *meanOcc* or *varOcc* is equal to 0.
ippStsNegOccErr	Indicates an error when *meanOcc* and *varOcc* are less than 0.
ippStsResFloor	Indicates a warning that all variances are floored.

UpdateWeight

Updates the weight values of Gaussian mixtures in the EM training algorithm.

Syntax

```
IppStatus ippsUpdateWeight_32f(const Ipp32f* pWgtAcc, Ipp32f* pWgt, int len,
    Ipp32f* pWgtSum, Ipp32f wgtOcc, Ipp32f wgtThresh);
IppStatus ippsUpdateWeight_64f(const Ipp64f* pWgtAcc, Ipp64f* pWgt, int len,
    Ipp64f* pWgtSum, Ipp64f wgtOcc, Ipp64f wgtThresh);
```

Parameters

pWgtAcc	Pointer to the weight accumulator [*len*].
pWgt	Pointer to the weight vector [*len*].
len	Number of Gaussian mixture components.
pWgtSum	Pointer to the output sum of weight values.
wgtOcc	Nominator of the weight update equation.
wgtThresh	Threshold for the weight values.

Description

The function ippsUpdateWeight is declared in the ippsr.h file. This function calculates the updated weight values for a Gaussian mixture. The accumulators are calculated from the training data. The update equation is as follows:

$$pWgt[i] = \max\left(\frac{pWgtAcc[i]}{wgtOcc}, wgtThresh\right), 0 \le i < len$$

$$pWgtSum[0] = \sum_{i=0}^{len-1} pWgt[i] .$$

Note that if *wgtOcc* ≤ 0 , the weight vector *pWgt* is not updated.

This function can also be used to update the HMM transition matrix.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pWgt*, *pWgtAcc*, or *pWgtSum* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning that *wgtOcc* is equal to 0.
ippStsNegOccErr	Indicates an error when *wgtOcc* is less than 0.
ippStsResFloor	Indicates a warning that all weights are floored.

UpdateGConst

Updates the fixed constant in the Gaussian output probability density function.

Syntax

```
IppStatus ippsUpdateGConst_32f(const Ipp32f* pVar, int len, Ipp32f* pDet);
IppStatus ippsUpdateGConst_64f(const Ipp64f* pVar, int len, Ipp64f* pDet);
IppStatus ippsUpdateGConst_DirectVar_32f(const Ipp32f* pVar, int len,
    Ipp32f* pDet);
IppStatus ippsUpdateGConst_DirectVar_64f(const Ipp64f* pVar, int len,
    Ipp64f* pDet);
```

Parameters

pVar	Pointer to the variance vector [*len*].
len	Dimension of the variance vector.
pDet	Pointer to the result value.

Description

The function `ippsUpdateGConst` is declared in the `ippsr.h` file. This function calculates the fixed variance constant.
For functions without the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be inverse diagonal:

$$pDet[0] = len \cdot \ln 2\pi - \sum_{i=o}^{len-1} \ln(pVar[i])$$

For functions with the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be diagonal:

$$pDet[0] = len \cdot \ln 2\pi + \sum_{i=o}^{len-1} \ln(pVar[i])$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pVar* or *pDet* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsLnZeroArg`	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to `–Inf` if there are no negative elements in the vector.
`ippStsLnNegArg`	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to `NaN`.

OutProbPreCalc

Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.

Syntax

```
IppStatus ippsOutProbPreCalc_32s(const Ipp32s* pWeight, const Ipp32s* pSrc,
    Ipp32s* pDst, int len);
```

```
IppStatus ippsOutProbPreCalc_32s_I(const Ipp32s* pWeight, Ipp32s* pSrcDst,
    int len);
IppStatus ippsOutProbPreCalc_32f(const Ipp32f* pWeight, const Ipp32f* pSrc,
    Ipp32f* pDst, int len);
IppStatus ippsOutProbPreCalc_64f(const Ipp64f* pWeight, const Ipp64f* pSrc,
    Ipp64f* pDst, int len);
IppStatus ippsOutProbPreCalc_32f_I(const Ipp32f* pWeight, Ipp32f* pSrcDst,
    int len);
IppStatus ippsOutProbPreCalc_64f_I(const Ipp64f* pWeight, Ipp64f* pSrcDst,
    int len);
```

Parameters

pWeight	Pointer to the Gaussian mixture weight vector [*len*].
pSrc	Pointer to the input vector calculated from `ippsUpdateGConst` function.
pSrcDst	Pointer to the input and output vector calculated from `ippsUpdateGConst` function.
pDst	Pointer to the result vector [*len*].
len	Number of mixtures in the HMM state.

Description

The function `ippsOutProbPreCalc` is declared in the `ippsr.h` file. This function pre-calculates the part of the Gaussian mixture output probability that is irrelevant to the observation vectors.

For the function `ippsOutProbPreCalc`,

$$pDst[i] = pWeight[i] - 0.5 \cdot pSrc[i], \quad 0 \le i < len.$$

For the function `ippsOutProbPreCalc_I`,

$$pSrcDst[i] = pWeight[i] - 0.5 \cdot pSrcDst[i], \quad 0 \le i < len.$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pWeight*, *pDet*, or *pVal* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when the *len* is less than or equal to zero.

DcsClustLAccumulate

Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.

Syntax

```
IppStatus ippsDcsClustLAccumulate_32f(const Ipp32f* pMean, const Ipp32f* pVar,
    Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);
IppStatus ippsDcsClustLAccumulate_64f(const Ipp64f* pMean, const Ipp64f* pVar,
    Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_32f(const Ipp32f* pMean, const
    Ipp32f* pVar, Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_64f(const Ipp64f* pMean, const
    Ipp64f* pVar, Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);
```

Parameters

pMean	Pointer to the mean vector of an HMM state in the cluster [*len*].
pVar	Pointer to the variance vector of an HMM state in the cluster [*len*].
pDstSum	Pointer to the summation part of the accumulator [*len*].
pDstSqr	Pointer to the square sum part of the accumulator [*len*].
len	Length of the mean and variance vectors.
occ	Occupation counts of the HMM state

Description

The function `ippsDcsClustLAccumulate` is declared in the `ippsr.h` file. This function updates the accumulators in the decision-tree clustering algorithm. The accumulators are used to calculate the likelihood of an HMM state cluster.

The accumulation equations are as follows:

For functions without the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be inverse diagonal, and

$$pDstSum[i] = pMean[i] \cdot occ \ ,$$

$$pDstSqr[i] = \left[\frac{1}{pVar[i]} + (pMean[i])^2\right] \cdot occ \ , \text{ for } 0 \le i < len.$$

For functions with the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be diagonal, and

$pDstSum[i] = pMean[i] \cdot occ$,

$pDstSqr[i] = [pVar[i] + (pMean[i])^2] \cdot occ$, for $0 \le i < len$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pMean*, *pVar*, *pDstSum*, or *pDstSqr* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

DcsClustLCompute

Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.

Syntax

```
IppStatus ippsDcsClustLCompute_64f(const Ipp64f* pSrcSum, const Ipp64f*
    pSrcSqr, int len, Ipp64f* pDst, Ipp64f occ);
IppStatus ippsDcsClustLCompute_32f64f(const Ipp32f* pSrcSum, const Ipp32f*
    pSrcSqr, int len, Ipp64f* pDst, Ipp32f occ);
```

Parameters

pSrcSum	Pointer to the summation part of the accumulator [*len*].
pSrcSqr	Pointer to the square sum part of the accumulator [*len*].
len	Length of the *pSrcSum* and *pSrcSqr* vectors.
pDst	Pointer to the result likelihood value.
occ	Occupation sum of the HMM state cluster.

Description

The function `ippsDcsClustLCompute` is declared in the `ippsr.h` file. This function calculates the likelihood of an HMM state cluster, according to the accumulators computed by `ippsDcsClustLAccumulate` function. The likelihood value is used to determine the splitting of a decision tree node in the decision-tree state-clustering algorithm. The calculation is as follows:

$$pDst[0] = -\frac{1}{2}occ \cdot \left\{ len \cdot [1 + \ln 2\pi - 2\ln(occ)] + \sum_{i=0}^{len-1} \ln[pSrcSqr[i] \cdot occ - (pSrcSum[i])^2] \right\}$$

Note that if $occ = 0$, $pDst[0]$ is set to `IPPLOGZERO`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrcSum*, *pSrcSqr* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0 or *occ* is less than or equal to 0.
`ippStsZeroOcc`	Indicates a warning that *occ* is equal to 0.
`ippStsNegOccErr`	Indicates an error when *occ* is less than 0.
`ippStsLnZeroArg`	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to `–Inf` if there are no negative elements in the vector.
`ippStsLnNegArg`	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to `NaN`.

Model Adaptation

This section describes functions that can be used to adapt the acoustic and language models. The adaptation algorithms adjust the parameters of existing models to match the characteristics set by the users, given a few learning samples.

AddMulColumn

Adds a weighted matrix column to the other column.

Syntax

```
IppStatus ippsAddMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height, int
    col1, int col2, int row1, const Ipp64f val);
```

Parameters

mSrcDst	Pointer to the source and destination matrix [*height*] [*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	Column number of the first operand.
col2	Column number of the second operand.
row1	Starting row number.
val	Weight factor.

Description

The function AddMulColumn is declared in the ippsr.h file. This function adds a matrix column weighted by *val* to the other matrix column.It is used for fast execution of the SVD algorithm.
The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] + mSrcDst[i][col1] \cdot val,$$

for $row1 \le i < height$.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *mSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, *col2*, or *row1* is less than or equal to 0; or *col1* or *col2* is greater than or equal to *width*; or *row1* is greater than or equal to *height*.

AddMulRow

Adds a weighted vector to the other vector.

Syntax

```
ippsAddMulRow_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
    const Ipp64f val);
```

pSrc	Pointer to the source vector [*len*].
pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the source and destination vectors.
val	Weight factor.

Description

The function AddMulRow is declared in the ippsr.h file. This function adds a vector weighted by *val* to the other vector. It is used for fast execution of the SVD algorithm.The calculation is as follows:

$$pSrcDst[i] = pSrcDst[i] + pSrc[i] \cdot val, \quad 0 \le i < len .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

QRTransColumn

Performs the QR transformation.

Syntax

```
IppStatus ippsQRTransColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
    int col1, int col2, const Ipp64f val1, const Ipp64f val2);
```

Parameters

mSrcDst	Pointer to the source matrix and destination matrix [*height*] [*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	First column number.
col2	Second column number.
val1	First weight factor.
val2	Second weight factor.

Description

The `ippsQRTransColumn` is declared in the `ippsr.h` file. This function performs the *QR* transform. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] \cdot val1 + mSrcDst[i][col1] \cdot val2 ,$$

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val1 + mSrcDst[i][col2] \cdot val2 ,$$

for $0 \le i < height$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *mSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *height*, *width*, *col1*, or *col2* is less than or equal to 0; or *col1* or *col2* is greater than or equal to *width*; or *col2* is greater than or equal to *height*.

DotProdColumn

Calculates the dot product of two matrix columns.

Syntax

```
IppStatus ippsDotProdColumn_64f_D2L(const Ipp64f** mSrc, int width,
    int height, Ipp64f* pSum, int col1, int col2, int row1);
```

Parameters

mSrc	Pointer to the source matrix [*height*] [*width*].
width	Number of columns in the matrix *mSrc*.
height	Number of rows in the matrix *mSrc*.
pSum	Pointer to the computed sum.
col1	First column number.
col2	Second column number.
row1	First row number.

Description

The function `ippsDotProdColumn` is declared in the `ippsr.h` file. This function calculates the dot product of two matrix columns. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$pSum[0] = \sum_{i=row1}^{height-1} mSrc[i][col1] \cdot mSrc[i][col2]$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *mSrc* or *pSum* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *height*, *width*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

MulColumn

Multiplies a matrix column by a value.

Syntax

```
IppStatus ippsMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
    int col1, int row1, const Ipp64f val);
```

Parameters

mSrcDst	Pointer to the source and destination matrix [*height*] [*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	First column number.
row1	First row number.
val	Weight factor.

Description

The function `ippsMulColumn` is declared in the `ippsr.h` file. This function multiplies a column of the *mSrcDst* matrix by *val*. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val, \quad row1 \le i < height .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *mSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumColumnAbs

Calculates the absolute sum of matrix column elements.

Syntax

```
IppStatus ippsSumColumnAbs_64f_D2L(const Ipp64f** mSrc, int width,
    int height, Ipp64f* pSum, int col1, int row1);
```

Parameters

mSrc	Pointer to the source matrix [*height*] [*width*].
width	Number of columns in the matrix *mSrc*.
height	Number of rows in the matrix *mSrc*.
pSum	Pointer to the computed sum.
col1	First column number.
row1	First row number.

Description

The function `ippsSumColumnAbs` is declared in the `ippsr.h` file. This function calculates the absolute sum of the matrix column elements. It is used for fast execution of the SVD algorithm.
The calculation is as follows:

$$pSum[0] = \sum_{i=row1}^{height-1} |mSrc[i][col1]|$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *mSrc* or *pSum* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *width*, *height*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumColumnSqr

Calculates the square sums of weighted matrix column elements.

Syntax

```
IppStatus ippsSumColumnSqr_64f_D2L(Ipp64f** mSrcDst, int width, int height,
    Ipp64f* pSum, int col1, int row1, const Ipp64f val);
```

Parameters

mSrcDst	Pointer to the source and destination matrix [*height*] [*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
pSum	Pointer to the value of the computed sum.
col1	First column number.
row1	Second row number.
val	Weight factor.

Description

The function `ippsSumColumnSqr` is declared in the `ippsr.h` file. This function multiplies the columns of the matrix *mSrcDst* by *val* and calculates the square sums of the column elements. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val, \quad 0 \le i < height$$

$$pSum[0] = \sum_{i=row1}^{height-1} (mSrcDst[i][col1])^2$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *mSrcDst* or *pSum* pointer is `NULL`.

ippStsSizeErr	Indicates an error when *width*, *height*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumRowAbs

Calculates the absolute sum of the vector elements.

Syntax

```
IppStatus ippsSumRowAbs_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
```

Parameters

pSrc	Pointer to the source vector [*len*].
pSum	Pointer to the value of the computed sum.
len	Length of the source vector *pSrc*.

Description

The function ippsSumRowAbs is declared in the ippsr.h file. This function calculates the absolute sum of the vector elements as follows:

$$pSum[0] = \sum_{i=0}^{len-1} |pSrc[i]|$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pSum* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

SumRowSqr

Calculates the square sum of weighted vector elements.

Syntax

```
IppStatus ippsSumRowSqr_64f(Ipp64f* pSrcDst, int len, Ipp64f* pSum,
    const Ipp64f val);
```

Parameters

pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the source vector *pSrcDst*.
pSum	Pointer to the value of the computed sum.
val	Weight factor.

Description

The function ippsSumRowSqr is declared in the ippsr.h file. This function multiplies the vector *pSrcDst* by *val* and calculates the square sum of the vector elements as follows:

$pSrcDst[i] = pSrcDst[i] \cdot val, \quad 0 \le i < len$

$$pSum[0] = \sum_{i=0}^{len-1} (pSrcDst[i])^2 .$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* or *pSum* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

SVD, SVDSort

Performs single value decomposition on a matrix.

Syntax

```
IppStatus ippsSVD_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int height,
    Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int height,
    Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVD_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW, Ipp64f*
    pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f* pDstW,
    Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVDSort_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int height,
    Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVDSort_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int height,
    Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVDSort_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW,
    Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVDSort_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f* pDstW,
    Ipp64f** mDstV, int width, int nIter);
```

Parameters

pSrcA	Pointer to the input vector *A* [*height***step*].
pDstU	Pointer to the output vector *U* [*height***step*].
pSrcDstA	Pointer to the input matrix *A* and output matrix *U* [*height***step*].
pDstV	Pointer to the output vector *V* [*width***step*].
mSrcA	Pointer to the input matrix *A* [*height*] [*width*].
mDstU	Pointer to the output matrix *U* [*height*] [*width*].
mSrcDstA	Pointer to the input matrix *A* and output matrix *U* [*height*] [*width*].
pDstW	Pointer to the output vector *W* [*width*].
mDstV	Pointer to the output matrix *V* [*width*] [*width*].
height	Number of rows in the input matrix.

width	Number of columns in the input matrix.
step	Row step in the vector *pSrcA*, *pSrcDstA*, or *pDstV*.
nIter	Number of iterations for diagonalization.

Description

The functions `ippsSVD` and `ippsSVDSort` are declared in the `ippsr.h` file. This function performs Single Value Decomposition (SVD) on the input matrix *A*.
The output matrices *U*, *W*, and *V* meet the following condition:

$$A = U \circ W \circ V^T ,$$

where the matrix *U* is column-orthogonal, the matrix *W* is diagonal (stored as a vector), and the matrix *V* is orthogonal.
V^T is the transpose of the matrix *V*.

The function `ippsSVDSort` transforms output matrices so that elements of the matrix *W* are sorted in descending order of their absolute values.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *height*, *width*, *step*, or *nIter* is less than or equal to 0, or *width* is greater than *step*.
`ippStsSVDCnvgErr`	Indicates an error when the SVD algorithm has not converged after *nIter* iterations.

WeightedSum

Calculates the weighted sums of two input vector elements.

Syntax

```
IppStatus ippsWeightedSum_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, Ipp32f weight1, Ipp32f weight2);
```

```
IppStatus ippsWeightedSum_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len, Ipp64f weight1, Ipp64f weight2);
IppStatus ippsWeightedSumHalf_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);
```

Parameters

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the input and output vectors.
weight1	First weight value.
weight2	Second weight value.

Description

The functions `ippsWeightedSum` and `ippsWeightedSumHalf` are declared in the `ippsr.h` file. The function `ippsWeightedSum` calculates the weighted sum as follows:

$$pDst[i] = \frac{weight1 \cdot pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, ..len-1 .$$

The function `ippsWeightedSumHalf` calculates the weighted sum as given by:

$$pDst[i] = \frac{pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, ..len-1$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1*, *pSrc2*, or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

ippStsDivByZero	Indicates a warning that a divisor vector element has zero value. The execution is not aborted. The value of the destination vector element for the floating-point operations is set as follows:
	NaN — For zero-valued dividend vector element;
	+Inf — For positive dividend vector element;
	-Inf — For negative dividend vector element.

Vector Quantization

This section describes some functions for vector quantization and codebook operations. These functions are commonly used in acoustic and language model compressions.

FormVector

Constructs an output vector of multiple streams from codebook entries.

Syntax

```
IppStatus ippsFormVector_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
    Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
    Ipp16s* pDst);
IppStatus ippsFormVector_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc, const
    Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
    Ipp16s* pDst);
IppStatus ippsFormVector_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
    Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
    Ipp32f*  pDst);
IppStatus ippsFormVector_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc, const
    Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
    Ipp32f*  pDst);

IppStatus ippsFormVector_2i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
    Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_2i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
    const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_2i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
    Ipp32s* pHeights, Ipp32f* pDst, int len);
IppStatus ippsFormVector_2i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
    const Ipp32s* pHeights, Ipp32f* pDst, int len);

IppStatus ippsFormVector_4i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
    Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_4i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
    const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_4i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
    Ipp32s* pHeights, Ipp32f* pDst, int len);
IppStatus ippsFormVector_4i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
    const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

Parameters

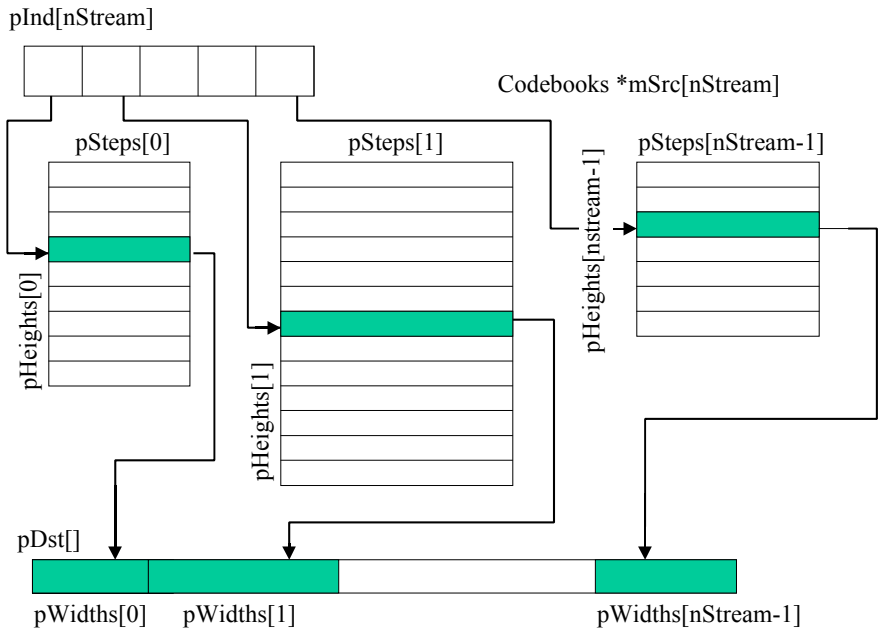
pInd	Pointer to the indexing vector [*nStream*].
mSrc	Pointer to the array of pointers to the codebooks [*nStream*].
pHeights	Pointer to the codebook lengths [*nStream*].
pWidths	Pointer to the stream lengths [*nStream*].
pSteps	Pointer to the codevector lengths [n*Stream*].
nStream	Number of codebooks.
pDst	Pointer to the output vector.
len	Length of the output vector.

Description

The function `ippsFormVector` is declared in the `ippsr.h` file. This function constructs an output vector of multiple streams. Each stream, of size *pWeights*[], is a codebook entry indexed by *pInd*[]. The codebooks are referenced by *mSrc*[], and have *pHeights*[] number of codevectors, each of which is of size *pSteps*[].

The Figure 8-5 illustrates the stream layout.

Figure 8-5 Stream Layout for the `ippsFormVector` Function



The output vector is obtained as follows:

$$pDst\left[i+\sum_{j=0}^{k-1} pWidths[j]\right] = mSrc[k][pInd[k]\cdot\ pSteps[k]+i]\ ,$$

$k = 0,\ldots nStream-1;\ \ i = 0,\ldots pWidths[k]-1$.

The function `ippsFormVector_2i` simplifies the extraction process by posting the constraints of $pWidths[] = pSteps[] = 2$ and $nStream = len/2$.

Similarly, the function `ippsFormVector_4i` implies the constraints of $pWidths[] = pSteps[] = 4$ and $nStream = len/4$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *pInd*[*k*] is less than 0, or when *len*, *nStream*, *pWidths*[*k*], or *pSteps*[*k*] is less than or equal to 0; or when *pHeights*[*k*] is less than or equal to *pInd*[*k*].

CdbkGetSize

Calculates the size in bytes of the codebook.

Syntax

```
IppStatus ippsCdbkGetSize_16s(int width, int step, int height, int cdbkSize,
    IppCdbk_Hint hint, int* pSize);
```

Parameters

width	Length of the input vectors (0 < *width* < 512).
step	Row step in the source vector pSrc (0 < *step* < 512).
height	Number of rows in the source vector pSrc (currently only *height* = *cdbkSize* is supported).
cdbkSize	Size of the codebook (0 < *cdbkSize* ≤ 8192).
hint	Flag indicating format of codebook. See ippsCdbkInit for a complete description.
pSize	Pointer to the variable to contain the size in bytes of the codebook structure and associated storage.

Description

The function ippsCdbkGetSize is declared in the ippsr.h file. This function calculates the size in bytes of the codebook and additional information to be used for fast search.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSize* pointer is NULL.

ippStsSizeErr	Indicates an error when *cdbkSize* is less than or equal to 0; or *cdbkSize* is greater than 8192; or *cdbkSize* is not equal to *height*; or when *width*, *step*, or *height* is less than or equal to 0; or *width* is greater than *step*.
ippStsCdbkFlagErr	Indicates an error when the *hint* value is incorrect or not supported.

NOTE. The only *hint* value that is currently supported for this function is IPP_CDBK_FULL. The *height* parameter is not needed in this mode and its value will be ignored.

CdbkInit

Initializes the structure that contains the codebook.

Syntax

```
IppStatus ippsCdbkInit_L2_16s(IppsCdbkState_16s* pCdbk, const Ipp16s* pSrc, int
    width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
```

Parameters

pSrc	Pointer to the source vector with *height***step* entries.
width	Length of the input vectors (0 < *width* < 512).
step	Row step in the source vector *pSrc* (0 < *step* < 512).
height	Number of rows in the source vector *pSrc* (0 < *height* ≤ *cdbkSize*).
cdbkSize	Size of the codebook (0 < *cdbkSize* ≤ 8192).
hint	One of the following values:
	IPP_CDBK_FULL The source data are entries of a codebook, *height* should be greater or equal to *nCluster*. The nearest codebook entry is located through a full search.

`IPP_CDBK_KMEANS_LONG`
: LBG algorithm with splitting of the most extensional cluster was used for the codebook building. The nearest codebook entry is located through a logarithmical search.

`IPP_CDBK_KMEANS_NUM`
: LBG algorithm with splitting of the most numerous clusters was used for the codebook building. The nearest codebook entry is located through a logarithmical search.

pCdbk Pointer to the codebook structure to be initialized.

Description

The function `ippsCdbkInit` is declared in the `ippsr.h` file. This function initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by the `ippsSplitVQ` function. The Euclidean distance is used to measure the similarity between two vectors.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCdbk* or *pSrc* pointers are null.
`ippStsSizeErr`	Indicates an error when *width*, *step*, *height*, or *cdbkSize* is less than or equal to 0; or width is greater than *step*; or *cdbkSize* is greater than 8192.
`ippStsCdbkFlagErr`	Indicates an error when the *hint* value is incorrect or not supported.

NOTE. The only *hint* value that is currently supported for this function is `IPP_CDBK_FULL`. State memory address stored in *pCdbk* must be aligned to 32-bit word boundary.

CdbkInitAlloc

Initializes the codebook structure.

Syntax

```
IppStatus ippsCdbkInitAlloc_L2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
    pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_L2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
    pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_WgtL2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
    pSrc, const Ipp16s* pWgt, int width, int step, int height, int cdbkSize,
    Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_WgtL2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
    pSrc, const Ipp32f* pWgt, int width, int step, int height, int cdbkSize,
    Ipp_Cdbk_Hint hint);
```

Parameters

pCdbk	Pointer to the codebook structure to be created.
pSrc	Pointer to the source vector [*height***step*].
pWgt	Pointer to the weight vector [width].
width	Length of the input vectors.
step	Row step in the source vector *pSrc*.
height	Number of rows in the source vector *pSrc*.
cdbkSize	Size of the codebook.
hint	One of the following values:

IPP_CDBK_FULL	The source data are entries of a codebook, *height* should be greater or equal to *cdbkSize*. The nearest codebook entry is located through a full search.
IPP_CDBK_KMEANS_LONG	LBG algorithm with splitting of the most extensional cluster is used for the codebook building. The nearest codebook entry is located through a logarithmical search.

IPP_CDBK_KMEANS_NUM	LBG algorithm with splitting of the most numerous clusters is used for the codebook building. The nearest codebook entry is located through a logarithmical search.

Description

The function ippsCdbkInitAlloc is declared in the ippsr.h file. This function initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by ippsVQ or ippsSplitVQ functions.
The ippsCdbkInitAlloc_L2 function uses the Euclidean distance to measure the similarity between two vectors, while the ippsCdbkInitAlloc_WgtL2 function uses the weighted Euclidean distance for these purposes.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *width*, *step*, or *cdbkSize* is less than or equal to 0; or *cdbkSize* is greater than *height*; or *width* is greater than *step*; or *cdbkSize* is greater than 16383.
ippStsCdbkFlagErr	Indicates an error when the *hint* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.
ippStsBadArgErr	Indicates an error when one of *pWgt*[i] is less than 0.
ippStsSmallerCodebook	Indicates a warning when size of the created codebook is less than the *cdbkSize* argument value.

CdbkFree

Destroys the codebook structure.

Syntax

```
IppStatus ippsCdbkFree_16s(IppsCdbkState_16s* pCdbk);
IppStatus ippsCdbkFree_32f(IppsCdbkState_32f* pCdbk);
```

Parameters

pCdbk	Pointer to the codebook structure.

Description

The function `ippsCdbkFree` is declared in the `ippsr.h` file. This function destroys the codebook structure and frees all memory associated with it.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCdbk* pointer is `NULL`.

GetCdbkSize

Retrieves the number of codevectors in the codebook.

Syntax

```
IppStatus ippsGetCdbkSize_16s(const IppsCdbkState_16s* pCdbk, int* pNum);
IppStatus ippsGetCdbkSize_32f(const IppsCdbkState_32f* pCdbk, int* pNum);
```

Parameters

pCdbk	Pointer to the codebook structure.
pNum	Pointer to the result number of codevectors.

Description

The function `ippsGetCdbkSize` is declared in the `ippsr.h` file. This function retrieves the number of codevectors in the codebook *pCdbk*. This number could be less than *cdbkSize* if the number of different vectors in *pSrc* is less than *cdbkSize*. The codebook structure *pCdbk* is initialized by the `ippsCdbkAlloc` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCdbk* or *pNum* pointer is `NULL`.

GetCodebook

Retrieves the codevectors from the codebook.

Syntax

```
IppStatus ippsGetCodebook_16s(const IppsCdbkState_16s* pCdbk, Ipp16s* pDst, int
    step);
IppStatus ippsGetCodebook_32f(const IppsCdbkState_32f* pCdbk, Ipp32f* pDst, int
    step);
```

Parameters

pCdbk	Pointer to the codebook structure.
pDst	Pointer to the destination vector for codevectors [*pNum*[0]**step*].
step	Row step in the destination vector *pDst*.

Description

The function `ippsGetCodebook` is declared in the `ippsr.h` file. This function retrieves the codevectors from the codebook structure *pCdbk* and stores them in the *pDst* vector with row step *step*.

The codebook structure *pCdbk* is initialized by the function `ippsCdbkAlloc`. The number of clusters can be obtained by the function `ippsGetCdbkSize`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCdbk* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *step* or equal to 0 or *step* is less than *width*.

VQ

Quantizes the input vectors given a codebook.

Syntax

```
IppStatus ippsVQ_16s(const Ipp16s* pSrc, int step, Ipp32s* pIndx, int height,
    const IppsCdbkState_16s* pCdbk);
IppStatus ippsVQ_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx, int height,
    const IppsCdbkState_32f* pCdbk);
IppStatus ippsVQDist_16s32s_Sfs(const Ipp16s* pSrc, int step, Ipp32s* pIndx,
    Ipp32s* pDist, int height, const IppsCdbkState_16s* pCdbk,
    int scaleFactor);
IppStatus ippsVQDist_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx,
    Ipp32f* pDist, int height, const IppsCdbkState_32f* pCdbk);
```

Parameters

pCdbk	Pointer to the codebook structure.
pSrc	Pointer to the source vector [*height***step*].
step	Row step in the source vector *pSrc*.
height	Number of rows in the source vector *pSrc*.
pIndx	Pointer to the result index vector of the closest codevectors [*height*].
pDist	Pointer to the result quantization distances from the source vector [*height*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The functions `ippsVQ` and `ippsVQDist` are declared in the `ippsr.h` file. The function `ippsVQ` performs Vector Quantization (VQ) on the input vectors. The resulting indexes of the closest codevectors are stored in the vector *pIndx*. The function `ippsVQDist` also stores the distances (scaled with *scaleFactor* for the integer versions) to the output distance vector *pDist*.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pCdbk*, *pSrc*, *pIndx*, or *pDist* pointer is NULL.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.

VQSingle_Sort, VQSingle_Thresh

Quantizes the input vector given a codebook and gets several closest clusters.

Syntax

```
IppStatus ippsVQSingle_Sort_32f(const Ipp32f* pSrc, Ipp32s* pIndx, const
    IppsCdbkState_32f* pCdbk, int num);
IppStatus ippsVQSingle_Sort_16s(const Ipp16s* pSrc, Ipp32s* pIndx, const
    IppsCdbkState_16s* pCdbk, int num);
IppStatus ippsVQDistSingle_Sort_32f(const Ipp32f* pSrc, Ipp32s* pIndx, Ipp32f*
    pDist, const IppsCdbkState_32f* pCdbk, int num);
IppStatus ippsVQDistSingle_Sort_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pIndx,
    Ipp32s* pDist, const IppsCdbkState_16s* pCdbk, int num, int scaleFactor);
IppStatus ippsVQSingle_Thresh_32f(const Ipp32f* pSrc, Ipp32s* pIndx, const
    IppsCdbkState_32f* pCdbk, Ipp32f val, int* pnum);
IppStatus ippsVQSingle_Thresh_16s(const Ipp16s* pSrc, Ipp32s* pIndx, const
    IppsCdbkState_16s* pCdbk, Ipp32f val, int* pnum);
IppStatus ippsVQDistSingle_Thresh_32f(const Ipp32f* pSrc, Ipp32s* pIndx,
    Ipp32f* pDist, const IppsCdbkState_32f* pCdbk, Ipp32f val, int* pnum);
IppStatus ippsVQDistSingle_Thresh_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pIndx,
    Ipp32s* pDist, const IppsCdbkState_16s* pCdbk, Ipp32f val, int* pnum, int
    scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
pIndx	Pointer to the destination indexes vector.
pDist	Pointer to the destination distances vector.
pCdbk	Pointer to the codebook structure.
val	Relative threshold value.

num	Number of closest clusters to search for each input vector.
pnum	Pointer to the number of clusters within threshold [1].
scaleFactor	Scaling factor for intermediate sums.

Discussion

The functions `ippsVQSingle_Sort` and `ippsVQSingle_Thresh` are declared in the `ippsr.h` file. These functions perform multiple vector quantization (VQ) for the input vector. Functions with `Sort` suffix provide *num* indexes of closest codebook centroids sorted in distance ascending order.

Functions with `Thresh` suffix provide indexes of codebook centroids with distance less than or equal to minimum multiplied by *val* and the number of such clusters. Functions with `Dist` suffix provide distance values of closest clusters. The length of vectors *pSrc* is equal to codevector length, the length of output vectors *pIndx* and *pDist* is equal to codebook size. Full search is done for all codebook types.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pIndx*, *pDist*, *pCdbk* or *pnum* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *num* is less than or equal to 0, or greater than the codebook size.
`ippStsBadArgErr`	Indicates an error when *val* is less than 1.

SplitVQ

Quantizes a multiple-stream vector given the codebooks.

Syntax

```
IppStatus ippsSplitVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s* pDst, int
    dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_16s8u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst, int
    dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_16s1u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst, int
    dstBitStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
```

```
IppStatus ippsSplitVQ_32f16s(const Ipp32f* pSrc, int srcStep, Ipp16s* pDst, int
    dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsSplitVQ_32f8u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst, int
    dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsSplitVQ _32f1u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst, int
    dstBitStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
```

Parameters

pCdbks	Pointer to the codebook structures [*nStream*].
pSrc	Pointer to the source vector [*height***srcStep*].
srcStep	Row step in the source vector *pSrc*.
pDst	Pointer to the destination indexing vector [*height***dstStep*].
dstStep	Row step in the destination vector *pDst*.
height	Number of rows in the source and destination vectors.
dstBitStep	Row step in the destination vector (in bits).
nStream	Number of streams in the source vectors.

Description

The functions `ippsSplitVQ` is declared in the `ippsr.h` file. This function quantizes the multiple-stream vectors *pSrc* against given codebooks *pDdbks*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors. The outputs *pDst* are indexes to the codebook entries.

For functions with the `1u` suffix, the output indexes are packed in bits. Each stream takes the least number of bits sufficient to represent its codebook indexes.

See also `ippsFormVector`, `ippsFormVectorVQ` functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pCdbk*, *pCdbk*[*k*], *pSrc*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *srcStep*, *dstStep*, *height*, or *nStream* is less than or equal to 0; or the sum of the stream length is greater than *srcStep;* or *nStream* is greater than *dstStep* for functions with the `16s` or `8u` suffix; or the number of bits sufficient to represent the indexes is greater

than *dstStep* for functions with the `1u` suffix;
or the codebook size is greater than 256 for functions with the `8u` suffix.

FormVectorVQ

Constructs multiple-stream vectors from codebooks, given indexes.

Syntax

```
IppStatus ippsFormVectorVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s*
    pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks, int
    nStream);
IppStatus ippsFormVectorVQ_8u16s(const Ipp8u* pSrc, int srcStep, Ipp16s* pDst,
    int dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsFormVectorVQ_1u16s(const Ipp8u* pSrc, int srcBitStep, Ipp16s*
    pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks, int
    nStream);
IppStatus ippsFormVectorVQ_16s32f(const Ipp16s* pSrc, int srcStep, Ipp32f*
    pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks, int
    nStream);
IppStatus ippsFormVectorVQ_8u32f(const Ipp8u* pSrc, int srcStep, Ipp32f* pDst,
    int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsFormVectorVQ_1u32f(const Ipp8u* pSrc, int srcBitStep, Ipp32f* pDst,
    int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
```

Parameters

pCdbks	Pointer to the codebook structures [*nStream*].
pSrc	Pointer to the indexing vectors [*height*srcStep*].
pDst	Pointer to the constructed vectors [*height*dstStep*].
srcStep	Row step in the indexing vectors *pSrc*.
srcBitStep	Row step in the indexing vectors *pSrc* (in bits).
dstStep	Row step in the constructed vectors *pDst*.
height	Number of rows in the vectors *pSrc*.
nStream	Number of streams.

Description

The function `ippsFormVectorVQ` is declared in the `ippsr.h` file. This function constructs multiple-stream vectors *pDst* from the codebooks *pCdbks* given indexes *pSrc*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors. For functions with the `1u` suffix, each stream index is assumed to be in a packed format. Each stream takes the number of bits sufficient to represent its codebook indexes. See also `ippsFormVector`, `ippsSplitVQ` functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pCdbk*, *pCdbk*[*k*], *pSrc*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *srcStep*, *dstStep*, *height*, or *nStream* is less than or equal to 0; or the codevector length sum is greater than *dstStep*; or *nStream* is greater than *srcStep* for functions with the `16s` or `8u` suffix; or the number of bits sufficient to represent the indexes is greater than *srcStep* for functions with the `1u` suffix.

Polyphase Resampling

The Intel IPP functions described in this section build, apply and free Kaiser-windowed polyphase filters for data resampling. Functions with `Fixed` suffix are intended for fixed rational resampling factor and provide faster speed. Functions without this suffix build universal resampling filter with linear interpolation of filter coefficients and allow to use a variable factor. The general description of the polyphase resampling algorithm can be found, for example, in *Multirate Digital Signal Processing* by R. Crochiere and L.Rabiner, Prentice Hall, 1983.

Some of these functions are used in the Intel IPP Speech recognition Sample. See *Speech Processing* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

ResamplePolyphaseInit

Initializes the structure for polyphase resampling without calculating the filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseFixedInit_16s( IppsResamplingPolyphaseFixed_16s*
    pSpec, int inRate, int outRate, int len, IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedInit_32f( IppsResamplingPolyphaseFixed_32f*
    pSpec, int inRate, int outRate, int len, IppHintAlgorithm hint);
```

Parameters

pSpec	The pointer to the resampling state structure.
inRate	The input rate for fixed factor resampling.
outRate	The output rate for fixed factor resampling.
len	The filter length for fixed factor resampling.
hint	Suggests using specific code (must be equal to `ippAlgHintFast`). The values for the *hint* argument are described in Table 7-3, "Flag and Hint Arguments".

Description

This function is declared in the `ippsr.h` file.
The function `ippsResamplePolyphaseFixedInit` initializes structures for data resampling with the factor equal to *inRate/outRate* without calculating the filter coefficients. This function is designed for using pre-calculated filter coefficients for polyphase resampling.

First, you need to allocate memory for the structure. Then use the function `ippsResamplePolyphaseSetFixedFilter` to import filter coefficients.
To calculate the memory size, filter length and the number of filters, use the `ippsResamplePolyphaseFixedGetSize` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSpec* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *inRate*, *outRate* or *len* is less than or equal to 0.
`ippStsBadArgErr`	Indicates that *hint* is not equal to `ippAlgHintFast`.

ResamplePolyphaseGetSize

Gets the size of the polyphase resampling structure.

Syntax

```
IppStatus ippsResamplePolyphaseFixedGetSize_16s(int inRate, int outRate, int
    len, int* pSize, int* pLen, int* pHeight, IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedGetSize_32f(int inRate, int outRate, int
    len, int* pSize, int* pLen, int* pHeight, IppHintAlgorithm hint);
```

Parameters

inRate	The input rate for fixed factor resampling.
outRate	The output rate for fixed factor resampling.
len	The filter length for fixed factor resampling.
pSize	The pointer to the variable that contains the size of the polyphase resampling structure.

pLen	The pointer to the variable that contains the real filter length.
pHeight	The pointer to the variable that contains the number of filters.
hint	Suggests using specific code (must be equal to `ippAlgHintFast`). The values for the *hint* argument are described in Table 7-3, "Flag and Hint Arguments".

Description

The function `ippsResamplePolyphaseFixedGetSize` is declared in the `ippsr.h` file. This function determines the size required for the fixed rate polyphase resampling structure and associated storage, the filter length and the number of filters in the filter bank. The returned length of the filter is equal to min{l≥len, l%4}, the filter length for zero phase is greater by 1. These values can be used for export and import of fixed polyphase resampling filter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSize*, *pLen*, or *pHeight* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *inRate*, *outRate* or *len* is less than or equal to 0.
`ippStsBadArgErr`	Indicates that *hint* is not equal to `ippAlgHintFast`.

ResamplePolyphaseSetFilter

Sets polyphase resampling filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseSetFixedFilter_16s(
    IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int step, int
    height);
IppStatus ippsResamplePolyphaseSetFixedFilter_32f(
    IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int step, int
    height);
```

Parameters

pSpec	The pointer to the resampling state structure.
pSrc	The pointer to the input vector of filter coefficients.

step	The row step in *pSrc* vector.
height	The number of filters (the number of rows in *pSrc* vector).

Description

The function `ippsResamplePolyphaseSetFixedFilter` is declared in the `ippsr.h` file. This function imports pre-calculated filter coefficients into the polyphase resampling structure. If the *step* value is less that the filter length, trailing filter coefficients are zeroed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSpec* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *step* or *height* is less than or equal to 0.
`ippStsBadArgErr`	Indicates that *height* is greater than the number of filters in *pSpec* structure.

ResamplePolyphaseGetFilter

Gets polyphase resampling filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseGetFixedFilter_16s(
    IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int step, int
    height);
IppStatus ippsResamplePolyphaseGetFixedFilter_32f(
    IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int step, int
    height);
```

Parameters

pSpec	The pointer to the resampling state structure.
pSrc	The pointer to the output vector of filter coefficients.
step	The row step in *pSrc* vector.
height	The number of filters (the number of rows in *pSrc* vector).

Description

The function `ippsResamplePolyphaseSetFixedFilter` is declared in the `ippsr.h` file. This function exports filter coefficients from the polyphase resampling structure. If the *step* value is less that the filter length, only first *step* coefficients are exported.

Example 8-3 Export and import of the polyphase resampling filter bank

```
int inRate=16000; // input frequency
int outRate=8000; // output frequency
int history; // half of filter length
char fname[]="filter.flt\0"; // coefficient file name
{
int size,len,height;
FILE *file;
short *pFilter;
IppsresamplingPolyphaseFixed_16s *state;
history=(int)(64.0f*0.5*IPP_MAX(1.0,1.0/(double)outRate/(double)inRate))+1;
ippsResamplePolyphaseFixedInitAlloc_16s(&state,inRate,outRate,2*(history-1),
0.95f,9.0f,ippAlgHintFast);
ippsResamplePolyphaseFixedGetSize_16s(inRate,outRate,2*(history-1),&size,&len,
&height,ippAlgHintFast);
pFilter=ippsMalloc_16s(len*height);
ippsResamplePolyphaseGetFixedFilter_16s(state,pFilter,len,height);
file=fopen(fname,"wb");
fwrite(&size,sizeof(int),1,file);
fwrite(&len,sizeof(int),1,file);
fwrite(&height,sizeof(int),1,file);
fwrite(pFilter,sizeof(short),len*height,file);
fclose(file);
ippsFree(pFilter);
ippsResamplePolyphaseFixedFree_16s(state);
}
{
```

```
int size,len,height;
FILE *file;
short *pFilter;
IppsresamplingPolyphaseFixed_16s *state;
history=(int)(64.0f*0.5*IPP_MAX(1.0,1.0/(double)outRate/(double)inRate))+1;
file=fopen(fname,"rb");
fread(&size,sizeof(int),1,file);
fread(&len,sizeof(int),1,file);
fread(&height,sizeof(int),1,file);
pFilter=ippsMalloc_16s(len*height);
fread(pFilter,sizeof(short),len*height,file);
fclose(file);
state=(IppsresamplingPolyphaseFixed_16s*)ippsMalloc_8u(size);
ippsResamplePolyphaseFixedInit_16s(state,inRate,outRate,2*(history-1),
ippAlgHintFast);
ippsResamplePolyphaseSetFixedFilter_16s((IppsresamplingPolyphaseFixed_16s*)
state, (const Ipp16s*) pFilter, len, height);
ippsFree(pFilter);
// use of polyphase filter
…
ippsFree(state);
}
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSpec* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.
ippStsBadArgErr	Indicates that *height* is greater than the number of filters in *pSpec* structure.

ResamplePolyphaseInitAlloc

Initializes the structure for polyphase data resampling.

Syntax

```
IppStatus ippsResamplePolyphaseInitAlloc_16s(IppsResamplingPolyphase_16s**
    pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha,
    IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseInitAlloc_32f(IppsResamplingPolyphase_32f**
    pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha,
    IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedInitAlloc_16s(
    IppsResamplingPolyphaseFixed_16s** pSpec, int inRate, int outRate, int len,
    Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedInitAlloc_32f(
    IppsResamplingPolyphaseFixed_32f** pSpec, int inRate, int outRate, int len,
    Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm hint);
```

Parameters

pSpec	The pointer to the resampling state structure to be created.
window	The size of the ideal lowpass filter window.
nStep	The discretization step for filter coefficients.
rollf	The roll-off frequency of the filter.
alpha	The parameter of the Kaiser window.
inRate	The input rate for resampling with fixed factor.
outRate	The output rate for resampling with fixed factor.
len	The filter length for resampling with fixed factor.
hint	Suggests using specific code. The values for the *hint* argument are described in Table 7-3, "Flag and Hint Arguments".

Description

These functions are declared in the `ippsr.h` file.

The functions `ippsResamplePolyphaseInitAlloc` create structures for data resampling using the ideal lowpass filter

$$h[i] = \begin{cases} rollf, & i = 0 \\ \dfrac{\sin(rollf \cdot i\pi / nStep)}{i\pi / nStep}, & i \neq 0 \end{cases}, \quad |i| \le window \cdot nStep,$$

$$0 < rollf \le 1$$

and applies the Kaiser window with *alpha* parameter and width *window* to it.

This means that the values of the ideal lowpass filtering function are calculated (discretized) for all *i* such that |*i* /*nStep*| ≤*window*.

The structure created could be used to resample input samples by the function `ippsResample` with arbitrary resampling factor. In this case, filter coefficients for each output sample are calculated on the fly by using linear interpolation between two nearest values. The filter size depends on the resampling factor.

The functions `ippsResamplePolyphaseFixedInitAlloc` create structures for data resampling with the factor equal to *inRate/outRate*. If we denote the number of filters created in the `IppsResamplingPolyphaseFixed` structure for input and output frequencies by *fnum*, then

$$fnum = outRate / GCD(inRate, outRate)$$

where GCD(*a,b*) stands for the greatest common divisor of *a* and *b*.
For example, if *inRate* = 8000 and *outRate* = 11025, then the number of filters will be *fnum* = 11025/GCD(8000, 11025) = 441.

Values *window* and *nStep* in the formula above are chosen to provide the length *flen* of each filter

$$flen = \min_{l \ge len,\, l\%4=0} l$$

Note that for zero phase, the filter length is *flen*+1.

Functions with the "`Fixed`" suffix in their name pre-calculate filter coefficients for each phase and store them in the data structure for better performance. These functions can be used for cases when the ratio *inRate/outRate* is rational only. They can be considerably faster but may require large data structures for some input and output rates.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSpec* pointer is `NULL`.

ippStsSizeErr	*For* `ippsResamplePolyphaseInitAlloc` *functions:* Indicates an error when *inRate*, *outRate*, *nStep* or *len* is less than or equal to 0. *For* `ippsResamplePolyphaseFixedInitAlloc` *functions:* Indicates an error when *inRate*, *outRate*, *nStep* or *len* is less than or equal to 0, or when *inRate*/LCD*(*outRate*+1)/LCD is greater than 2^{29} (here LCD stands for the lowest common denominator of *inRate* and *outRate*) .
ippStsBadArgErr	Indicates an error when *rollf* is less than or equal to 0 or is greater than 1, or if *alpha* is less than 1, or if *window* is less than 2/*nStep*.
ippStsMemAllocErr	Indicates a memory allocation error.

ResamplePolyphaseFree

Frees structure for polyphase data resampling.

Syntax

```
IppStatus ippsResamplePolyphaseFree_16s(IppsResamplingPolyphase_16s* pSpec);
IppStatus ippsResamplePolyphaseFree_32f(IppsResamplingPolyphase_32f* pSpec);
IppStatus ippsResamplePolyphaseFixedFree_16s(IppsResamplingPolyphaseFixed_16s*
    pSpec);
IppStatus ippsResamplePolyphaseFixedFree_32f(IppsResamplingPolyphaseFixed_32f*
    pSpec);
```

Parameters

pSpec	The pointer to the resampling state structure.

Description

The functions `ippsResamplePolyphaseFree` and `ippsResamplePolyphaseFixedFree` are declared in the `ippsr.h` file. These functions close the resampling structure by freeing all memory allocated for it by `ippsResamplePolyphaseInitAlloc` or `ippsResamplePolyphaseFixedInitAlloc` functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSpec* pointer is NULL.

ResamplePolyphase

Resamples input data using polyphase filters.

Syntax

```
IppStatus ippsResamplePolyphase_16s(const IppsResamplingPolyphase_16s* pSpec,
    const Ipp16s* pSrc, int len, Ipp16s* pDst, Ipp64f factor, Ipp32f norm,
    Ipp64f* pTime, int* pOutlen);
IppStatus ippsResamplePolyphase_32f(const IppsResamplingPolyphase_32f* pSpec,
    const Ipp32f* pSrc, int len, Ipp32f* pDst, Ipp64f factor, Ipp32f norm,
    Ipp64f* pTime, int* pOutlen);
IppStatus ippsResamplePolyphaseFixed_16s(const
    IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int len,
    Ipp16s* pDst, Ipp32f norm, Ipp64f* pTime, int* pOutlen);
IppStatus ippsResamplePolyphaseFixed_32f(const
    IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int len,
    Ipp32f* pDst, Ipp32f norm, Ipp64f* pTime, int* pOutlen);
```

Parameters

pSpec	The pointer to the resampling state structure.
pSrc	The pointer to the input vector.
pDst	The pointer to the output vector.
len	The number of input vector elements to resample.
norm	The norm factor for output samples.
factor	The resampling factor.
pTime	The pointer to the start time of resampling (in input vector elements). Keeps the input sample number and the phase for the first output sample from the next input data portion.
pOutlen	The number of calculated output vector elements.

Description

The functions `ippsResamplePolyphase` and `ippsResamplePolyphaseFixed` are declared in the `ippsr.h` file. These functions convert data from the input vector changing their frequency and compute all output samples that can be correctly calculated for the given input and the filter length. The ratio of output and input frequencies is defined by the *factor* argument for the function `ippsResamplePolyphase`. For the function `ippsResamplePolyphaseFixed`, this ratio is defined during creation of the resampling structure. The value *pTime*[0] defines the time value for which the first output sample is calculated.

The history data of filters are in the input vector with indexes less than *pTime*[0]. The history length is equal to *flen*/2 for function `ippsResamplePolyphaseFixed`, and

$$\left[\frac{1}{2} window \cdot \max(1, 1/factor)\right] + 1$$

for function `ippsResamplePolyphase`. Here *flen* is filter length and *window* is the size of the ideal lowpass filter window (see `ippsResamplePolyphaseInitAlloc`). The input vector should contain the same number of elements with indexes greater than *pTime*[0]+*len* for the right filter wing for the last element.

After function execution, the time value is updated and *pOutlen*[0] contains the number of calculated output samples.

The output samples are multiplied by $norm \cdot \min(1, factor)$ before saturation.

Example 8-4 Resampling of the input mono pcm file

```
void resampleIPP(
   int      inRate,     // input frequency
   int      outRate,    // output frequency
   FILE    *infd,       // input pcm file
   FILE    *outfd)      // output pcm file
{
   short *inBuf,*outBuf;
   int bufsize=4096;
   int history=128;
   double time=history;
   int lastread=history;
   int inCount=0,outCount=0,inLen,outLen;
```

```
   IppsResamplingPolyphaseFixed_16s *state;

   ippsResamplePolyphaseFixedInitAlloc_16s(&state,inRate,outRate,2*history,
                                0.95f,9.0f,ippAlgHintAccurate);
   inBuf=ippsMalloc_16s(bufsize+history+2);
   outBuf=ippsMalloc_16s((int)((bufsize-history)*outRate/(float)inRate+2));

   ippsZero_16s(inBuf,history);
   while ((inLen=fread(inBuf+lastread,sizeof(short),bufsize-lastread,infd))>0)
{
      inCount+=inLen;
      lastread+=inLen;
      ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-history-(int)time,
                        outBuf,0.98f,&time,&outLen);
      fwrite(outBuf,outLen,sizeof(short),outfd);
      outCount+=outLen;
      ippsMove_16s(inBuf+(int)time-history,inBuf,lastread+history-(int)time);
      lastread-=(int)time-history;
      time-=(int)time-history;
   }
   ippsZero_16s(inBuf+lastread,history);
   ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-(int)time,
                      outBuf,0.98f,&time,&outLen);
   fwrite(outBuf,outLen,sizeof(short),outfd);
   outCount+=outLen;

   printf("%d inputs resampled to %d outputs\n",inCount,outCount);
   ippsFree(outBuf);
   ippsFree(inBuf);
   ippsResamplePolyphaseFixedFree_16s(state);
}
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSpec*, *pSrc*, *pDst*, *pTime* or *pOutlen* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

ippStsBadArgErr	Indicates an error when *factor* is less than or equal to 0.

Advanced Aurora Functions

The Intel IPP functions discussed later in this section are implemented to support the speech processing and compression algorithm described in ETSI ES 202 050 V1.1.1 standard (see [ES202]).

The use of these functions is demonstrated in Intel IPP Speech Recognition Samples. See *Advanced Aurora Encoder-Decoder Sample* downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

SmoothedPowerSpectrumAurora

Calculates smoothed magnitude of the FFT output.

Syntax

```
IppStatus ippsSmoothedPowerSpectrumAurora_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len);
IppStatus ippsSmoothedPowerSpectrumAurora_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len);
IppStatus ippsSmoothedPowerSpectrumAurora_32s64s_Sfs(Ipp32s* pSrc, Ipp64s*
    pDst, Ipp32s len, int scaleFactor)
IppStatus  ippsSmoothedPowerSpectrumAurora_32s_Sfs(Ipp32s* pSrc, Ipp32s* pDst,
    Ipp32s len, int scaleFactor)
```

Parameters

pSrc	Pointer to the input vector in PERM format [*len*].
pDst	Pointer to the output vector [*len*/4+1].
len	The input vector length (multiple of 4).
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsSmoothedPowerSpectrumAurora` is declared in the `ippsr.h` file. This function calculates the smoothed square magnitude for the Fast Fourier Transform (FFT) output vector in PERM format ([ES202] , 5.1.3-5.1.4):

$$pDst[0] = \left(pSrc[0]^2 + pSrc[2]^2 + pSrc[3]^2\right)/2$$

$$pDst[i] = \left(pSrc[4 \cdot i]^2 + pSrc[4 \cdot i + 1]^2 + pSrc[4 \cdot i + 2]^2 + pSrc[4 \cdot i + 3]^2\right)/2$$

$$pDst[len/4] = pSrc[1]^2$$

where 0 < *i* < *len*/4 .

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0, or *len* is not a multiple of 4.

NoiseSpectrumUpdate_Aurora

Updates the noise spectrum.

Syntax

```
IppStatus ippsNoiseSpectrumUpdate_Aurora_32f(const Ipp32f* pSrc, const Ipp32f*
    pSrcNoise,  Ipp32f* pDst, int len);
IppStatus  ippsNoiseSpectrumUpdate_Aurora_16s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pSrcNoise, Ipp16s* pDst, int len, int scaleFactor);
IppStatus  ippsNoiseSpectrumUpdate_Aurora_32s_Sfs(const Ipp32s* pSrc, const
    Ipp32s* pSrcNoise, Ipp32s* pDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the input vector of the power spectral density mean [*len*].

pSrcNoise	Pointer to the input vector of the previous noiseless signal spectrum [*len*].
pDst	Pointer to the output vector of the improved transfer function [*len*].
len	The length of input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function `ippsNoiseSpectrumUpdate` is declared in the `ippsr.h` file. This function updates the noise spectrum estimate on the second stage of noise reduction according to the formula ([ES202], 5.1.5):

$$pDst[i] = pSrcNoise[i] \cdot \left(0.9 + \frac{0.1 \cdot pSrc[i]}{pSrc[i] + pSrcNoise[i]} \cdot \left(1 + \frac{pSrcNoise[i]}{pSrcNoise[i] + 0.1 \cdot pSrc[i]} \right) \right)$$

for $0 \le i < len$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pSrcNoise* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

WienerFilterDesign_Aurora

Calculates an improved transfer function of the adaptive Wiener filter.

Syntax

```
IppStatus ippsWienerFilterDesign_Aurora_32f(const Ipp32f* pSrc, const Ipp32f*
    pNoise, const Ipp32f* pDen, Ipp32f* pDst, int len);
IppStatus ippsWienerFilterDesign_Aurora_16s(const Ipp16s* pSrc, const Ipp16s*
    pNoise, const Ipp16s* pDen, Ipp16s* pDst, int len);
```

Parameters

pSrc	Pointer to the input vector of square roots of the power spectral density mean [*len*].
pNoise	Pointer to the input vector of square roots of the noise spectrum estimate [*len*].
pDen	Pointer to the input vector of square roots of the previous noiseless signal spectrum [*len*].
pDst	Pointer to the output vector of the improved transfer function [*len*] (in Q14).
len	The length of input and output vectors.

Description

The function `ippsWienerFilterDesign_Aurora` is declared in the `ippsr.h` file. This function processes the input vector according to the following formulas ([ES202], 5.1.5):

1. The noiseless signal is estimated:

$$p[i] = 0.98 \cdot pDen[i] + 0.02 \cdot \max(pSrc[i] - pNoise[i], 0)$$

2. The a priori Signal to Noise Ratio (SNR) is computed:

$$\eta[i] = p[i] / pNoise[i]$$

3. Then the filter transfer function

$$H[i] = \frac{\eta[i]}{1 + \eta[i]}$$

is used to improve the estimation of the noiseless signal spectrum:

$$p2[i] = H[i] \cdot pSrc[i]$$

4. The improved a priori SNR is calculated as:

$$\eta 2[i] = \max\left(p2[i] / pNoise[i], 0.079432823\right)$$

5. The improved filter transfer function is obtained as:

$$pDst[i] = \frac{\eta 2[i]}{1 + \eta 2[i]}$$

for $0 \le i < len$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pNoise*, *pDen*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsBadArg	Indicates an error when a negative or NaN argument of a square root operation is detected.

MelFBankInitAlloc_Aurora

Initializes the structure for performing the Mel-frequency filter bank analysis.

Syntax

```
IppStatus ippsMelFBankInitAllocLow_Aurora_16s(IppsFBankState_16s** pFBank);
IppStatus ippsMelFBankInitAllocLow_Aurora_32f(IppsFBankState_32f** pFBank);
IppStatus ippsMelFBankInitAllocHigh_Aurora_16s(IppsFBankState_16s** pFBank);
IppStatus ippsMelFBankInitAllocHigh_Aurora_32f(IppsFBankState_32f** pFBank);
```

Parameters

pFBank	Pointer to the Mel-scale filter bank structure to be created.

Description

The function ippsMelFBankInitAlloc_Aurora is declared in the ippsr.h file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis. The function with Low suffix builds a filter bank of 23 filters for 8 KHz data processing. The function with High suffix builds a filter bank of 3 filters for high frequency band of 16 KHz data processing ([ES202], 5.1.7).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* pointer is NULL.

TabsCalculation_Aurora

Calculates filter coefficients for residual filter.

Syntax

```
IppStatus ippsTabsCalculation_Aurora_16s(const Ipp16s* pSrc, Ipp16s* pDst);
IppStatus ippsTabsCalculation_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst);
```

Parameters

pSrc	Pointer to the input vector of Mel-scaled filter bank output [10].
pDst	Pointer to the output filter coefficients vector [17].

Description

The function ippsTabsCalculation_Aurora is declared in the ippsr.h file. This function calculates residual filter coefficients for the Mel-scaled filter bank output ([ES202], 5.1.9-5.1.10).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is not equal to 23 or 26.

ResidualFilter_Aurora

Calculates a denoised waveform signal.

Syntax

```
IppStatus ippsResidualFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    Ipp16s* pTabs, int scaleFactor);
IppStatus ippsResidualFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    Ipp32f* pTabs);
```

Parameters

pTabs	Pointer to the filter coefficients vector [17].

pSrc	Pointer to the input vector [96].
pDst	Pointer to the output vector [80].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsResidualFilter_Aurora is declared in the ippsr.h file.
This function filters the input vector to get the low band and high band parts ([ES202], 5.1.10):

$$pDst[j] = \sum_{i=0}^{16} pTabs[i] \cdot pSrc[i+j] \quad , \qquad 0 \le j < 80$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pTabs* pointer is NULL.

WaveProcessing_Aurora

Processes waveform data after noise reduction.

Syntax

```
IppStatus ippsWaveProcessing_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst);
IppStatus ippsWaveProcessing_Aurora_16s(const Ipp16s* pSrc, Ipp16s* pDst);
```

Parameters

pSrc	Pointer to the input vector [200].
pDst	Pointer to the output vector [200].

Description

The function ippsWaveProcessing_Aurora is declared in the ippsr.h file. This function processes the input vector according to the following formulas ([ES202], 5.2):

1. Smoothed Teager operator is calculated:

$$s[i] = \frac{1}{9}\sum_{k=-4}^{4} s_T[i] \quad , \qquad 0 \le i < 200$$

where

$$s_T[i] = \left| pSrc[i]^2 - pSrc[\max(0, i-1)] \cdot pSrc[\min(199, i+1)] \right|$$

for $0 \le i < 200$,

and $s[i] = 0$ for $i < 0$ and $200 \le i$.

2. Peaks

$$p[l_{\min}], \ldots p[l_{\max}]$$

are found:

$$p[0] = \underset{0 \le i < 200}{\arg\max}\, s[i]$$

$$p[l+1] = \underset{\substack{p[l]+25 \le k < p[l]+80 \\ k < 200}}{\arg\max}\, s[k] \quad , \qquad l = 0, \ldots l_{\max} - 1$$

$$p[l-1] = \underset{\substack{p[l]-25 \ge k > p[l]-80 \\ k \ge 0}}{\arg\max}\, s[k] \quad , \qquad l = l_{\min} + 1, \ldots 0$$

3. Weighting function is calculated:

$$w[i] = \begin{cases} 1.2, & p[k] - 4 < i < p[k] + 0.8 \cdot \left]p[k+1] - p[k]\right[\\ 1.0, & i = p[k] - 4, i = p[k] + 0.8 \cdot \left]p[k+1] - p[k]\right[\\ 0.8 & \text{otherwise} \end{cases}$$

$l_{\min} \le k < l_{\max}$

4. Output vector is computed as:

$$pDst[i] = w[i] \cdot pSrc[i] \quad , \qquad 0 \le i < 200$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.

LowHighFilter_Aurora

Calculates low band and high band filters.

Syntax

```
IppStatus ippsLowHighFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDstLow,
    Ipp16s* pDstHigh, int len, const Ipp16s* pTabs, int tapsLen, int
    scaleFactor);
IppStatus ippsLowHighFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh, int len, const Ipp32f* pTabs, int tapsLen);
```

Parameters

pSrc	Pointer to the input vector [*len*+*tapsLen*-1].
pDstLow	Pointer to the low frequency band output vector [*len*/2].
pDstHigh	Pointer to the high frequency band output vector [*len*/2].
len	The input samples number (even).
pTabs	Pointer to the filter coefficients vector [*tapsLen*].
tapsLen	The filter taps number (even).
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsLowHighFilter_Aurora is declared in the ippsr.h file. This function filters the input vector to get the low band and high band parts ([ES202], 5.5.1):

$$pDstLow[j] = \sum_{i=0}^{tapslen-1} pTabs[i] \cdot pSrc[i + 2 \cdot j]$$

$$pDstHigh[j] = \sum_{i=0}^{tapslen-1} pTabs[i] \cdot pSrc[i + 2 \cdot j] \cdot (-1)^{i+j} \quad , \qquad 0 \le j < len/2$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDstLow*, *pDstHigh*, or *pTabs* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or *tapsLen* is less than or equal to 0, or *len* or *tapsLen* is not even.

HighBandCoding_Aurora

Codes and decodes the high frequency band energy values.

Syntax

```
IppStatus ippsHighBandCoding_Aurora_32f(const Ipp32f* pSrcHFB, const Ipp32f*
    pInSWP, const Ipp32f* pDSWP, Ipp32f* pDstHFB);
IppStatus ippsHighBandCoding_Aurora_32s_Sfs(const Ipp32s* pSrcHFB, const
    Ipp32s* pInSWP, const Ipp32s* pDSWP, Ipp32s* pDstHFB, int scaleFactor)
```

Parameters

pSrcHFB	Pointer to the input high frequency band energy vector [3].
pInSWP	Pointer to the input signal smoothed power spectrum vector [65].
pDSWP	Pointer to the denoised signal power spectrum vector [129].
pDstHFB	Pointer to the output coded high frequency band log energy vector [3].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The function ippsHighBandCoding_Aurora is declared in the ippsr.h file. This function performs coding and decoding of high frequency band energies according to the following formulas ([ES202], 5.5.3):

1. Logarithm operation is applied to high frequency band energies:

$$S_{HFB}[k] = \ln pSrcHFB[k] \quad , \qquad 0 \le k < 3$$

2. Auxiliary log energies of the input signal power spectrum are calculated:

$$S_{LFB_aux}[l] = \max\left(-50, \ln\left(\sum_{i=n1[l]}^{n1[l+1]-1} pInSWP[i]\right)\right) \quad , \qquad 0 \le l < 3$$

*n*1[0] = 33, *n*1[1] = 39, *n*1[2] = 49, *n*1[3] = 65.

3. Energies are coded as:

$$code[k,l] = S_{LFB_aux}[k] - S_{HFB}[l] \quad , \qquad 0 \le k,l < 3$$

4. Auxiliary bands for decoding are calculated:

$$S_{Dec_aux}[l] = \max\left(-50, \ln\left(\frac{1}{2}\sum_{i=n2[l]}^{n2[l+1]-1} pDSWP[i]\right)\right) \quad , \qquad 0 \le l < 3$$

*n*2[0] = 66, *n*2[1] = 77, *n*2[2] = 97, *n*2[3] = 129.

5. Decoded high frequency band energies are computed as:

$$pDstHFB[k] = \sum_{l=0}^{2} w[l] \cdot \left(S_{Dec_aux}[l] - code[l,k]\right) \quad , \qquad 0 \le k < 3$$

w[0] = 0.1, *w*[1] = 0.2, *w*[2] = 0.7.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcHFB*, *pInSWP*, *pDSWP*, or *pDstHFB* pointer is NULL.
ippStsBadArg	Indicates an error when a non-positive or NaN argument of the logarithm operation is detected.

BlindEqualization_Aurora

Equalizes the cepstral coefficients.

Syntax

```
IppStatus ippsBlindEqualization_Aurora_32f(const Ipp32f* pRefs, Ipp32f* pCeps,
    Ipp32f* pBias, int len, Ipp32f val);
IppStatus ippsBlindEqualization_Aurora_16s(const Ipp16s* pRefsQ6, Ipp16s*
    pCeps, Ipp16s* pBias, int len, Ipp32s valQ6);
```

Parameters

pRefs	Pointer to the input vector of reference cepstrum [*len*].
pCeps	Pointer to the input and output vector of cepstrum [*len*].
pBias	Pointer to the input and output vector of bias [*len*].
len	The number of cepstral coefficients.
val	The log energy value.

Description

The function `ippsBlindEqualization_Aurora` is declared in the `ippsr.h` file. This function equalizes cepstral coefficients using the LMS algorithm ([ES202], 5.4):

$$step = 0.0087890625 \cdot \min(1, \max(0, val - 211/64))$$

$$pCeps[i] = pCeps[i] - pBias[i]$$

$$pBias[i] += step \cdot (pCeps[i] - pRefs[i]) \quad , \qquad 0 \le i < len$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pRefs*, *pCeps*, or *pBias* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

DeltaDelta_Aurora

Calculates the first and second derivatives according to ETSI ES 202 050 standard.

Syntax

```
IppStatus ippsDeltaDelta_Aurora_16s_D2Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
    dstStep, int height, int deltaMode, int scaleFactor);
IppStatus ippsDeltaDeltaMul_Aurora_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pVal, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
    scaleFactor);
IppStatus ippsDeltaDelta_Aurora_32f_D2(const Ipp32f* pSrc, Ipp32f* pDst, int
    dstStep, int height, int deltaMode);
IppStatus ippsDeltaDeltaMul_Aurora_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pVal, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Parameters

pSrc	Pointer to the input feature sequence [*height**14].
pDst	Pointer to the output feature sequence [*height***dstStep*].
dstStep	Length of the output feature in the output sequence *pDst*.
height	Number of feature vectors.
deltaMode	Execution mode.
pVal	Pointer to the delta coefficients vector [39].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Description

The functions `ippsDeltaDelta_Aurora` and `ippsDeltaDeltaMul_Aurora` are declared in the `ippsr.h` file. These functions calculate full feature vectors according to ETSI ES 202 050 standard.
The input vectors of length 14 contain values $c_1,\ldots c_{12}$, c_0, $\ln E$. First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated ([ES202], 9.1-9.2).

The function implies the following constraints on the delta coefficients:

$val[j] = pVal[j]$ for `ippsDeltaDelta_Aurora` function
$val[j] = 1$ for `ippsDeltaDeltaMul_Aurora` function.

vel[-4] = -1.0, *vel*[-3] = -0.75, *vel*[-2] = -0.5, *vel*[-1] = -0.25, *vel*[0] = 0.0, *vel*[1] = 0.25, *vel*[2] = 0.5, *vel*[3] = 0.75, *vel*[4] = 1.0.

acc[-4] = 1.0, *acc*[-3] = 0.25, *acc*[-2] = -0.285714, *acc*[-1] = -0.607143, *acc*[0] = -0.714286, *acc*[1] = -0.607143, *acc*[2] = -0.285714, *acc*[3] = 0.25, *acc*[4] = 1.0.

The execution mode *deltaMode* provides additional controls for the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to IPP_DELTA_BEGIN|IPP_DELTA_END

Functions perform the offline delta feature calculation. All base features are assumed available at the time of the calculation. The base features are copied from the input stream *pSrc* to the output stream *pDst* as follows:

$$pDst[i \cdot dstStep + j] = val[j] \cdot pSrc[i \cdot 14 + j] \quad , \quad 0 \le j < 12.$$

$$pDst[i \cdot dstStep + 12] = val[12] \cdot (pSrc[i \cdot 14 + 12] \cdot 0.6/23 + pSrc[i \cdot 14 + 13] \cdot 0.4)$$

$$0 \le i < height \quad . \tag{8.11}$$

Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[max(0, min(i + k, height - 1)) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[max(0, min(i + k, height - 1)) \cdot dstStep + j]$$

for $0 \le i < height$, $0 \le j < 13$.

2. *deltaMode* is equal to 0

Online delta feature calculation is performed. The input features are the current segment of a continuous stream. The function ippsDeltaDelta_Aurora calculates the partial delta features accordingly. First, the base features are copied as follows:

$$pDst[(i + 8) \cdot dstStep + j] = val[j] \cdot pSrc[i \cdot 14 + j] \quad , \quad 0 \le j < 12.$$

$$pDst[(i + 8) \cdot dstStep + 12] = val[12] \cdot (pSrc[i \cdot 14 + 12] \cdot 0.6/23 + pSrc[i \cdot 14 + 13] \cdot 0.4)$$

$$0 \le i < height \quad . \tag{8.12}$$

Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[(i + k) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[(i + k) \cdot dstStep + j]$$

for $0 \le i - 4 < height$, $0 \le j < 13$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the first and second derivatives in $pDst[k \cdot dstStep + 13 + l]$ are assumed available through a previous derivative calculation for $0 \le j < 13$, $0 \le i < 8$, $0 \le k < 26$, $0 \le l < 4$.

3. *deltaMode* is equal to IPP_DELTA_BEGIN

Functions perform the partial online delta feature calculation, where the beginning of the input stream is known. First, the base features are copied as in equation (8.11). Then, the first and second derivatives are calculated as follows:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[\max(0, i + k) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[\max(0, i + k) \cdot dstStep + j]$$

for $0 \le i < height - 4$, $0 \le j < 13$.

4. *deltaMode* is equal to IPP_DELTA_END

Functions perform the partial online delta feature calculation, where the ending of the input stream is known. First, the base features are copied as in equation (8.12). Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[min(i + k, height - 1) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[min(i + k, height - 1) \cdot dstStep + j]$$

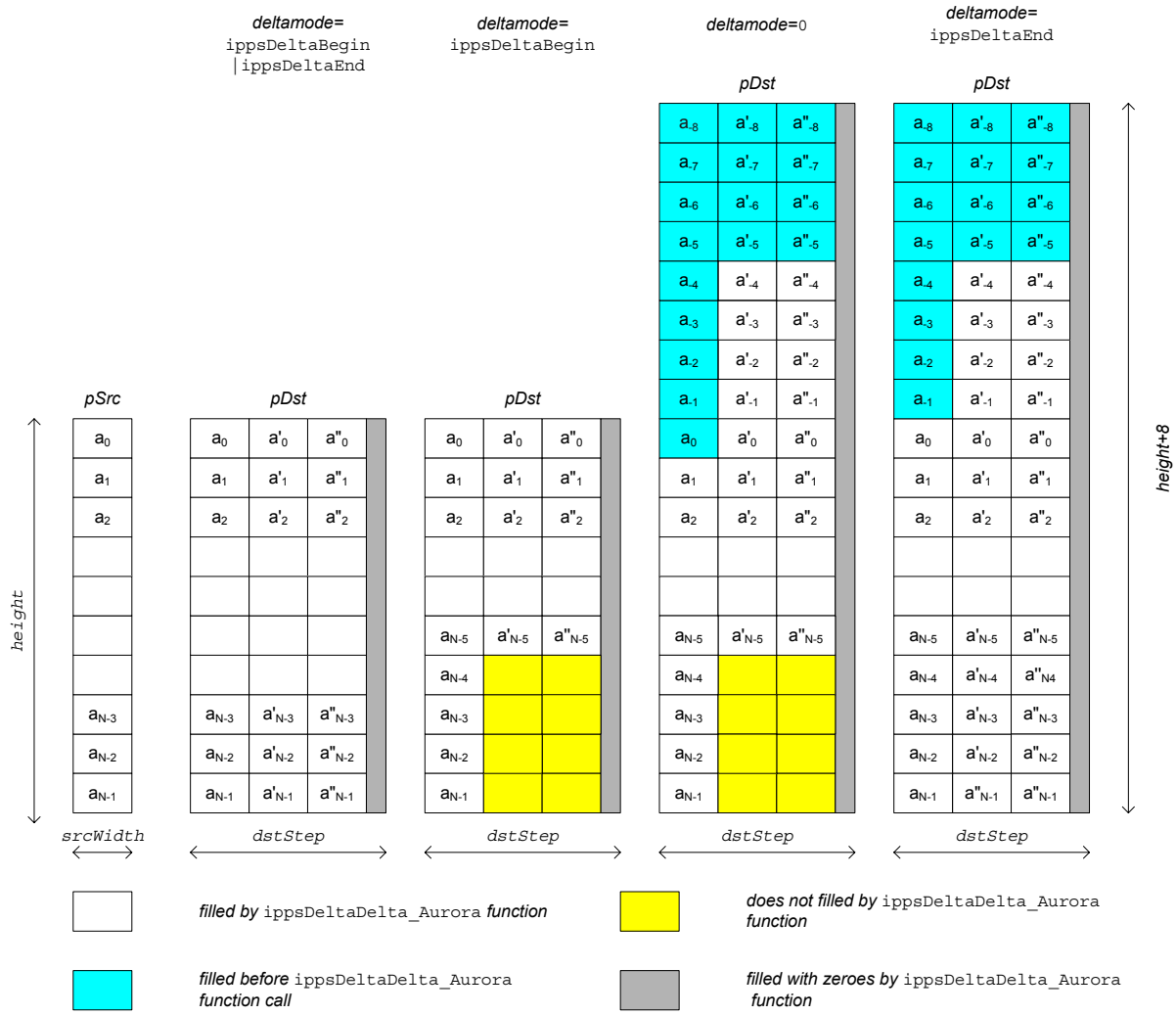
for $0 \le i - 4 < height + 4$, $0 \le j < 13$.
In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the first and second derivatives in $pDst[k \cdot dstStep + 13 + l]$ are assumed available through a previous derivative calculation for $0 \le j < 13$, $0 \le i < 8$, $0 \le k < 26$, $0 \le l < 4$.

The Figure 8-6 illustrates the above four delta calculation modes.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pVal* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* is less than 0; or *height* is less than 8 when *deltaMode* is equal to IPP_DELTA_BEGIN; or *height* is equal to 0 when *deltaMode* is not equal to IPP_DELTA_END.
ippStsStrideErr	Indicates an error when *dstStep* is less than 39.

Figure 8-6 Execution Modes of `ippsDeltaDelta_Aurora` function



VADGetBufSize_Aurora

Queries the memory size for VAD decision.

Syntax

```
IppStatus ippsVADGetBufSize_Aurora_32f(int* pSize);
IppStatus ippsVADGetBufSize_Aurora_16s(int* pSize);
```

Parameters

pSize	Pointer to the output value of the memory size needed for VAD decision.

Description

The function `ippsVADGetBufSize_Aurora` is declared in the `ippsr.h` file. This function returns the size of memory that should be allocated by user. The memory block of *pSize*[0] bytes is used for VAD algorithm initialization by `ippsVadInit_Aurora` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSize* pointer is `NULL`.

VADInit_Aurora

Gets the VAD structure size.

Syntax

```
IppStatus ippsVADInit_Aurora_32f(char* pVADmem);
IppStatus ippsVADInit_Aurora_16s(char* pVADmem);
```

Parameters

pVADmem	Pointer to the VAD decision memory.

Description

The function `ippsVADInit_Aurora` is declared in the `ippsr.h` file. This function initializes the VAD decision process.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pVADmem* pointer is `NULL`.

VADDecision_Aurora

Takes the VAD decision.

Syntax

```
IppStatus ippsVADDecision_Aurora_32f(const Ipp32f* pCoeff, const Ipp32f*
    pTrans, IppVADDecision_Aurora*  pRes, int nbSpeechFrame, char* pVADmem);
IppStatus ippsVADDecision_Aurora_16s(const Ipp16s* pCoeff, const Ipp16s*
    pTrans, IppVADDecision_Aurora*  pRes,int nbSpeechFrame, char* pVADmem);
```

Parameters

pCoeff	Pointer to the input vector of Mel-warped Wiener filter coefficients [25].
pTrans	Pointer to the input vector of Wiener filter transfer function [64].
pRes	Pointer to VAD decision (“1” if voice is detected, “0” otherwise).
nbSpeechFrame	Speech frame hangover counter.
pVADmem	Pointer to the VAD decision memory.

Description

The function `ippsVADDecision_Aurora` is declared in the `ippsr.h` file. This function takes VAD decision for the input frame according to ([ES202], Annex A).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCoeff*, *pTrans*, *pRes* or *pVADmem* pointer is `NULL`.

VADFlush_Aurora

Takes VAD decision for zero input frame.

Syntax

```
IppStatus ippsVADFlush_Aurora_32f(IppVADDecision_Aurora* pRes, char* pVADmem);
IppStatus ippsVADFlush_Aurora_16s(IppVADDecision_Aurora* pRes, char* pVADmem);
```

Parameters

pCoeff	Pointer to the input vector of Mel-warped Wiener filter coefficients [25].
pTrans	Pointer to the input vector of Wiener filter transfer function [64].
pRes	Pointer to VAD decision (“1” if voice is detected, “0” otherwise).
pVADmem	Pointer to the VAD decision memory.

Description

The function `ippsVADDecision_Aurora` is declared in the `ippsr.h` file. This function takes VAD decision for the zero frame. It is used when speech is finished.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCoeff*, *pTrans*, *pRes* or *pVADmem* pointer is `NULL`.

Ephraim-Malah Noise Suppressor

This section describes the Intel IPP functions that can be combined to construct an implementation of the Ephraim-Malah Noise Suppressor (EMNS) originally described in [Eph84]. The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The EMNS is a frequency domain noise suppression algorithm that seeks to minimize the minimum mean squared error of the speech short-time spectral amplitude estimate.

Intel IPP EMNS primitives support the following features:

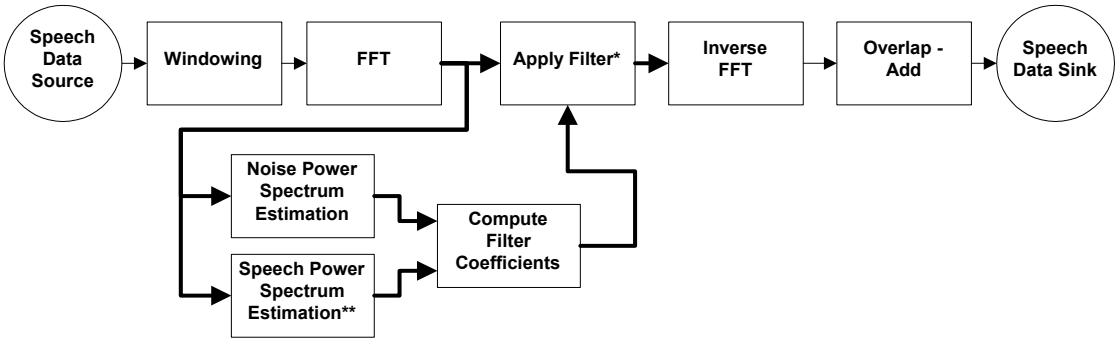
- Filter update
- Noise floor estimation

The use of these functions is demonstrated in Intel IPP Speech Recognition Samples. See *Audio Processing Sample* downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

Noise Suppressor Architecture

The major building blocks of the Ephraim-Malah Noise Suppressor are shown in Figure 8-7, where the steps are given to apply noise reduction to a speech signal stream.

Figure 8-7 Major blocks in an Ephraim-Malah Noise Suppression System



* applying the filter consists of multiplying each FFT bin by a real-valued filter coefficient

** speech power spectral estimation is typically accomplished via spectral subtraction

Ephraim-Malah Noise Suppressor Details

Algorithm Steps

1. **New input block.** The most recently acquired N/2 input samples ζ_n and the previous N/2 input samples ζ_{n-1} make up the new input block $\mathbf{z}_n$.

$$\mathbf{z}_n = \begin{bmatrix} \zeta_{n-1} \\ \zeta_n \end{bmatrix}$$

2. **Windowed DFT.** The input block is multiplied by the square root of a window function. The window function is constrained such that when its first half is added to its second half, all values add to one. A convenient window function is the triangular window defined as

$$w(m) = \begin{cases} \dfrac{m+0.5}{N/2} & \text{for } m = 0,\ldots,N/2-1 \\ 1 - w(m - N/2) & \text{for } m = N/2,\ldots,N-1 \end{cases}$$

The discrete Fourier transform of the input is calculated as follows.

$$\mathbf{Z}_n = \mathbf{F}\left(\mathbf{z}_n \bullet \sqrt{\mathbf{w}}\right)$$

Here, • denotes point-wise multiplication and $\sqrt{\mathbf{w}}$ denotes a vector containing the square root of the entries of **w**. F is the Fourier transform matrix with entries, $f(m,n)=\exp(-j2\pi\pi mn/N)$, where N is the size of the transform.

3. **Update noisy speech PSD.** The noisy speech magnitude-squared spectral components are averaged to provide an estimate of the noisy speech power spectrum.

$$\mathbf{P}_n^z(k) = \beta_n \cdot \left|\mathbf{Z}_n(k)\right|^2 + (1-\beta_n)\cdot \mathbf{P}_{n-1}^z(k)$$

The adaptive step size is defined as

$$\beta_n = \beta_{\min} + \rho_{n-1}^y(\beta_{\max} - \beta_{\min})$$

where $\beta_{\min} = 0.9$, $\beta_{\max} = 1.0$, and ρ_{n-1}^{y} is the likelihood of speech presence in bin k.

4. **Update clean speech PSD.** A course estimate of the clean speech power spectral components is obtained by spectral subtraction and averaging.

$$\mathbf{P}_n^y(k) = \alpha_n \cdot \left|\hat{\mathbf{Y}}_{n-1}(k)\right|^2 + (1-\alpha_n)\cdot \psi_0\left(\mathbf{P}_n^z(k) - \mathbf{P}_{n-1}^v(k)\right)$$

The thresholding operator ψ is defined as

$$\psi_c(x) = \begin{cases} c, & x \le c \\ x, & x > c \end{cases}$$

The adaptive step size is defined as

$$\alpha_n = \alpha_{\min} + \left(1 - \rho_{n-1}^y\right)\left(\alpha_{\max} - \alpha_{\min}\right)$$

where $\alpha_{\min} = 0.91$, $\alpha_{\max} = 0.95$, and ρ_{n-1}^{y} is the likelihood of speech presence in bin k.

NOTE. The previous frame's noise power spectral component is used in this calculation. If the noise floor estimator is independent of the rest of the algorithm, it may be possible to use the current frame's noise estimate instead. In this document, a dependency is assumed.

5. **Update Wiener filter weights.** It turns out that one of the quantities used to compute the Ephraim-Malah suppression rule is actually the Wiener filter (a different noise suppression rule). The Wiener filter weights are given by:

$$\mathbf{W}_n^y(k) = \psi_{W_{\min}}\left(\frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^y(k) + \mathbf{P}_{n-1}^v(k)}\right)$$

where $W_{\min}$ corresponds to the threshold defined in [Cap94]. There it was recommended that a lower limit for a priori SNR

$$\Re_n^{prio}(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_{n-1}^v(k)}$$

of $\Re_{\min dB}^{prio} = -15.0\,\mathrm{dB}$ be imposed to avoid musical noise. This corresponds to

$$W_{\min} = \frac{1}{1 + 10^{-\frac{\Re_{\min dB}^{prio}}{10}}}$$

Note that if the Wiener filter is written in terms of the a priori SNR, then the Wiener filter calculation can be replaced by a table lookup. This may be advantageous on processors where division is expensive.

6. **Update a posteriori SNR.** The a posteriori signal to noise ratio for each frequency bin is defined as follows.

$$\Re_n^{post}(k) = \frac{\mathbf{P}_n^z(k)}{\mathbf{P}_{n-1}^v(k)}$$

7. **Update Ephraim-Malah filter weights.** The Ephraim-Malah filter weights are given by

$$\mathbf{H}_n^y(k) = \frac{1}{\Re_n^{post}(k)} \cdot M\left(\mathbf{W}_n^y(k)\Re_n^{post}(k)\right)$$

where $M(\cdot)$ is the function

$$M(\theta) = \frac{1}{2} \cdot \sqrt{\pi\theta} \cdot e^{-\frac{\theta}{2}}\left[(1+\theta)\cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right)\right]$$

Note that this function can be replaced by a table lookup on processors where Bessel function evaluation is expensive

8. **Update noise PSD estimate.** A noise power spectral estimator must be employed to calculate $\mathbf{P}_n^v(k)$. One such estimator is the method of Martin based on minimum statistics and optimal smoothing [Mar01].

9. **Update speech presence likelihood.** The probability of speech presence is not calculated directly. Rather, it is roughly approximated by the MMSE (Wiener) estimator of the overall speech energy.

$$\rho_n^y = \frac{\sum_{k=0}^{N/2} \mathbf{P}_n^y(k)}{\sum_{k=0}^{N/2} \mathbf{P}_n^y(k) + \sum_{k=0}^{N/2} \mathbf{P}_n^v(k)}$$

10. **Apply additional heuristic modifications to filter coefficients.**

The filter coefficients $\mathbf{H}_n^v(k)$ may be modified to improve perceptual speech quality or reduce perceptible musical tones.For example, to efficiently handle loud, low-pass noise such as that encountered in automotive environments, low-frequency filter coefficients (e.g., below 60 Hz) may be set to zero.

11. **Calculate filter output.** The filter output is defined as follows.

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n^y(k) \cdot \mathbf{Z}_n(k)$$

12. **Inverse DFT and overlap-add.** The time-domain filter output is defined as follows.

$$\hat{\mathbf{y}}_{n-1} = \begin{bmatrix} \mathbf{0}_{\frac{N}{2}\times\frac{N}{2}} & \mathbf{I}_{\frac{N}{2}\times\frac{N}{2}} \end{bmatrix} \cdot \sqrt{\mathbf{w}} \bullet \mathbf{F}^{-1}\hat{\mathbf{Y}}_{n-1} + \begin{bmatrix} \mathbf{I}_{\frac{N}{2}\times\frac{N}{2}} & \mathbf{0}_{\frac{N}{2}\times\frac{N}{2}} \end{bmatrix} \cdot \sqrt{\mathbf{w}} \bullet \mathbf{F}^{-1}\hat{\mathbf{Y}}_n$$

Note that he algorithm introduces a delay of N/2 samples in the output.

Data Structures

There is one structure associated with the MCRA noise floor estimator: `IppsMCRAParam`. This structure is used internally and should not be modified by the programmer.

Filter Update Primitives

FilterUpdateEMNS

Calculates the noise suppression filter coefficients.

Syntax

```
IppStatus ippsFilterUpdateEMNS_32s(const Ipp32s* pSrcWienerCoefsQ31, const Ipp32s*
    pSrcPostSNRQ15, Ipp32s* pDstFilterCoefsQ31, int len);
IppStatus ippsFilterUpdateEMNS_32f(const Ipp32f* pSrcWienerCoefs, const Ipp32f*
    pSrcPostSNR, Ipp32f* pDstFilterCoefs, int len);
```

Parameters

pSrcWienerCoefsQ31	Pointer to a real-valued vector containing the Q31 format Wiener filter coefficients ($0.0_{Q31} \le pSrcWienerCoefsQ31[k] < 1.0_{Q31}$).
pSrcWienerCoefs	Pointer to a real-valued vector containing Wiener filter coefficients ($0.0 \le pSrcWienerCoefs[k] < 1.0$).
pSrcPostSNRQ15	Pointer to a real-valued vector containing an estimate of the a posteriori signal to noise ratio in Q15 format ($0_{Q15} < pSrcPostSNRQ15 < 32768.0_{Q15}$).
pSrcPostSNR	Pointer to a real-valued vector containing an estimate of the a posteriori signal to noise ratio ($0 < pSrcPostSNR < 32768.0$).
pDstFilterCoefsQ31	Pointer to a real-valued vector containing the Q31 format filter coefficients ($0.0_{Q31} \le pDstFilterCoefsQ31[k] < 1.0_{Q31}$).
pDstFilterCoefs	Pointer to a real-valued vector containing the filter coefficients ($0.0 \le pDstFilterCoefs[k] < 1.0$).
len	Number of elements contained in input and output vectors ($0 < len < 65536$).

Description

The function `ippsFilterUpdateEMNS` is declared in the `ippsr.h` file. This function calculates the noise suppression filter coefficients. Three filter sizes are typically used: 65, 129, and 257 (corresponding to FFT sizes of 128, 256, and 512). These are recommended

for sample rates Fs ≤11025 Hz, 11025 Hz < Fs ≤22050 Hz, and 22050 Hz < Fs ≤44100 Hz, respectively. The noise suppression filter coefficients are the gains (scalar values between zero and one) that are applied to each FFT bin. These gains appear in the solution of the equation

$$\min E\left\{\left(\left|\mathbf{Y}_n(k)\right| - \left|\hat{\mathbf{Y}}_n(k)\right|\right)^2\right\}$$

where $\mathbf{Y}_n(k)$ is the *k*-th DFT component corresponding to the *n*-th block of speech samples and $\hat{\mathbf{Y}}_n(k)$ is an estimate of $\mathbf{Y}_n(k)$. Speech spectral components are assumed to have a Gaussian probability density. Since speech spectral components are not directly observable, the solution is written in terms of noisy observations, $\mathbf{Z}_n(k)$. The noise is assumed to be additive and Gaussian. The minimum mean squared error amplitude estimate is

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n(k) \cdot \mathbf{Z}_n(k)$$

where

$$\mathbf{H}_n(k) = \frac{\sqrt{\pi}}{2} \cdot \frac{\sqrt{\mathbf{W}_n^y(k) \cdot \Re_n^{post}(k)}}{\Re_n^{post}(k)} \cdot M\left(\mathbf{W}_n^y(k) \cdot \Re_n^{post}(k)\right)$$

$$M(\theta) = e^{-\frac{\theta}{2}}\left[(1+\theta) \cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right)\right]$$

$$\mathbf{W}_n^y(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^y(k) + \mathbf{P}_{n-1}^v(k)}$$

$$\Re_n^{post}(k) = \frac{\left|\mathbf{Z}_n(k)\right|^2}{\mathbf{P}_{n-1}^v(k)}$$

I_0 and I_1 are Bessel functions, $\mathbf{P}_n^v(k)$ is an estimate of the *k*-th component of the power spectrum of the noise,

$\mathbf{P}_n^y(k)$ is an estimate of the *k*-th speech power spectral component.

The function `ippsFilterUpdate_EMNS` computes the filter coefficients $\mathbf{H}_n(k)$ according to the above equation. Power spectral estimates are typically computed using the `ippsAddWeighted` primitive.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrcWienerCoefsQ31*, *pSrcPostSNRQ15*, *pDstFilterCoefsQ31*, *pSrcWienerCoefs*, *pSrcPostSNR*, or *pDstFilterCoefs* pointer is `NULL`.
`ippStsLengthErr`	Indicates an error when *len* has an illegal value.

FilterUpdateWiener

Calculates the Wiener filter coefficients.

Syntax

```
IppStatus ippsFilterUpdateWiener_32s(const Ipp32s* pSrcPriorSNRQ15,Ipp32s*
    pDstFilterCoefsQ31, int len);
IppStatus ippsFilterUpdateWiener_32f(const Ipp32f* pSrcPriorSNR, Ipp32f*
    pDstFilterCoefs, int len);
```

Parameters

pSrcPriorSNRQ15	Pointer to a real-valued Q15 format vector containing an estimate of the a priori signal to noise ratio ($0.0_{Q15} < pSrcPriorSNRQ15[k] < 32768.0_{Q15}$).
pSrcPriorSNR	Pointer to a real-valued vector containing an estimate of the a priori signal to noise ratio ($0.0 < pSrcPriorSNRQ15[k] < 32768.0$).
pDstFilterCoefsQ31	Pointer to a real-valued vector containing the Q31 format filter coefficients ($0.0_{Q31} \le pDstFilterCoefsQ31[k] < 1.0_{Q31}$).
pDstFilterCoefs	Pointer to a real-valued vector containing the filter coefficients ($0.0 \le pDstFilterCoefsQ31[k] < 1.0$).
len	Number of elements contained in input and output vectors ($0 < len < 65536$).

Description

The function `ippsFilterUpdateWiener` is declared in the `ippsr.h` file. This function calculates the Wiener filter coefficients. Three filter sizes are typically used: 65, 129, and 257 (corresponding to FFT sizes of 128, 256, and 512). These are recommended for sample rates Fs ≤11025 Hz, 11025 Hz < Fs ≤22050 Hz, and 22050 Hz < Fs ≤44100 Hz, respectively. The Wiener filter coefficients are the gains (scalar values between zero and one) that are applied to each FFT bin. These gains appear in the solution of the equation

$$\min E\left\{\left|\mathbf{Y}_n(k)-\hat{\mathbf{Y}}_n(k)\right|^2\right\}$$

where $\mathbf{Y}_n(k)$ is the *k*-th DFT component corresponding to the *n*-th block of speech samples and $\hat{\mathbf{Y}}_n(k)$ is an estimate of $\mathbf{Y}_n(k)$. Speech spectral components are assumed to have a Gaussian probability density. Since speech spectral components are not directly observable, the solution is written in terms of noisy observations, $\mathbf{Z}_n(k)$. The noise is assumed to be additive and Gaussian. The minimum mean squared error amplitude estimate is

$$\hat{\mathbf{Y}}_n(k)=\mathbf{W}_n^y(k)\cdot\mathbf{Z}_n(k)$$

where

$$\mathbf{W}_n^y(k)=\frac{1}{1+1/\Re_n^{prio}(k)}$$

$$\Re_n^{prio}(k)=\frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^v(k)}$$

This primitive implements a coarse approximation to $\mathbf{W}_n^y(k)$ in order to achieve very low execution time. This coarse approximation was found to be sufficient for most noise reduction applications. However, if high accuracy is desired one may choose to call `ippsDiv_32s_Sfs` with the proper parameters instead of `ippsFilterUpdateWiener_32s`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcPriorSNRQ15*, *pDstFilterCoefsQ31*, *pSrcPriorSNR*, or *pDstFilterCoefs* pointer is NULL.
ippStsLengthErr	Indicates an error when *len* has an illegal value.
ippStsBadArgErr	Indicates an error when at least one of the 'a priori' SNR array elements is negative. Zero filter coefficient is returned for such elements.

Noise Floor Estimation Primitives

GetSizeMCRA

Calculates the size in bytes required for the IppMCRAState *state structure.*

Syntax

```
IppStatus ippsGetSizeMCRA_32s(int nFFTSize, int* pDstSize);
IppStatus ippsGetSizeMCRA_32f(int nFFTSize, int* pDstSize);
```

Parameters

nFFTSize	Size of the FFT used for noise PSD estimations (8 ≤ *nFFTSize* ≤ 8192).
pDstSize	Pointer to the variable to contain the size in bytes.

Description

The function ippsGetSizeMCRA is declared in the ippsr.h file. This function calculates the size in bytes that should be allocated for the IppMCRAState state structure required by the ippsUpdateNoisePSDMCRA function. The function ippsGetSizeMCRA should be called before allocating memory and before calling ippsInitMCRA.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pDstSize* pointer is NULL.
ippStsSizeErr	Indicates an error when *nFFTSize* has an Illegal value.

InitMCRA

Initializes the IppMCRAState *state structure.*

Syntax

```
IppStatus ippsInitMCRA_32s(int nSamplesPerSec, int nFFTSize, IppMCRAState*
    pDst);
IppStatus ippsInitMCRA_32f(int nSamplesPerSec, int nFFTSize, IppMCRAState32f*
    pDst);
```

Parameters

nSamplesPerSec	Input sample rate (8000 ≤ *nSamplesPerSec* ≤ 48000).
nFFTSize	Size of the FFT used to for noise PSD estimation (8 ≤ *nFFTSize* ≤ 8192).
pDst	Pointer to the state structure.

Description

The function ippsInitMCRA is declared in the ippsr.h file. This function initializes the state structure for the ippsUpdateNoisePSDMCRA function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates no error when *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *nFFTSize* has an illegal value.
ippStsRangeErr	Indicates an error when *nSamplesPerSec* is out of range.

NOTE. State memory address stored in *pDst* must be aligned to 32-bit word boundary.

AltInitMCRA

Allocates memory and initializes the `IppMCRAState` *state structure.*

Syntax

```
IppStatus ippsAltInitMCRA_32s(int nSamplesPerSec, int nFFTSize, IppMCRAState**
    ppDst);
IppStatus ippsAltInitMCRA_32f(int nSamplesPerSec, int nFFTSize, int
    nUpdateSamples, IppMCRAState32f* pDst);
```

Parameters

nSamplesPerSec	Input sample rate (0 < *nSamplesPerSec* ≤ 48000).
nFFTSize	Size of the FFT used to for noise PSD estimation (8 ≤ *nFFTSize* ≤ 8192).
nUpdateSamples	Number of new samples per frame.
ppDst	Pointer to the pointer to the state structure.
pDst	Pointer to the state structure.

Description

The function `ippsInitAllocMCRA` is declared in the `ippsr.h` file. This function allocates memory and initializes the state structure for the `ippsUpdateNoisePSDMCRA` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *nFFTSize* has an illegal value.
`ippStsRangeErr`	Indicates an error when *nSamplesPerSec* is out of range.

UpdateNoisePSDMCRA

Re-estimates the noise power spectrum.

Syntax

```
IppStatus ippsUpdateNoisePSDMCRA_32s_I(const Ipp32s* pSrcNoisySpeech,
    IppMCRAState* pSrcDstState, Ipp32s* pSrcDstNoisePSD);
IppStatus ippsUpdateNoisePSDMCRA_32f_I(const Ipp32f* pSrcNoisySpeech,
    IppMCRAState32f* pSrcDstState, Ipp32f* pSrcDstNoisePSD);
```

Parameters

pSrcNoisySpeech	Pointer to a real-valued vector containing the magnitude squared of the FFT of the noisy speech (*pSrcNoisySpeech* is greater than or equal to zero).
pSrcDstState	Pointer to the state structure.
pSrcDstNoisePSD	Pointer to the noise power spectrum vector.

Description

The function `ippsUpdateNoisePSDMCRA` is declared in the `ippsr.h` file. This function re-estimates the noise power spectrum given a new measurement of the magnitude squared noisy speech. The algorithm is based on the Minima Controlled Recursive Averaging approach described in [Coh02].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrcNoisySpeech*, *pSrcDstState*, or *pSrcDstNoisePSD* pointer is `NULL`.

Voice Activity Detector

This section describes the Intel IPP functions that can be combined to construct a Voice Activity Detector (VAD). The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The VAD consists of parameter estimation and speech modeling heuristics.

IPP VAD primitives support the following features:

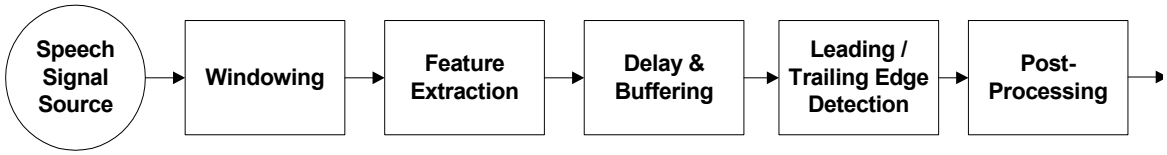
- Peak picking
- Periodicity
- Zero crossing rate

The use of these functions is demonstrated in Intel IPP Speech Recognition Samples. See *Audio Processing Sample* downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

Voice Activity Detector Architecture

A typical voice activity detector is illustrated below. The speech signal is windowed and separated into (possibly overlapping) frames. Feature extraction is then performed. Feature extraction may include one or more of the following: zero crossing rate calculation, energy calculation, segmental SNR estimation, periodicity detection, and sample or sub-band histogram measurement. Measurements are buffered so that speech onset and ending can be distinguished from other spurious events. A delay of 400 milliseconds is typical. Speech leading and trailing edge detection is carried out based on heuristic rules. Post-processing (e.g., median filtering) may be applied in low-delay implementations to eliminate false detections.

Figure 8-8 Major blocks in a Voice Activity Detector



Voice Activity Detection Primitives

FindPeaks

Identifies peaks in the input vector.

Syntax

```
IppStatus ippsFindPeaks_32s8u(const Ipp32s* pSrc, Ipp8u* pDstPeaks, int len,
    int searchSize, int movingAvgSize);
IppStatus ippsFindPeaks_32f8u(const Ipp32f* pSrc,  Ipp8u* pDstPeaks, int len,
    int searchSize, int movingAvgSize);
```

Parameters

pSrc	Pointer to an input vector.
pDstPeaks	Pointer to the output vector containing a one in positions corresponding to a peak in the input vector and zeros elsewhere.
len	Number of elements contained in the input and output vectors (0 < *len* < 65536).
searchSize	Number of elements on either side to consider when picking peak (0 < *searchSize* < 128).
movingAvgSize	Number of elements on either side to include in moving average window that is applied before peak picking (0 ≤ *movingAvgSize* < 128).

Description

The function `ippsFindPeaks` is declared in the `ippsr.h` file. This function identifies peaks in the input vector, places a one in the output vector at the locations of the peaks, and places a zero elsewhere.

A peak is defined as a point *pSrc*[i] such that *pSrc*[i-L] < *pSrc*[i-L+1] < … < *pSrc*[i-1] < *pSrc*[i] > … > *pSrc*[i+L-1] > *pSrc*[i+L] , where *searchSize* is L.

If *movingAvgSize* is greater than 0, then the source vector is smoothed via moving average before peaks are selected.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc* or *pDstPeaks* pointer is NULL.
ippStsLengthErr	Indicates an error when *len* is out of range.
ippStsSizeErr	Indicates an error when *SearchSize* or *movingAvgSize* is out of range.

PeriodicityLSPE

Computes the periodicity of the input speech frame.

Syntax

```
IppStatus ippsPeriodicityLSPE_16s(const Ipp16s* pSrc, int len, Ipp16s*
    pPeriodicityQ15, int* period, int maxPeriod, int minPeriod);
IppStatus ippsPeriodicityLSPE_32f(const Ipp32f* pSrc, int len, Ipp32f*
    pPeriodicity, int* period, int maxPeriod, int minPeriod);
```

Parameters

pSrc	Pointer to an input speech vector.
len	Number of elements contained in the input vector subject to 6 < len ≤ min(16**minPeriod*, 1024)
pPeriodicityQ15	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ($0.0_{Q15} \le$ *pPeriodicityQ15* $\le 1.0_{Q15}$).
pPeriodicity	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling (0.0 ≤ *pPeriodicity* ≤ 1.0).
period	The period (in samples) that minimizes the LSPE cost function.
maxPeriod	Maximum period to search (*minPeriod* < *maxPeriod* < *len*).
minPeriod	Minimum period to search (6 ≤ *minPeriod* < *maxPeriod*).

Description

The function ippsPeriodicityLSPE is declared in the ippsr.h file. This function computes the periodicity of the input speech frame. The periodicity is calculated according the least squares periodicity estimate (LSPE) algorithm defined in [Tuc92]. The periodicity search is designed for speech signals with sample rate between 2000-24000 Hz and may not perform well on music or other sources. If only periodicity (voicing) is required, it is

more efficient to downsample input speech before calling this primitive. For example, at 2000 Hz sample rate, the settings *minPeriod*=10, *maxPeriod*=20, and *len*=64, provide adequate periodicity for some applications.

Return Values

ippStsNoErr	Indicates no error.
ippStsLengthErr	Indicates an error when *len* is out of range.
ippStsRangeErr	Indicates an error when *maxPeriod* or *minPeriod* is out of range.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pPeriodicityQ15* pointer is NULL.

Periodicity

Computes the periodicity of the input block.

Syntax

```
IppStatus ippsPeriodicity_32s16s(const Ipp32s* pSrc, int len, Ipp16s*
    pPeriodicityQ15, int* period, int maxPeriod, int minPeriod);
IppStatus ippsPeriodicity_32f (const Ipp32f* pSrc,  int len, Ipp32f*
    pPeriodicity, int* period, int maxPeriod, int minPeriod);
```

Parameters

pSrc	Pointer to an input vector containing non-negative entries.
len	Number of elements contained in the input vector ($0 < len \le 4096$).
pPeriodicityQ15	Pointer to the Q15 format value corresponding to the normalized sum of the largest harmonic sampling ($0.0_{Q15} \le pPeriodicityQ15 \le 1.0_{Q15}$).
pPeriodicity	Pointer to the value corresponding to the normalized sum of the largest harmonic sampling ($0.0 \le pPeriodicity \le 1.0$).
period	Pointer to the period (in samples) that provided the maximum-energy harmonic sampling.
maxPeriod	Maximum period to search ($minPeriod < maxPeriod < len$).

minPeriod	Minimum period to search (0 < *minPeriod* < *maxPeriod*).

Description

The function `ippsPeriodicity` is declared in the `ippsr.h` file. This function computes the periodicity of the input block. In typical applications, the input block is the magnitude-squared of the discrete Fourier transform of windowed speech. The periodicity is defined as the periodic sampling of the input block that preserves the most energy.

$$\max_{k,T_0} \sum_n x(k + nT_0) \quad \text{where } 0 < k \le T_0$$

Bias removal is performed prior to the search to ensure an accurate measurement.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsLengthErr`	Indicates an error when *len* is out of range.
`ippStsRangeErr`	Indicates an error when *pSrc*[k], *maxPeriod*, or *minPeriod* is out of range.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *period*, or *pPeriodicityQ15* pointer is `NULL`.

Speech Coding Functions

9

This chapter describes Intel® IPP functions that can be used for implementing speech codecs which follow ITU-T recommendations for G711, G.722, G.722.1, G.723.1, G726, G.728 and G.729 codecs, the G.167, G.167 for Echo Canceller, ETSI specifications for GSM-AMR and GSM-FR codecs, as well as 3GPP specification for AMRWB and AMRWB+ codecs.

When properly built, such speech codecs and other components can be compliant with the bit-exact and other specifications for published test vectors.

The chapter also includes the logarithmic encoder-decoder functions that perform the μ-law and A-law companding operations.

The organization of this chapter is the following. The introductory part includes description of rounding modes, defines notational conventions, header files and data structures used by Intel IPP speech coding functions. This is succeeded by the major functionality sections, which include:

- Common Functions
- G.729 Related Functions
- G.723.1 Related Functions
- GSM-AMR Related Functions
- AMR Wideband Related Functions
- AMR Wideband Plus Related Functions
- GSM Full Rate Related Functions
- G.722.1 Related Functions
- G.726 Related Functions
- G.728 Related Functions
- Echo Canceller Related Functions
- G722 Sub-Band ADPCM Speech Codec Related Functions
- Companding Functions

- RT Audio Related Functions

Each section starts with the table that gives the full list of Intel IPP functions specific for that functional group, followed by detailed description of the respective API.

The use of the Intel IPP speech coding functions is demonstrated in Intel IPP Samples. See *Intel IPP Speech Coding Samples* downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

Rounding mode

As many speech codecs have to meet the bit-exact requirement, Intel IPP functions described in this chapter use rounding modes that are different from the default rounding mode used in general signal processing functions.
For general signal processing functions, the default rounding mode can be described as "nearest even", so that the fixed point number $x = N+\alpha,\ 0 \le \alpha < 1$, where N is an integer number, is rounded as given by:

$$\lceil x \rceil = \begin{cases} N\ ,\ 0 \le \alpha < 0.5 \\ N+1\ ,\ 0.5 < \alpha < 1 \\ N\ ,\ \alpha = 0.5\ ,\ N-even \\ N+1\ ,\ \alpha = 0.5\ ,\ N-odd \end{cases}$$

For example, 1.5 will be rounded to 2 and 2.5 to 2.

For functions in this chapter, there are two rounding modes.

The default rounding mode is "clipping", so that the fractional part of the fixed point number is cut off and the result of rounding is always less than the initial value. Specifically, the fixed point number $x = N+\alpha,\ 0 \le \alpha < 1$, where N is an integer number, is always rounded to N.
For example, -1.3 will be rounded to -2 and 1.7 to 1. No special suffix is added to the names of functions that use this default mode.

Another rounding mode is "nearest right", in which the fixed point number $x = N+\alpha,\ 0 \le \alpha < 1$, where N is an integer number, is rounded as follows:

$$\lceil x \rceil = \begin{cases} N\ ,\ 0 \le \alpha < 0.5 \\ N+1\ ,\ 0.5 \le \alpha < 1 \end{cases}$$

For example, 1.5 will be rounded to 2 (same as for “nearest even” mode) but –1.5 to –1 (different from “nearest even” mode where –1.5 is rounded to –2).

The suffix “NR” is added to names of functions that use the “nearest right” rounding mode.

Notational Conventions

This chapter, in addition to conventions used throughout the manual, uses the following notational conventions:

- In the description of function arguments, when an argument refers to a vector, the expression [n] in square brackets that may be given after the explanation of the argument specifies the number of elements (length) of that vector;
- Most of the speech coding functions for their proper execution interpret their integer and integer array arguments as fixed point numbers, which represent real numbers that vary in their specific ranges. Notation Q*n* used in the argument description means that this argument values are used in integer calculations inside the function as real numbers equal to the integer value multiplied by 2^{-n} (where “*n*” is called a scale factor).
  For example, if an argument value is described as “4096 in Q12”, then it is interpreted as the real number 1.0; the value described as “15565 with scale factor 14” represents the real number 0.95; an argument described as “Q14 in [0, 1]” must be passed as an integer value in the range [0,16386].

Definitions

This section identifies the header files required for using the Intel IPP speech coding API described later in this chapter. For the definition of bit rate specifiers, refer to Data Structures section.

The header files ippdefs.h and ippsc.h must be included in order to link against any of the speech coding primitives, as shown in the following example code:

```
#include "ippdefs.h"
#include "ippsc.h"
int main()
{
    ...
    /* call GSM-AMR IPP functions */
     ippsLevinsonDurbin_GSMAMR(pSrcAutoCorr, pSrcDstLpc);
```

```
    ...
}
```

Data Structures

Some of the speech coding functions use a bit rate parameter of the enumerated type `IppSpchBitRate`. The set contains one specifier for each of the supported bit rates, as given in the structure below:

```
typedef enum {
    IPP_SPCHBR_4750
    IPP_SPCHBR_5150
    IPP_SPCHBR_5300
    IPP_SPCHBR_5900
    IPP_SPCHBR_6300
    IPP_SPCHBR_6600
    IPP_SPCHBR_6700
    IPP_SPCHBR_7400
    IPP_SPCHBR_7950
    IPP_SPCHBR_8850
    IPP_SPCHBR_9600
    IPP_SPCHBR_10200
    IPP_SPCHBR_12200
    IPP_SPCHBR_12650
    IPP_SPCHBR_12800
    IPP_SPCHBR_14250
    IPP_SPCHBR_15850
    IPP_SPCHBR_16000
    IPP_SPCHBR_18250
    IPP_SPCHBR_19850
    IPP_SPCHBR_23050
    IPP_SPCHBR_23850
    IPP_SPCHBR_24000
    IPP_SPCHBR_32000
    IPP_SPCHBR_40000
    IPP_SPCHBR_DTX
} IppSpchBitRate;
```

The specifiers `IPP_SPCHBR_4750`, `IPP_SPCHBR_5150`, `IPP_SPCHBR_5900`, `IPP_SPCHBR_6700`, `IPP_SPCHBR_7400`, `IPP_SPCHBR_7950`, `IPP_SPCHBR_10200` and `IPP_SPCHBR_12200` are used in GSM-AMR functions and correspond to 4.75, 5.15, 5.9, 6.7, 7.4, 7.95, 10.2 and 12.2 Kbits/s transmitting rates, respectively.

The specifiers IPP_SPCHBR_5300 and IPP_SPCHBR_6300 are used in G.723.1 codec and correspond to 5.3 (low) and 6.3 (high) Kbits/s transmitting rates.
The specifiers IPP_SPCHBR_9600, IPP_SPCHBR_12800 and IPP_SPCHBR_16000 are used in G.728 codec and correspond to 9.6, 12.8 and 16 Kbits/s transmitting rates.

The specifiers IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000 and IPP_SPCHBR_40000 are used in G.726 codec and correspond to 16, 24, 32 and 40 Kbits/s transmitting rates, respectively.

The specifiers IPP_SPCHBR_6600, IPP_SPCHBR_8850, IPP_SPCHBR_12650, IPP_SPCHBR_14250, IPP_SPCHBR_15850, IPP_SPCHBR_18250, IPP_SPCHBR_19850, IPP_SPCHBR_23050, and IPP_SPCHBR_23850 are used in AMR WB codec and correspond to 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 Kbits/s transmitting rates, respectively.

Common Functions

This section describes common functions used in different speech codecs. The list of these functions is given in the following table:

Table 9-1 Intel IPP Common Speech Coding Functions

Function Base Name	Operation
ConvPartial	Performs linear convolution of 1D signals.
Mul_NR	Multiplies the elements of two vectors.
MulC_NR	Multiplies each element of a vector by a constant value.
MulPowerC_NR	Performs power weighting for each element of a vector.
AutoScale	Scales by the maximal elements.
DotProdAutoScale	Computes the dot product of two vectors using the automatic scaling.
InvSqrt	Computes inverse square root of vector elements.
AutoCorr	Calculates autocorrelation of a vector.
AutoCorrLagMax	Estimates the maximum auto-correlation of a vector.
AutoCorr_NormE	Estimates normal auto-correlation of a vector.
CrossCorr	Estimates the cross-correlation of two vectors.
CrossCorrLagMax	Estimates the maximum cross-correlation between two vectors.

Table 9-1 Intel IPP Common Speech Coding Functions (continued)

Function Base Name	Operation
SynthesisFilter	Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).

ConvPartial

Performs linear convolution of 1D signal.

Syntax

```
IppStatus ippsConvPartial_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsConvPartial_16s32s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    ipp32s* pDst, int len);
IppStatus ippsConvPartial_NR_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
IppStatus ippsConvPartial_NR_Low_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the source and destination vectors.
scaleFactor	Scale factor used for output data scaling.

Description

The function `ippsConvPartial` is declared in the `ippsc.h` file. The function flavor `ippsConvPartial_16s32s` computes the convolution of the vectors *pSrc1* and *pSrc2* as:

$$pDst[i] = \sum_{j=0}^{i} pSrc1[i] \cdot pSrc2[i-j] \ , \quad i = 0, \ldots len-1 \ .$$

The function flavor `ippsConvPartial_16s_Sfs` computes the convolution of the vectors *pSrc1* and *pSrc2* and scales output data as given by:

$$pDst[i] = 2^{-scaleFactor} \cdot \sum_{j=0}^{i} pSrc1[i] \cdot pSrc2[i-j] \ , \quad i = 0, ..len-1$$

Computed results are clipped.

The function flavor `ippsConvPartial_NR_16s` performs the same operation as the function `ippsConvPartial_16s32s`, but uses NR rounding and the default scale factor equal to 15.

The function flavor `ippsConvPartial_NR_Low_16s` performs the same operation as the function `ippsConvPartial_NR_16s`, but it requires each value of the sum does not exceed the `Ipp32s` data type range.

Note that the output results produced by the partial convolution function are the same as the first *len* results of the general signal processing convolution function `ippsConv_16s_Sfs` , if the difference due to rounding is not taken into account.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.
`ippStsScaleRangeErr`	Indicates an error when *scaleFactor* is negative.

InterpolateC_NR

Computes the weighted sum of two vectors

Syntax

```
IppStatus ippsInterpolateC_NR_16s(const Ipp16s* pSrc1, Ipp16s val1, int
    val1ScaleFactor, const Ipp16s* pSrc2, Ipp16s val2, int val2ScaleFactor,
    Ipp16s* pDst, int len);
```

Parameters

pSrc1	Pointer to the first source vector
val1	First factor
val1ScaleFactor	First factor scale
pSrc2	Pointer to the second source vector
val2	Second factor
val2ScaleFactor	Second factor scale
pDst	Pointer to the destination vector
len	Length of the vectors

Description

This function computes the weighted sum of two vectors as

$$pDst[i] = (2^{val1ScaleFactor} \cdot val1 \cdot pSrc1[i] + 2^{val2ScaleFactor} \cdot val2 \cdot pSrc2[i]) \cdot 2^{-16}, \quad i = 0, \ldots, len-1$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1, pSrc2 or pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.
ippStsSizeErr	Indicates an error when *ScaleFactor1 or ScaleFactor2* is less than zero.

Mul_NR

Multiplies the elements of two vectors.

Syntax

```
IppStatus ippsMul_NR_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_NR_16s_ISfs (const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
    int scaleFactor);
```

Parameters

pSrc1, *pSrc2*	Pointers to the source vectors to be multiplied.
pDst	Pointer to the destination vector.
pSrc	Pointer to the vector to be multiplied by the elements of *pSrcDst* (for in-place operation).
pSrcDst	Pointer to the source and destination vector (for in-place operation).
len	Number of elements in each vector.
scaleFactor	Scale factor for output data scaling.

Description

The function `ippsMul_NR` is declared in the `ippsc.h` file. This function multiplies the first source vector *pSrc1* by the second source vector *pSrc2* element-wise, and stores results in *pDst*.

The in-place function flavor multiplies the vector *pSrc* by the vector *pSrcDst* element-wise, and stores results in *pSrcDst*.

Both function flavors scale the multiplication results in accordance with the *scaleFactor* value (see "Integer Scaling" in Chapter 2). If output values exceed the data range, the results are saturated.

Functions `ippsMul_NR` perform "nearest right" rounding needed to meet the bit-to-bit exactness requirement.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, *pSrc*, *pSrcDst*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsScaleRangeErr`	Indicates an error when *scaleFactor* is less than 0.

MulC_NR

Multiplies each element of a vector by a constant value.

Syntax

```
IppStatus ippsMulC_NR_16s_Sfs (Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsMulC_NR_16s_ISfs (Ipp16s val, Ipp16s* pSrcDst, int len, int
    scaleFactor);
```

Parameters

val	The scalar value used to multiply to each element of the source vector.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Scale factor for output data scaling.

Description

The function `ippsMulC_NR` is declared in the `ippsc.h` file. This function multiplies the source vector *pSrc* by the scalar value *val*, and stores the result in *pDst*.

The in-place function flavor multiplies the vector *pSrcDst* by *val*, and stores the result in *pSrcDst*.

Both function flavors scale the multiplication results in accordance with the *scaleFactor* value (see "Integer Scaling" in Chapter 2). If output values exceed the data range, the results are saturated.

Functions `ippsMulC_NR` perform "nearest right" rounding needed to meet the bit-to-bit exactness requirement.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsScaleRangeErr	Indicates an error when *scaleFactor* is less than 0.

MulPowerC_NR

Performs power weighting for each element of a vector.

Syntax

```
IppStatus ippsMulPowerC_NR_16s_Sfs (const Ipp16s* pSrc, Ipp16s val, Ipp16s*
    pDst, int len, int scaleFactor);
```

Parameters

pSrc	Pointer to the source vector.
val	The scalar value to be multiplied to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in each vector.
scaleFactor	Scale factor for output data scaling.

Description

The function ippsMulPowerC_NR is declared in the ippsc.h file. This function multiplies each element of the source vector *pSrc* by a power of a scalar value, *val*, and stores the results in *pDst*. The calculation is performed as:

$$pDst[i] = val^{i} \cdot pSrc[i] \ , \quad 0 \le i < len$$

The function scales the multiplication results in accordance with the *scaleFactor* value (see "Integer Scaling" in Chapter 2). If output values exceed the data range, the results are saturated.

The function ippsMulPowerC_NR performs "nearest right" rounding needed to meet the bit-to-bit exactness requirement.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsScaleRangeErr	Indicates an error when *scaleFactor* is less than 0.

AutoScale

Scales by the maximal elements.

Syntax

```
IppStatus ippsAutoScale_16s (const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int* pScale);
IppStatus ippsAutoScale_16s_I(Ipp16s* pSrcDst, int len, int* pScale);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in each vector.
pScale	Pointer to the input/output scaling factor.

Description

The function ippsAutoScale is declared in the ippsc.h file. This function scales the input vector as follows:

$$pDst[i] = 2^{scaleFactor} \cdot pSrc[i] \ , \quad i = 0, ..len-1 \ ,$$

where $scaleFactor = normMax\text{-}pScale[0]$,
and $normMax$ is calculated so that the maximal absolute value of the vector can be normalized. The value $pScale[0]$ is the input scaling factor. This scaling factor is replaced by *scaleFactor* after processing.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *pSrcDst*, or *pScale* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsScaleRangeErr	Indicates an error when the scale factor pointed by *pScale* is less than 0.

DotProdAutoScale

Computes the dot product of two vectors using the automatic scaling.

Syntax

```
IppStatus ippsDotProdAutoScale_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32s* pDp, int* pSfs);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
len	Number of elements in each vector.
pDp	Pointer to the output result.
pSfs	Pointer to return the scaling factor.

Description

The function ippsDotProdAutoScale is declared in the ippsc.h file. This function computes the dot product of two vectors and automatically scales it during calculation, adjusting the scale factor so as to eliminate possible overflow in the process:

$$pDp = 2^{scaleFactor} \cdot \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i]$$

The final scaling factor is returned via *pSfs*.
Vectors *pSrc1* and *pSrc2* must have the same length, *len*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, *pDp*, or *pSfs* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsOverflow	Indicates a warning that at least one result value was saturated.

InvSqrt

Computes inverse square root of vector elements.

Syntax

```
IppStatus ippsInvSqrt_32s_I (Ipp32s* pSrcDst, int len );
```

Parameters

pSrcDst	Pointer to the source and destination vector.
len	Number of elements in the vector.

Description

The function ippsInvSqrt is declared in the ippsc.h file. This function computes the inverse square roots as:

$$pDst[i] = \frac{2^{31}}{\sqrt{pSrc[i]}}, \quad i = 0, ..len-1,$$

using the approximation table. The input and output vectors are scaled by 30 bits.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

AutoCorr

Calculates autocorrelation of a vector.

Syntax

```
IppStatus ippsAutoCorr_16s32s (const Ipp16s* pSrc, int srcLen, Ipp32s* pDst,
    int dstLen );
```

Parameters

pSrc	Pointer to the first source vector [*srcLen*].
srcLen	Length of the source vector.
pDst	Pointer to the destination vector [*dstLen*].
dstLen	Length of the destination vector (the number of autocorrelation values to calculate).

Description

The function `ippsAutoCorr` is declared in the `ippsc.h` file. This function calculates autocorrelation of the input vector as follows:

$$pDst[n] = \sum_{i=n}^{srcLen-1} pSrc[i-n] \cdot pSrc[i] \ , \ n = 0,\ldots,dstLen-1 \ .$$

and stores the result in *pDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *srcLen* or *dstLen* is less or equal to 0.

AutoCorrLagMax

Estimates the maximum auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorrLagMax_Inv_16s (const Ipp16s* pSrc, int len, int
    lowerLag, int upperLag, Ipp32s* pMax, int* pMaxLag);
IppStatus ippsAutoCorrLagMax_Fwd_16s (const Ipp16s* pSrc, int len, int
    lowerLag, int upperLag, Ipp32s* pMax, int* pMaxLag);
IppStatus ippsAutoCorrLagMax_32f (const Ipp32f* pSrc, int len, int
    lowerLag, int upperLag, Ipp32f* pMax, int* pMaxLag);
```

Parameters

pSrc	Pointer to the input vector [-*upperLag*, *len*].
len	Length of autocorrelation.
lowerLag	Lower input lag value.
upperLag	Upper input lag value.
pMax	Pointer to the output maximum of the correlation.
pMaxLag	Pointer to the output lag which holds the maximum of the correlation.

Description

These functions are declared in the `ippsc.h` file. The functions `ippsAutoCorrLagMax_Inv` and `ippsAutoCorrLagMax_32f` find the maximum autocorrelation within the given lag range as:

$$pMax = \max_{n} \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i-n] \ , \ lowerLag \le n \le upperLag \ .$$

The function `ippsAutoCorrLagMax_Fwd` finds the maximum autocorrelation within the given lag range as given by:

$$pMax = \max_{n} \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n] \ , \ lowerLag \le n \le upperLag \ .$$

If several maximums exist, functions return the first of them.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pMax*, or *pMaxLag* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

AutoCorr_NormE

Estimates normal auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorr_NormE_16s32s (const Ipp16s* pSrc, int len, Ipp32s* pDst,
    int lenDst, int* pNorm);
IppStatus ippsAutoCorr_NormE_NR_16s (const Ipp16s* pSrc, int len, Ipp16s* pDst,
    int lenDst, int* pNorm);
```

Parameters

pSrc	Pointer to the source vector in Q12.
len	Number of elements in the source vector.
pDst	Pointer to the destination vector that stores the estimated auto-correlation results of the source vector.
lenDst	The number of elements in the destination vector.
pNorm	Pointer to the output scale factor.

Description

The function `ippsAutoCorr_NormE` is declared in the `ippsc.h` file. This function calculates the autocorrelation of the source vector *pSrc*. The results are scaled according to the first autocorrelation coefficient (energy) value. Specifically, autocorrelation coefficients are multiplied by the factor 2^{norm}, where $norm \geq 0$ is calculated so as to make the first coefficient normalized:

$$pDst[n] = 2^{norm} \cdot \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n], \qquad 0 \leq n < lenDst,\ 0 \leq norm$$

where

$$pSrc[i] = \begin{cases} pSrc[i] & 0 \leq i < len \\ 0 & otherwise \end{cases}$$

The corresponding scaling factor, *norm*, is returned via *pNorm*.

The function that has the `NR` suffix performs the “nearest right” rounding (see Rounding mode).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* or *pNorm* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* or *lenDst* is less than or equal to zero.
ippStsOverflow	Indicates a warning that at least one result value was saturated.

CrossCorr

Estimates the cross-correlation of two vectors.

Syntax

```
IppStatus ippsCrossCorr_16s32s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32s* pDst, int scaleFactor);
IppStatus ippsCrossCorr_NormM_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp16s* pDst);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
len	Number of elements in the source and destination vectors.
pDst	Pointer to the destination vector.
scaleFactor	Scale factor of the destination vector.

Description

The function ippsCrossCorr is declared in the ippsc.h file. This function estimates the cross-correlation between the vector *pSrc1* and the vector *pSrc2* as given by:

$$corr[n] = 2^{scaleFactor} \cdot \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n] \quad ,$$

where

$$pSrc2[i] = \begin{cases} pSrc2[i] \ , & 0 \le i < len \\ 0 \ , & otherwise \end{cases}$$

Results are stored in the vector *pDst*. Correlation sums are saturated.
Scaling is performed according to the *scaleFactor* value.

The function that has the `NormM` suffix uses the scale factor that normalizes the absolute maximum of the correlation sums.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is `NULL`.
`ippStsScaleRangeErr`	Indicates an error when *scaleFactor* is negative.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

CrossCorrLagMax

Estimates the maximum cross-correlation between two vectors.

Syntax

```
IppStatus ippsCrossCorrLagMax_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, int lag, Ipp32s* pMax, int* pMaxLag);
IppStatus ippsCrossCorrLagMax_32f64f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    int len, int lag, Ipp64f* pMax, int* pMaxLag);
```

Parameters

pSrc1	Pointer to the first source vector [*len*].
pSrc2	Pointer to the second source vector [*len*+*lag*+1].
len	Length of the cross-correlation.
lag	The maximal lag value.
pMax	Pointer to the maximum cross-correlation.
pMaxLag	Pointer to the lag which holds the cross-correlation maximum.

Description

The functions `ippsCrossCorrLagMax` are declared in the `ippsc.h` file. These functions find the maximum of the cross-correlation between two input vectors by the formula:

$$pMax = \max_{0 \le n \le lag} \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n]$$

The functions return the value of the cross-correlation maximum and the lag which holds it. The function `ippsCrossCorrLagMax_16s` returns *pMax* value multiplied by two.

If more than one lag which holds the cross-correlation maximum exist, the function returns the biggest lag.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, *pMax*, or *maxLag* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *lag* is less than `0`.
`ippStsSizeErr`	Indicates an error when *len* is less or equal to `0`.

SynthesisFilter

Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).

Syntax

```
IppStatus ippsSynthesisFilter_NR_16s_Sfs(const Ipp16s*  pLPC, const Ipp16s*
    pSrc, Ipp16s*  pDst, int len, int scaleFactor, const Ipp16s* pMem);
IppStatus ippsSynthesisFilterLow_NR_16s_ISfs(const Ipp16s*  pLPC, Ipp16s*
    pSrcDst, int len, int scaleFactor, const Ipp16s* pMem);
IppStatus ippsSynthesisFilter_NR_16s_ISfs(const Ipp16s*  pLPC, Ipp16s*
    pSrcDst, int len, int scaleFactor, const Ipp16s* pMem);
```

Parameters

pLPC	Pointer to the input filter coefficients a_0, a_1,...,a_{10}.
pSrc	Pointer to the input vector.

pSrcDst	Pointer to the history input/filtered output vector.
pDst	Pointer to the filtered output.
len	Number of elements in the vector.
scaleFactor	Scale factor for the destination vector.
pMem	Pointer to the memory supplied for filtering. Should be updated outside the function with last 10 values of the destination vector.

Description

The function `ippsSynthesisFilter` is declared in the `ippsc.h` file. This function filters the input signal through the synthesis filter as follows:

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) \quad , n = 0,1,\ldots,len-1 .$$

The function flavor `ippsSynthesisFilterLow_NR_16s_ISfs` performs no saturation and assumes that input data are small enough to ensure correct calculation without checking the overflow condition. This mode of operation is indicated by the suffix `Low` in the function name.

All flavors perform “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pLPC*, *pSrcDst*, *pDst*, or *pMem* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsScaleRangeErr`	Indicates an error when *scaleFactor* is not equal to 12 or 13.
`ippStsOverflow`	Indicates a warning that at least one result value was saturated.

G.729 Related Functions

Intel IPP functions described in this section implement building blocks that can be used to create speech codecs compliant with the ITU-T Recommendation G.729 (see [ITU729], [ITU729A], [ITU729B]).

These functions are used in the Intel IPP *G.729 Speech Encoder-Decoder* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

The list of these functions is given in the following table:

Table 9-2 Intel IPP G.729 Related Functions

Function Base Name	Operation
Basic Functions	
DotProd_G729	Computes the dot product of two vectors.
Interpolate_G729	Computes the weighted sum of two vectors.
Linear Prediction Analysis Functions	
AutoCorr_G729	Estimates the auto-correlation of a vector.
LevinsonDurbin_G729	Calculates LP coefficients from the autocorrelation coefficients.
LPCToLSP_G729	Converts LP coefficients to LSP coefficients.
LSFToLSP_G729	Converts Line Spectral Frequencies to LSP coefficients.
LSFQuant_G729	Performs quantization of LSF coefficients.
LSFDecode_G729	Decodes quantized LSFs.
LSFDecodeErased_G729	Reconstructs quantized LSFs in case when a frame is erased.
LSPToLPC_G729	Converts LSP coefficients to LP coefficients.
LSPQuant_G729	Quantizes the LSP coefficients.
LSPToLSF_G729	Converts LSP coefficients to LSF coefficients.
LagWindow_G729	Applies 60Hz bandwidth expansion.
Codebook Search Functions	
OpenLoopPitchSearch_G729	Searches for an optimal pitch value.
AdaptiveCodebookSearch_G729	Searches for the integer delay and the fraction delay, and computes the adaptive vector.
DecodeAdaptiveVector_G729	Restores the adaptive codebook vector by interpolating the past excitation.
FixedCodebookSearch_G729	Searches for the fixed codebook vector.

Table 9-2 Intel IPP G.729 Related Functions (continued)

Function Base Name	Operation
ToeplizMatrix_G729	Calculates 616 elements of the Toepliz matrix for the fixed codebook search.
Codebook Gain Functions	
DecodeGain_G729	Decodes the adaptive and fixed-codebook gains.
GainControl_G729	Calculates adaptive gain control.
GainQuant_G729	Quantizes the codebook gain using a two-stage conjugate-structured codebook.
AdaptiveCodebookContribution_G729	Updates target vector for codebook search by subtracting adaptive codebook contribution.
Filter Functions	
ResidualFilter_G729	Implements an inverse LP filter and obtains the residual signal.
SynthesisFilter_G729	Reconstructs the speech signal from LP coefficients and residuals.
LongTermPostFilter_G729	Restores the long-term information from the old speech signal.
ShortTermPostFilter_G729	Restores speech signal from the residuals.
TiltCompensation_G729	Compensates for the tilt in the short-term filter.
HarmonicFilter	Calculates the harmonic filter.
HighPassFilterSize_G729	Returns the G729 high-pass filter size.
HighPassFilterInit_G729	Initializes the high-pass filter.
HighPassFilter_G729	Performs G729 high-pass filtering.
IIR16s_G729	Performs IIR filtering.
PhaseDispersionGetStateSize_G729D	Queries the memory length of the phase dispersion filter.
PhaseDispersionInit_G729D	Initializes the phase dispersion filter memory.
PhaseDispersionUpdate_G729D	Updates the phase dispersion filter state.
PhaseDispersion_G729D	Performs the phase dispersion filtering.
Preemphasize_G729A	Computes pre-emphasis of a post filter.
WinHybridGetStateSize_G729E	Queries the length of the hybrid windowing module memory.
WinHybridInit_G729E	Initializes the hybrid windowing module memory.
WinHybrid_G729E	Applies the hybrid window and computes autocorrelation coefficients.
RandomNoiseExcitation_G729B	Initializes a random vector with a Gaussian distribution.
FilteredExcitation_G729	Computes filtered exitation.

Basic Functions

These functions may be applied in both the encoding and decoding process.

DotProd_G729

Computes the dot product of two vectors.

Syntax

```
IppStatus ippsDotProd_G729A_16s32s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    int len, Ipp32s* pDp);
IppStatus ippsDotProd_G729A_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int
    len, Ipp32f* pDp);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
len	Number of elements in each vector.
pDp	Pointer to the output result.

Description

The functions `ippsDotProd_G729A` are declared in the `ippsc.h` file. The function `ippsDotProd_G729A_16s32s` computes the dot product of two source vectors *pSrc1* and *pSrc2* as:

$$pDp = 2 \cdot \sum_{i=0}^{len/2} pSrc1[2 \cdot i] \cdot pSrc2[2 \cdot i],$$

while for the `ippsDotProd_G729A_32f` function the result is

$$pDp = \sum_{i=0}^{len/2} pSrc1[2 \cdot i] \cdot pSrc2[2 \cdot i]$$

Vectors *pSrc1* and *pSrc2* must have the same length, *len*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pDp* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsOverflow	Indicates a warning that at least one result value was saturated.

Interpolate_G729

Computes the weighted sum of two vectors.

Syntax

```
IppStatus ippsInterpolate_G729_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
IppStatus ippsInterpolateC_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s val1,
    const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsInterpolateC_NR_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s val1,
    const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsInterpolateC_G729_32f(const Ipp32f* pSrc1, Ipp32f val1, const
    Ipp32f* pSrc2, Ipp32f val2, Ipp32f* pDst, int len);
```

Parameters

pSrc1	Pointer to the first source vector.
val1	Factor to multiply to the first source vector.
pSrc2	Pointer to the second source vector.
val2	Factor to multiply to the second source vector.
pDst	Pointer to the destination (interpolated) vector.
len	Number of elements in the vectors.
scaleFactor	Scale factor for the destination vector.

Description

These functions are declared in the ippsc.h file. The function ippsInterpolate_G729_16s computes the weighted sum as:

$$pDst[i] = (pSrc1[i] + \mathrm{sign}(pSrc1[i])) >> 1 + (pSrc2[i] + \mathrm{sign}(pSrc2[i])) >> 1 \quad ,$$

for $i = 0, ..len-1$.

Functions `ippsInterpolateC_G729_16s_Sfs` and `ippsInterpolateC_NR_G729_16s_Sfs` both use the same formula to compute the weighted sum as:

$$pDst[i] = (val1 \cdot pSrc1[i] + val2 \cdot pSrc2[i]) >> scaleFactor \ , \ i = 0, ..len-1 .$$

However, these functions differ in the rounding mode they use for output results (see Rounding mode).

The function `ippsInterpolateC_G729_32f` computes the weighted sum as:

```
pDst(i) = val1*pSrc1(i) + val2*pSrc2(i) , 0 <= i < len
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsScaleRangeErr`	Indicates an error when *scaleFactor* is less than 0.

Linear Prediction Analysis Functions

Functions described in this section implement LSP coding (quantization) and decoding, as well as transformation between LPC, LSP and LSF coefficients.

AutoCorr_G729

Estimates the auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorr_G729B(const Ipp16s*  pSrcSpch,
    Ipp16s*  pResultAutoCorrExp, Ipp32s*  pDstAutoCorr);
```

Parameters

pSrcSpch	Pointer to the input speech signal vector [240].
pResultAutoCorrExp	Pointer to the exponent of autocorrelation coefficients.
pDstAutoCorr	Pointer to the autocorrelation coefficients vector [13].

Description

The function `ippsAutoCorr_G729` is declared in the `ippsc.h` file. This function calculates a set of 11 autocorrelation coefficients and their exponent for the input speech signal. The function is applied to the vector of 240 speech samples, which include 120 samples from past speech frames, 80 samples from the present speech frame, and 40 samples from the future frame. The functionality is as follows.

1. First, apply to speech samples the asymmetric windows given by:

$$w_{1p}(n) = \begin{cases} 0.54 - 0.46\cos\left(\frac{2n\pi}{399}\right), & n = 0, 1, \ldots, 199 \\ \cos\left(\frac{2(n-200)\pi}{159}\right), & n = 200, 201, \ldots, 239 \end{cases}$$

2. Next, calculate autocorrelations of the windowed speech samples $s(i)$, $i = 0, 1, \ldots, 239$, using the formula:

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k), \quad k = 0, 1, \ldots, 11$$

3. Finally, the autocorrelation coefficients are scaled according to the first autocorrelation coefficient (energy) value. Specifically, autocorrelation coefficients are multiplied by the factor 2^{norm}, where *norm* ≤0 is calculated so as to make the first coefficient normalized. The corresponding scaling factor, *norm*, is returned via *pResultAutoCorrExp*.

NOTE. The function ippsAutoCorr_G729B is actually a combination of ippsMul_NR and ippsAutoCorr_NormE functions. The following code details the correspondence.

```
{
        short sig_win[240];
        ippsMul_NR_16s_Sfs(pSrcSpch,window,sig_win,240,15);
        ippsAutoCorr_NormE_16s32s(sig_win,240,pDstAutoCorr,11,
        &pResultAutoCorrExp);
}
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcSpch*, *pResultAutoCorrExp* or *pDstAutoCorr* pointer is NULL.

LevinsonDurbin_G729

Calculates LP coefficients from the autocorrelation coefficients.

Syntax

```
IppStatus ippsLevinsonDurbin_G729_32s16s(const Ipp32s* pSrcAutoCorr, int order,
    Ipp16s* pDstLPC, Ipp16s* pDstRc, Ipp16s* pResultResidualEnergy);
IppStatus ippsLevinsonDurbin_G729_32f(const Ipp32f* pSrcAutoCorr, int order,
    Ipp32f* pDstLpc, Ipp32f* pDstRc, Ipp32f* pResultResidualEnergy);
IppStatus ippsLevinsonDurbin_G729B(const Ipp32s*  pSrcAutoCorr, Ipp16s*
    pDstLPC, Ipp16s*  pDstRC, Ipp16s*  pResultResidualEnergy);
```

Parameters

order	The LP order.
pSrcAutoCorr	Pointer to the autocorrelation coefficients vector [*order*+1].
pDstLPC	Pointer to the output LP coefficients [*order*+1].
pDstRC	Pointer to the output reflection coefficients vector [*order*].
pResultResidualEnergy	Pointer to the residual energy.

Description

The functions `ippsLevinsonDurbin_G729` and `ippsLevinsonDurbin_G729B` are declared in the `ippsc.h` file. These functions calculate Linear Prediction (LP) coefficients of the LP filter with the given order from the autocorrelation coefficients, using Levinson-Durbin algorithm. The function `ippsLevinsonDurbin_G729` uses the parameter *order*, while the function `ippsLevinsonDurbin_G729B` operates for the default *order* = 10.

To obtain LP coefficients a_i, i = 1, 2, ..., *order*, the following set of equations is to be solved:

$$\sum_{i=1}^{order} a_i \times r(|i-k|) = -r(k) \quad , \; k = 1,2,\ldots,order \; .$$

The functions perform the following steps:

1. Levinson-Durbin algorithm is applied to solve the above set of equations. This algorithm uses the following recursion.

$$E^{[0]} = r(o)$$

for i = 1 to *order*

$$a_0^{[i-1]} = 1$$

$$k_i = -\left[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j)\right] / \; E^{[i-1]}$$

$$a_i^{[i]} = k_i$$

for j = 1 to i-1

$$a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$$

end

$$E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$$

end

2. Set E as the output residual energy and k_i as reflection coefficients.

3. If the LPC filter used in this algorithm is unstable, that is some $|k_i|$ is very close to 1.0 during recursion, the elements of input vectors *pSrcDstLPC* and *pSrcDstRC* RC coefficients are not changed and the residual energy is set to 0.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcAutoCorr*, *pDstRC*, *pResultResidualEnergy*, or *pDstLPC* pointer is NULL.
ippStsSizeErr	Indicates an error when *order* is less than or equal to 0, or when *order* is greater than 30.

LPCToLSP_G729

Converts LP coefficients to LSP coefficients.

Syntax

```
IppStatus ippsLPCToLSP_G729_16s(const Ipp16s*  pSrcLPC, const Ipp16s*
    pSrcPrevLSP, Ipp16s*  pDstLSP);
IppStatus ippsLPCToLSP_G729A_16s(const Ipp16s*  pSrcLPC, const Ipp16s*
    pSrcPrevLSP, Ipp16s*  pDstLSP);
IppStatus ippsLPCToLSP_G729A_32f(const Ipp32f* pSrcLPC, const Ipp32f*
    pSrcPrevLsp, Ipp32f* pDstLSP);
IppStatus ippsLPCToLSP_G729_32f(const Ipp32f* pSrcLPC, const Ipp32f*
    pSrcPrevLsp, Ipp32f* pDstLSP);
```

Parameters

pSrcLPC	Pointer to the LP coefficients vector [11], in Q12 for 16s data.
pSrcPrevLSP	Pointer to the previous LSP coefficients vector [10], in Q15 for 16s data.

pDstLSP	Pointer to the computed LSP coefficients vector [10].

Description

These functions are declared in the `ippsc.h` file. They convert 10th order LP coefficients to LSP coefficients.
The first function `ippsLPCToLSP_G729` is designed for G.729/B codec while the second function `ippsLPCToLSP_G729A` is designed for G.729A codec.
Functions perform the following steps:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the following recursive relations:

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i) \ , \ i = 0,1,\ldots,4 \ ,$$

where $f_1(0) = f_2(0) = 1.0$.

2. Use Chebyshev polynomials to evaluate $F_1(z)$ and $F_2(z)$. The Chebyshev polynomials are given by.

$$C_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2$$

$$C_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2$$

For G.729/B, evaluate $F_1(z)$ and $F_2(z)$ at 60 points equally spaced between 0 and π and check for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided four times to track the root.
For G.729A, evaluate $F_1(z)$ and $F_2(z)$ at 50 points equally spaced between 0 and π and check for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided two times to track the root.

3. If all 10 roots needed to determine LSP coefficients are not found, just use the previous set of LSP coefficients.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLPC*, *pSrcPrevLSP* or *pDstLSP* pointer is `NULL`.

LSFToLSP_G729

Converts Line Spectral Frequencies to LSP coefficients.

Syntax

```
IppStatus ippsLSFToLSP_G729_16s (const Ipp16s* pLSF, Ipp16s* pLSP);
```

Parameters

pLSF	Pointer to the LSF vector [10], in Q13 in the range [0,π].
pLSP	Pointer to the LSP vector [10], in Q15 in the range[-1;1].

Description

The function `ippsLSPToLSF_G729` is declared in the `ippsc.h` file. This function converts the Line Spectral Frequencies (LSF) to the LSP coefficients as given by:

$pLSP[i] = \cos(pLSF[i])$, $i = 1,\ldots, 10$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pLSP* or *pLSF* pointer is `NULL`.
`ippStsOverflow`	Indicates a warning that at least one result value was saturated.

LSFQuant_G729

Performs quantization of LSF coefficients.

Syntax

```
IppStatus ippsLSFQuant_G729_16s (const Ipp16s* pLSF, Ipp16s* pQuantLSFTable,
    Ipp16s* pQuantLSF, Ipp16s* quantIndex);
IppStatus ippsLSFQuant_G729B_16s (const Ipp16s* pLSF, Ipp16s* pQuantLSFTable,
    Ipp16s* pQuantLSF, Ipp16s* quantIndex);
IppStatus ippsLSFQuant_G729B_32f (const Ipp32f* pLSF, Ipp32f* pQuantLSFTable,
    Ipp32f* pQuantLSF,  int*  quantIndex);
```

Parameters

pLSF	Pointer to the LSF vector [10], in Q13 in the range [0,π] for `16s` data.
pQuantLSFTable	Pointer to a matrix of 4 rows and 10 columns, containing previously quantized LSFs.
pQuantLSF	Pointer to the quantized LSF vector, in Q13 in the range [0,π] for `16s` data.
quantIndex	Pointer to the output combined codebook indices *L*0, *L*1, *L*2, and *L*3.

Description

These functions are declared in the `ippsc.h` file.
The function `ippsLSFQuant_G729_16s` quantizes the difference between the input LSF coefficients and previously quantized LSF coefficients, using the switched Moving Average (MA) predictor.
Quantization is performed using a two-stage Vector Quantizer (VQ): 10-dimensional VQ using codebook **L**1 (128 entries), and 10-bit split VQ (5 dimensions each) on two codebooks, **L**2 and **L**3 (32 entries each).

The function `ippsLSFQuant_G729B_16s` quantizes the LSF coefficients using a two-stage split VQ, (in 5 and 4 bits). The second order MA predictor used in the SID-LPC quantization procedure is calculated as a linear interpolation of the first and second MA predictors, with weight values 0.6 and 0.4, respectively. The first stage of quantization is the same as that in G.729, but only part of the quantization table (32 entries) is used. The second stage is different. No splitting is done and also only portion of the second table (16 entries) is used.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pLSF*, *pQuantLSFTable*, *pQuantLSF*, or *quantIndex* pointer is `NULL`.
`ippStsLSFLow`	Indicates a warning that the corresponding filter has low stability.
`ippStsLSFHigh`	Indicates a warning that the corresponding filter has high stability.
`ippStsLSFLowAndHigh`	Indicates a warning that the corresponding filter has both low and high stability.

LSFDecode_G729

Decodes quantized LSFs.

Syntax

```
IppStatus ippsLSFDecode_G729_16s (const Ipp16s* quantIndex, Ipp16s*
    pQuantLSFTable, Ipp16s* pQuantLSF);
IppStatus ippsLSFDecode_G729B_16s(const Ipp16s* quantIndex, Ipp16s*
    pQuantLSFTable, Ipp16s*  pQuantLSF);
IppStatus ippsLSFDecode_G729B_32f (const int* quantIndex, Ipp32f*
    pQuantLSFTable, Ipp32f* pDstQLsp);
IppStatus ippsLSFDecode_G729_32f (const int* quantIndex, Ipp32f*
    pQuantLSFTable, Ipp32f* pQuantLSF);
```

Parameters

quantIndex	Pointer to the vector of indices: *L*0, *L*1, *L*2, *L*3 (see Table 8/G.729 in [ITU729]) or *L*0, *L*1, *L*2 (see also [ITU729B] , section 4.3) .
pQuantLSFTable	Pointer to the input/output table [4][10] of 4 previously quantized LSFs vectors, in Q13 for `16s` data.
pQuantLSF	Pointer to the quantized LSF output vector [10], in Q13 for `16s` data.

Description

These functions are declared in the `ippsc.h` file.

The function `ippsLSFDecode_G729` retrieves the quantized LSF coefficients from the quantization table and indices.

The function `ippsLSFDecode_G729B` retrieves the quantized LSF coefficients for the SID frame. Only *L*0, *L*1, *L*2 indices are used.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *quantIndex* , *pQuantLSFTable*, or *pQuantLSF* pointer is `NULL`.
`ippStsLSFLow`	Indicates a warning of only low stability of LSF.
`ippStsLSFHigh`	Indicates a warning of only high stability of LSF.

`ippStsLSFLowAndHigh` Indicates a warning of both low and high stability of LSF.

LSFDecodeErased_G729

Reconstructs quantized LSFs in case when a frame is erased.

Syntax

```
IppStatus ippsLSFDecodeErased_G729_16s (Ipp16s maIndex, Ipp16s* pQuantLSFTable,
    Ipp16s* pQuantLSF);
IppStatus ippsLSFDecodeErased_G729_32f (int maIndex, Ipp32f* pQuantLSFTable,
    const Ipp32f* pSrcPrevLSF);
```

Parameters

maIndex	Switched MA predictor (*L*0) .
pQuantLSFTable	Pointer to the input/output table [4][10] of 4 previously quantized LSFs vectors, in Q12 for `16s` data.
pQuantLSF	Pointer to the quantized LSF output vector [10], in Q12 for `16s` data.

Description

The function `ippsLSFDecodeErased_G729` is declared in the `ippsc.h` file. This function retrieves the quantized LSF coefficients through the previously decoded switched MA-predictor index (*L*0).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pQuantLSFTable* or *pQuantLSF* pointer is `NULL`.
`ippStsOutOfRangeErr`	Indicates an error when *maIndex* < 0 or *maIndex* ≥ `MA_NP`.

LSPToLPC_G729

Converts LSP coefficients to LP coefficients.

Syntax

```
IppStatus ippsLSPToLPC_G729_16s(const Ipp16s*  pSrcLSP, Ipp16s*  pDstLPC);
IppStatus ippsLSPToLPC_G729_32f(const Ipp32f* pSrcLsp, Ipp32f* pDstLpc);
```

Parameters

pSrcLSP	Pointer to the LSP coefficients vector [10], in Q15 for `16s` data.
pDstLPC	Pointer to the LP coefficients vector [11], in Q12 for `16s` data.

Description

The function `ippsLSPToLPC_G729` is declared in the `ippsc.h` file. This function converts a set of 10th order LSP coefficients to LP coefficients.
It performs the following steps:

1. Calculates the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursive relations:

```
for i = 1 to 5
```

$$f_1(i) = -2q_{2i-1} \times f_1(i-1) + 2f_1(i-2)$$

```
    for j = i-1 down to 1
```

$$f_1^{[i]}(j) = f_1^{[i-1]}(j) - 2q_{2i-1} \times f_1^{[i-1]}(j-1) + f_1^{[i-1]}(j-2)$$

```
    end
end
```

Here the initial values are set to $f_1(0) = 1$ and $f_1(-1) = 0$.
The coefficients $f_2(i)$ are computed similarly by replacing q_{2i-1} by q_{2i} .

2. $F_1(z)$ and $F_2(z)$ are then multiplied by $1 + z^{-1}$ and $1 - z^{-1}$ respectively to obtain $F_1'(z)$ and $F_2'(z)$ as given by:

$$f_1'(i) = f_1(i) + f_1(i-1) \ , \ i = 1,2,\ldots 5$$

$$f_2'(i) = f_2(i) - f_2(i-1) \ , \ i = 1,2,\ldots 5$$

3. Finally, the function computes the LP coefficients from $f_1'(i)$ and $f_2'(i)$ as:

$$a_i = \begin{cases} 0.5 \times f_1'(i) + 0.5 \times f_2'(i) \ , \ i = 1, 2, .., 5 \\ 0.5 \times f_1'(11-i) - 0.5 \times f_2'(11-i) \ , \ i = 6, 7, .., 10 \end{cases}$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcLSP* or *pDstLPC* pointer is NULL.

LSPQuant_G729

Quantizes the LSP coefficients.

Syntax

```
IppStatus ippsLSPQuant_G729_16s(const Ipp16s*  pSrcLSP, Ipp16s*
    pSrcDstPrevFreq, Ipp16s*  pDstQLSP, Ipp16s*  pDstQLSPIndex);
IppStatus ippsLSPQuant_G729E_32f (const Ipp32f*  pSrcLSP,  Ipp32f*
    pSrcDstPrevFreq, Ipp32f* pDstQLSF, Ipp32f*  pDstQLSP, int*  pDstQLspIndex);
```

Parameters

pSrcLSP	Pointer to the LSP coefficients vector [10], in Q15 for 16s data.
pSrcDstPrevFreq	Pointer to the four previous and updated quantized frequencies vector [40], in Q13 for 16s data. Elements 0 to 9 are for the newest frame and elements 30 to 39 are for the oldest frame.
pDstQLSF	Pointer to the LSF coefficients vector [10].
pDstQLSP	Pointer to the LSP coefficients vector [10], in Q15 for 16s data.
pDstQLSPIndex	Pointer to the combined codebook indices *L*0, *L*1, *L*2 and *L*3, of length 2. Its first element represents indices *L*0 and *L*1 in the following combination: *L*1 occupies the first 7 bits starting from LSB, whereas *L*0 resides in the next 1 bit adjacent to *L*1. The second element represents indices *L*2 and *L*3 in the following combination: *L*3 occupies the first 5 bits starting from LSB, and *L*2 resides in the next 5 bits adjacent to *L*3.

Description

The functions ippsLSPQuant_G729 are declared in the ippsc.h file. These functions obtain the quantized LSP coefficients and the codebook indices. They perform the following operations:

1. Converts the LSP coefficients q_i to LSF coefficients ω_i in the normalized frequency domain [0, π] as:

$$\omega_i = \arccos(q_i) \ , \ i = 1,\ldots,10$$

2. Uses a switched 4th order MA prediction to predict the LSF coefficients of the current frame. The difference between the computed and predicted coefficients is quantized using a two-stage vector quantizer. The first stage is a 10-dimensional VQ using codebook **L**1 with 128 entries (7 bits). The second stage is a 10 bit VQ which was implemented as a split VQ using two 5-dimensional codebooks, **L**2 and **L**3, containing 32 entries (5 bits) each. To explain the quantization process, it is convenient to first describe the decoding process. Each coefficient is obtained from the sum of two codebooks:

$$\hat{l} = \begin{cases} \mathbf{L}1_i(L1) + \mathbf{L}2_i(L2) \ , \ i = 1, \ldots, 5 \\ \mathbf{L}1_i(L1) + \mathbf{L}3_{i-5}(L3) \ , \ i = 6, \ldots, 10 \end{cases}$$

where $L1$, $L2$ and $L3$ are the codebook indices.

3. The vector to be quantized for the current frame *m* is obtained from:

$$l_i = \left[\omega_i^{(m)} - \sum_{k=1}^{4} \hat{p}_{i,k} \hat{l}_i^{(m-k)} \right] \Big/ \left(1 - \sum_{k=1}^{4} \hat{p}_{i,k} \right) \ , \ i = 1,\ldots,10 \ ,$$

where $\hat{p}_{i,k}$ are the coefficients of the switched MA predictor. The first codebook **L**1 is searched and the entry $L1$ that minimizes the (unweighted) mean-squared error is selected. This is followed by a search of the second codebook **L**2, which defines the lower part of the second stage. To avoid sharp resonances in the quantized LP synthesis filter, the partial vector $\hat{l}_i$, i = 1,…,5, is rearranged such that adjacent coefficients have a minimum distance of *J*. The rearrangement procedure is shown below:

```
for i = 2,…,10
     if l̂_{i-1} > l̂_i - j

          l̂_{i-1} = (l̂_i + l̂_{i-1} - j)/ 2
          l̂_i = (l̂_i + l̂_{i-1} + j)/ 2
      end
end
```

where J is 0.0012 for the first pass. Using the selected first stage vector $L1$ and the lower part of the second stage $L2$, the higher part of the second stage is searched from the codebook **L**3. Again, the rearrangement procedure is used to guarantee a minimum distance of 0.0012. The resulting vector $\hat{l}_i$, i = 1, ...,10 , is rearranged to guarantee a minimum distance of 0.0006. The quantized LSF coefficients $\hat{\omega}_i^{(m)}$ for the current frame m are obtained from the weighted sum of previous quantizer outputs $\hat{l}_i^{(m-k)}$, and the current quantizer output $\hat{l}_i^{(m)}$:

$$\hat{\omega}_i^{(m)} = \left(1 - \sum_{k=1}^{4} \hat{p}_{i,k}\right)\hat{l}_i^{(m)} + \sum_{k=1}^{4} \hat{p}_{i,k}\hat{l}_i^{(m-k)} \quad , \quad i = 1,\ldots,10$$

There are two MA predictors. Which MA predictor to use is defined by a single bit $L0$. It is determined by the one that minimizes the weighted mean-squared error:

$$E_{lsf} = \sum_{i=1}^{10} w_i(\omega_i - \hat{\omega}_i)^2$$

The weights w_i are made adaptive as a function of the unquantized LSF coefficients,

$$w_1 = \begin{cases} 1.0 & , \; if \;\; \omega_2 - 0.04\pi - 1 > 0 \\ 10(\omega_2 - 0.04\pi - 1)^2 + 1 , & otherwise \end{cases}$$

$$w_i = \begin{cases} 1.0 & , \; if \;\; \omega_{i+1} - \omega_{i-1} - 1 > 0 \\ 10(\omega_{i+1} - \omega_{i-1} - 1)^2 + 1 , & otherwise \end{cases} \quad , \; 2 \le i \le 9$$

$$w_{10} = \begin{cases} 1.0 & , \; if \;\; -\omega_9 + 0.92\pi - 1 > 0 \\ 10(-\omega_9 + 0.92\pi - 1)^2 + 1 , & otherwise \end{cases}$$

In addition, the weights w_5 and w_6 are multiplied by 1.2 each.

4. To avoid sharp resonance in the LP filter, the rearrangement procedure is applied twice to the resulting vector $\hat{l}_i$ with J = 0.0012 and 0.0006 sequentially.

5. After computing $\hat{\omega}_i$, the corresponding filter is checked for stability. This is done as follows:

1) Order the coefficients $\hat{\omega}_i$ in increasing value;

2) If $\hat{\omega}_i$ < 0.005 then $\hat{\omega}_i$ = 0.005;

3) If $\hat{\omega}_{i+1} - \hat{\omega}_i < 0.0391$ then $\hat{\omega}_{i+1} = \hat{\omega}_i + 0.0391$, i = 1, ...,9;

4) If $\hat{\omega}_{10} > 3.315$ then $\hat{\omega}_{10} = 3.315$.

6. Convert the quantized LSF vectors to LSP vectors.

NOTE. The function `ippsLSPQuant` is actually a combination of `ippsLSPToLSF`, `ippsLSFQuant`, and `ippsLSFToLSP` functions. The following code details the correspondence.

```
IppStatus ippsLSPQuant_G729_16s(const Ipp16s*  pSrcLSP, Ipp16s*
pSrcDstPrevFreq, Ipp16s*  pDstQLSP, Ipp16s*  pDstQLSPIndex) {
   __ALIGN(8) short lsf[LP_ORDER];
   __ALIGN(8) short lsp_q[LP_ORDER];
   short q_index[4];
   IppStatus sts=ippsLSPToLSF_G729_16s(pSrcLSP, lsf);
   if (sts != ippStsNoErr) return sts;
   sts = ippsLSFQuant_G729_16s(lsf, pSrcDstPrevFreq, lsf_q, q_index );
   if(sts != ippStsNoErr) return sts;
   pDstQLSPIndex[0] = (q_index[0]<<G729_NC0_B) | q_index[1];
   pDstQLSPIndex[1] = (q_index[2]<<G729_NC1_B) | q_index[3];
   ippsLSFToLSP_G729_16s(lsp_q, pDstQLSP)
   return ippStsNoErr;
}
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLSP*, *pSrcDstPrevFreq*, *pDstQLSP*, *pDstQLSF* or *pDstQLSPIndex* pointer is `NULL`.

LSPToLSF_G729

Converts LSP coefficients to LSF coefficients.

Syntax

```
IppStatus ippsLSPToLSF_Norm_G729_16s (const Ipp16s* pLSP, Ipp16s* pLSF);
```

```
IppStatus ippsLSPToLSF_G729_16s (const Ipp16s* pLSP, Ipp16s* pLSF);
```

Parameters

pLSP	Pointer to the LSP vector, in Q15 in the range [-1:1].
pLSF	Pointer to the LSF vector. Values must be scaled by 13 bits in the range [0:π] (for `ippsLSPToLSF_G729_16s` function) or by 15 bits in the range [0:0.5] (for `ippsLSPToLSF_Norm_G729_16s` function).

Description

These functions are declared in the `ippsc.h` file.
The function `ippsLSPToLSF_Norm_G729` converts the LSP coefficients to LSF coefficients as follows:

$$pLSF[i] = 2^{scaleFactor} \cdot \arccos(pLSP[i]) \quad , 0 \le i < 10 .$$

The scale factor is chosen such that the first LSF coefficient is normalized to the interval [0:0.5] by multiplying to 1/π.
The function `ippsLSPToLSF_G729` retains the result within the interval [0:π]. Both functions use the same approximation table to calculate the inverse cosine.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pLSP* or *pLSF* pointer is `NULL`.

LagWindow_G729

Applies 60Hz bandwidth expansion.

Syntax

```
IppStatus ippsLagWindow_G729_32s_I (Ipp32s* pSrcDst, int len);
```

Parameters

pSrcDst	Pointer to the autocorrelation vector.
len	Number of elements in the vector.

Description

The function `ippsLagWindow_G729` is declared in the `ippsc.h` file. This function applies a 60Hz bandwidth expansion to the autocorrelation vector as follows:

$$r[0] = 1.0001 \cdot r[0]$$

$$r[i] = w_{lag}[i] \cdot r[i] \ , \ i = 0, \ldots len-1 \ ,$$

where

$$w_{lag}[i] = \exp\left(-\frac{1}{2} \cdot \left(2\pi f_0 \cdot \frac{i}{f_s}\right)^2\right)$$

and

f_0 = 60Hz, f_s = 8000Hz.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsRangeErr`	Indicates an error when *len* > 12.

Codebook Search Functions

These functions perform open loop pitch estimation using the adaptive codebook, and search in ACELP excitation (fixed) codebook for optimal signs and positions of the pulses.

OpenLoopPitchSearch_G729

Searches for an optimal pitch value.

Syntax

```
IppStatus ippsOpenLoopPitchSearch_G729_16s(const Ipp16s*  pSrc, Ipp16s*
    bestLag);
IppStatus ippsOpenLoopPitchSearch_G729A_16s(const Ipp16s*  pSrc, Ipp16s*
    bestLag);
IppStatus ippsOpenLoopPitchSearch_G729A_32f(const Ipp32f* pSrc, Ipp16s*
    bestLag);
```

Parameters

pSrc	Pointer to the perceptually weighted speech signal vector [170].
bestLag	Pointer to the open-loop pitch search result.

Description

These functions are declared in the `ippsc.h` file. They use an open-loop method to get the pitch value. To reduce the complexity of the search for the best adaptive-codebook delay, the search range is limited around a candidate delay, which is obtained from maximizing the correlation coefficients of the perceptually weighted speech.

The function `ippsOpenLoopPitchSearch_G729A` is designed for G.729A codec and implements the following algorithm:

1. Overflow and underflow check:

$$nSumTemp = \sum_{j=-71}^{40} sw(2j-1) \times sw(2j-1) \;;$$

if $nSumTemp \geq 2^{30}$, overflow happens, $sw'(i) = sw(i) >> 3$, $i = -143,\ldots,79$;

else if $nSumTemp < 2^{19}$, underflow happens, $sw'(i) = sw(i) << 3$, $i = -143,\ldots,79$;

else $sw'(i) = sw(i)$, $i = -143,\ldots,79$.

2. Compute the correlation:

$$R(k) = \sum_{n=0}^{39} sw'(2n) \times sw'(2n-k) \ , \ k = 20,\ldots,143$$

3. In the range [20, 39], find the maximum of $R(k)$, and optimal k_1. In the range [40, 79], find the maximum of $R(k)$, and optimal k_2. In the range [80, 143], among the even k, find the maximum of $R(k)$, and suboptimal k_3. Let $k_3' = k_3$.

4. if $R(k_3' + 1) > R(k_3)$, then $k_3 = k_3' + 1$.

5. if $R(k_3' - 1) > R(k_3)$, then $k_3 = k_3' - 1$.

6. For the three optimal k_i, $i = 1,2,3$, normalize the three peak values of correlation:

$$R'(k_i) = \frac{R(k_i)}{\sqrt{\sum_{n=0}^{39} sw'(2n-k_i) \times sw'(2n-k_i)}}$$

7. Post process $R'(k_1)$ and $R'(k_2)$:

$$\begin{aligned}
&if\ (|2 \times k_2 - k_3| < 5)\ \{ \\
&\quad R'(k_2) = R'(k_2) + 0.25 \times R'(k_3)\ ; \\
&\} \\
&if\ (|3 \times k_2 - k_3| < 7)\ \{ \\
&\quad R'(k_2) = R'(k_2) + 0.25 \times R'(k_3)\ ; \\
&\quad \} \\
&if\ (|2 \times k_1 - k_2| < 5)\ \{ \\
&\quad R'(k_1) = R'(k_1) + 0.2 \times R'(k_2)\ ; \\
&\} \\
&if\ (|3 \times k_1 - k_2| < 7)\ \{ \\
&\quad R'(k_1) = R'(k_1) + 0.2 \times R'(k_2)\ ; \\
&\}
\end{aligned}$$

8. Find the optimal delay k_{opt} from k_1 to k_3, which satisfies

$$R'(k_{opt}) = \max\{R'(k_i),\ i = 1,\ldots,3\}.$$

The function `ippsOpenLoopPitchSearch_G729` is designed for G.729/B codec and implements the following algorithm:

1. Overflow and underflow check:

$$nSumTemp = \sum_{i=-143}^{79} sw(i) \times sw(i)$$

if $nSumTemp \geq 2^{30}$, overflow happens, $sw'(i) = sw(i) >> 3$, i = -143,...,79;

else if $nSumTemp < 2^{19}$, underflow happens, $sw'(i) = sw(i) << 3$, i= -143,...,79;

else $sw'(i) = sw(i)$, i = -143,...,79 .

2. Compute the correlation:

$$R(k) = \sum_{i=0}^{79} sw'(i) \times sw'(i-k) \ , \ k = 20,...,143$$

3. In the range [20, 39], find the maximum of $R(k)$, and optimal $k = k_1$. In the range [40,79], find the maximum of $R(k)$, and optimal $k = k_2$. In the range [80,143], find the maximum of $R(k)$, and optimal $k = k_3$.

4. For the three optimal k_i, i = 1,2,3, normalize the three peak values of correlation:

$$R'(k_i) = \frac{R(k_i)}{\sqrt{\sum_{n=0}^{79} sw'(n-k_i) \times sw'(n-k_i)}}$$

2. Find the optimal delay k_{opt} from k_1 to k_3 , according to the following relations:

$$k_{opt} = k_1 , \quad R'(k_{opt}) = R'(k_1)$$

$$if\ (R'(k_2) \geq 0.85 R'(k_{opt})) \ , \ k_{opt} = k_2 , \quad R'(k_{opt}) = R'(k_2)$$

$$if\ (R'(k_3) \geq 0.85 R'(k_{opt})) \ , \ k_{opt} = k_3 , \quad R'(k_{opt}) = R'(k_3)$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *bestLag* pointer is `NULL`.

AdaptiveCodebookSearch_G729

Searches for the integer delay and the fraction delay, and computes the adaptive vector.

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_G729_16s(Ipp16s valOpenDelay, const
    Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
    pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s* pDstAdptVector, Ipp16s
    subFrame);
IppStatus ippsAdaptiveCodebookSearch_G729A_16s(Ipp16s valOpenDelay, const
    Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
    pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s* pDstAdptVector, Ipp16s
    subFrame);
IppStatus ippsAdaptiveCodebookSearch_G729D_16s (Ipp16s valOpenDelay, const
    Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
    pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s subFrame);
```

Parameters

valOpenDelay	Open-loop delay, in the range [18,145].
pSrcAdptTarget	Pointer to the target signal for adaptive-codebook search vector [40].
pSrcImpulseResponse	Pointer to the impulse response of weighted synthesis filter vector [40], in Q12.
pSrcDstPrevExcitation	Pointer to the previous and updated excitation vector [194].
pDstDelay	Pointer to the integer delay and fraction delay vector [2].
pDstAdptVector	Pointer to the adaptive vector [40].
subFrame	Subframe number, either 0 or 1.

Description

These functions are declared in the `ippsc.h` file. They search for the integer delay T and the fraction delay t, using the target signal and the past filtered excitation, and compute the adaptive vector. They are applied in subframes.

The function **ippsAdaptiveCodebookSearch_G729A** is designed for G.729A codec and implements the following algorithm:

1. Search for the optimal delay k, which maximizes the following expression:

$$R_N(k) = \sum_{n=0}^{39} x(n)y_k(n) = \sum_{n=0}^{39} x(n) \sum_{i=0}^{n} u_k(i)h(n-i) = \sum_{i=0}^{39} \sum_{n=i}^{39} u_k(i)x(n)h(n-i)$$

$$= \sum_{n=0}^{39} u_k(n)x_d(n), \quad k = t_{min}, \ldots, t_{max}$$

$$x_d(n) = \sum_{i=n}^{39} x(i)h(i-n) \ , \ n = 0,1,\ldots,39 \ .$$

Denote the optimal delay k as *T*.

2. If the search result *T* is less than 85 in the first subframe or it is in the second subframe, search for the fraction delay by maximizing the expression below:

$$R_{Nt}(k) = \sum_{n=0}^{39} x_d(n)u_{kt}(n) \ , \ t = -1,0,1 \ , \ k = T$$

$$u_{kt}(n) = \sum_{i=0}^{9} u_k(n-k_p-i)b_{30}(t_p+3i) + \sum_{i=0}^{9} u_k(n-k_p+1+i)b_{30}(3-t_p+3i) \ ,$$

$n = 0,1,\ldots,39$

$$k_p = \begin{cases} T \ , & t = 0, -1 \\ T+1 \ , & t = 1 \end{cases} \quad , \qquad t_p = \begin{cases} -t \ , & t = 0, -1 \\ 2 \ , & t = 1 \end{cases}$$

3. If the fraction result is *t*, then the new adaptive vector is computed as:

$$v(n) = u_{kt}(n) \ , \ n = 0,1,\ldots,39 \ .$$

The function **ippsAdaptiveCodebookSearch_G729** is designed for G.729/B codec and implements the following algorithm:

1. Search for the integer delay *k* which maximizes the following expression:

$$R(k) = \frac{\sum_{n=0}^{39} x(n) y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n) y_k(n)}}$$

where y_k is the past filter excitation at delay *k*. Denote the optimal *k* as *T*.

2. If the search result *T* is less than 85 in the first subframe or it is in the second subframe, search for the fraction delay *t* by maximizing the expression below:

$$R_t(T) = \sum_{i=0}^{3} R(T-i) b_{12}(t+3i) + \sum_{i=0}^{3} R(T+1+i) b_{12}(3-t+3i) \quad , \; t = 0,1,2 \; .$$

3. Finally, obtain the adaptive vector by interpolating the past excitation.
The interpolation method is the same as the method used to get the new adaptive vector in the `ippsAdaptiveCodebookSearch_G729A` function.

The function **ippsAdaptiveCodebookSearch_G729D** differs from `ippsAdaptiveCodebookSearch_G729` in that the number of lags in the second subframe has been reduced from 32 to 16.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcAdptTarget*, *pSrcImpulseResponse*, *pSrcDstPrevExcitation*, *pDstDelay*, or *pDstAdptVector* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *valOpenDelay* is not in the range [18, 145], or *subFrame* is not 0 or 1.

DecodeAdaptiveVector_G729

Restores the adaptive codebook vector by interpolating the past excitation.

Syntax

```
IppStatus ippsDecodeAdaptiveVector_G729_16s(const Ipp16s* pSrcDelay, Ipp16s*
    pSrcDstPrevExcitation, Ipp16s*  pDstAdptVector);
IppStatus ippsDecodeAdaptiveVector_G729_16s_I(const Ipp16s* pSrcDelay, Ipp16s*
    pSrcDstPrevExcitation);
IppStatus ippsDecodeAdaptiveVector_G729_32f_I(const Ipp32s* pSrcDelay, Ipp32f*
    pSrcDstPrevExcitation);
```

Parameters

pSrcDelay	Pointer to the integer and fraction delay for each subframe.
pSrcDstPrevExcitation	Pointer to the past and updated excitations vector [194].
pDstAdptVector	Pointer to the adaptive codebook vector [40].

Description

The function `ippsDecodeAdaptiveVector_G729` is declared in the `ippsc.h` file. This function restores the adaptive codebook vector by interpolating the past excitation. It performs the following steps:

1. Decodes the integer part and fractional part of the pitch delay *T* and *t*.
2. Interpolates the past excitation to get the current adaptive excitation as:

$$v(n) = \sum_{i=0}^{9} u(n-T-i) \times b_{30}(t+3i) + \sum_{i=0}^{9} u(n-T+1+i) \times b_{30}(3-t+3i) \ , \ n = 0,1,\ldots,39$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDelay*, *pSrcDstPrevExcitation*, or *pDstAdptVector* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *pSrcDelay*[0] is not in the range [18, 145], or *pSrcDelay*[1] is not -1, 0 or 1.

FixedCodebookSearch_G729

Searches for the fixed codebook vector.

Syntax

```
IppStatus ippsFixedCodebookSearch_G729_16s(const Ipp16s*  pSrcFixedCorr,
    Ipp16s*  pSrcDstMatrix, Ipp16s*  pDstFixedVector, Ipp16s*  pDstFixedIndex,
    Ipp16s*  pSearchTimes, Ipp16s subFrame);
IppStatus ippsFixedCodebookSearch_G729_32s16s(const Ipp16s*  pSrcFixedCorr,
    Ipp32s*  pSrcDstMatrix, Ipp16s*  pDstFixedVector, Ipp16s*  pDstFixedIndex,
    Ipp16s*  pSearchTimes, Ipp16s subFrame);
IppStatus ippsFixedCodebookSearch_G729A_16s(const Ipp16s*  pSrcFixedCorr,
    Ipp16s*  pSrcDstMatrix, Ipp16s*  pDstFixedVector, Ipp16s*  pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729A_32s16s(const Ipp16s*  pSrcFixedCorr,
    Ipp32s*  pSrcDstMatrix, Ipp16s*  pDstFixedVector, Ipp16s*  pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729E_16s(int mode, const Ipp16s*
    pSrcFixedTarget, const Ipp16s* pSrcLtpResidual, const Ipp16s*
    pSrcImpulseResponse, Ipp16s* pDstFixedVector, Ipp16s* pDstFltFixedVector,
    Ipp16s* pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729D_16s(const Ipp16s* pSrcFixedCorr, const
    Ipp16s* pSrcImpulseResponse, Ipp16s* pSrcDstMatrix, Ipp16s* pDstFixedVector,
    Ipp16s* pDstFltFixedVector, Ipp16s* pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729_32f (const Ipp32f* pSrcFixedCorr,
    Ipp32f* pSrcDstMatrix, Ipp32f* pDstFixedVector, Ipp32s* pDstFixedIndex,
    Ipp32s* pSearchTimes, Ipp32s subFrame);
IppStatus ippsFixedCodebookSearch_G729A_32f (const Ipp32f* pSrcFixedCorr,
    Ipp32f* pSrcDstMatrix, Ipp32f* pDstFixedVector, Ipp32s* pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729D_32f (Ipp32f* pSrcDstFixedCorr, Ipp32f*
    pSrcDstMatrix, const Ipp32f* pSrcImpulseResponse, Ipp32f* pDstFixedVector,
    Ipp32f* pDstFltFixedVector, Ipp32s* pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729E_32f (int mode, Ipp32f*
    pSrcDstFixedCorr, const Ipp32f* pSrcLtpResidual, const Ipp32f*
    pSrcImpulseResponse, Ipp32f* pDstFixedVector, Ipp32f* pDstFltFixedVector,
    Ipp32s* pDstFixedIndex);
```

Parameters

mode	Indicates forward (*mode* = 0) of backward (*mode* = 1) LP analysis mode.

pSrcFixedCorr	Pointer to the correlation signal of fixed-codebook search vector [40], in Q13.
pSrcDstMatrix	Pointer to the Toepliz matrix of length 616 (elements of `Ipp16s` type in Q9 or `Ipp32s` type in Q25).
pSrcFixedTarget	Pointer to the input updated target speech vector.
pSrcLtpResidual	Pointer to the input residual after long term prediction vector.
pSrcImpulseResponse	Pointer to the LP synthesis filter [40].
pDstFixedVector	Pointer to the fixed-codebook vector [40], in Q13.
pDstFltFixedVector	Pointer to the output filtered fixed-codebook vector.
pDstFixedIndex	Pointer to the fixed-codebook index. The first element is the sign codeword *S* and the second element is the position codeword *C*. Index length is the following: 2 - for all `G729`, all `G729A`, and `G729D_32f` flavors 4 - for all `G729E` flavors 1 - for `G729D_16s` flavor.
pSearchTimes	Pointer to the maximum search time for the remaining subframes.
subFrame	Subframe number, 0 or 1.

Description

These functions are declared in the `ippsc.h` file. They search for the fixed-codebook vector and corresponding vector index, respectively. They are applied in subframes.

The first function `ippsFixedCodebookSearch_G729` is designed for G.729/B codec and uses the nested-loop search approach. The Toepliz matrix is updated in this function. This function performs the following steps:

1. First, the 616 elements in the Toepliz matrix are updated as:

$$\Phi'(i, j) = \mathrm{sign}[d(i)]\mathrm{sign}[d(j)]\Phi(i, j)$$

where $\Phi(i, j)$ is the element of the Toepliz matrix, and $d(n)$ is the correlation signal.

2. Then, a threshold thr_3 is calculated using:

$$thr_3 = av_3 + 0.4(max_3 - av_3)$$

where max_3 and av_3 are the absolute maximum and absolute average of the first three pulses in $d(n)$.

3. Next, search for the fixed-codebook vector to minimize the term C^2/E , and obtain the vector index, where *C* is the correlation given by:

$$C = |d(m_0)| + |d(m_1)| + |d(m_2)| + |d(m_3)| \tag{9-1}$$

and *E* is the energy obtained by:

$$\begin{aligned} E = 2(&\Phi'(m_0, m_0) + \Phi'(m_1, m_1) + \Phi'(m_0, m_1) \\ &+ \Phi'(m_2, m_2) + \Phi'(m_0, m_2) + \Phi'(m_1, m_2) \\ &+ \Phi'(m_3, m_3) + \Phi'(m_0, m_3) + \Phi'(m_1, m_3) + \Phi'(m_2, m_3)) \end{aligned} \tag{9-2}$$

The search procedure is a kind of 4-loop search, and the last loop is entered only if the pre-computed correlation has exceeded the threshold thr_3. Also, the maximum number of times the loop can be entered is fixed to 180 so that a low percentage of the codebook is searched.

The sign codeword of the index, *S*, is obtained as follows.

$$S = s_0 + 2s_1 + 4s_2 + 8s_3 \tag{9-3}$$

The position codeword of the index, *C*, is computed as:

$$C = (m_0/5) + 8(m_1/5) + 64(m_2/5) + 512(2(m_3/5) + jx) \quad , \tag{9-4}$$

where $jx = 0$ if $m_3 = 3, 8, \ldots, 38$, and $jx = 1$ if $m_3 = 4, 9, \ldots, 39$.

4. Finally, the fixed-codebook vector is calculated.

The function `ippsFixedCodebookSearch_G729A` is designed for G.729A codec and uses the iterative depth-first, tree search approach.
The functionality is as follows:

1. First, calculate 616 elements of the new Toepliz matrix, $\Phi'(i, j)$, from the old Toepliz matrix Φ, using the formula:

$$\Phi'(i, j) = \mathrm{sign}[d(i)]\mathrm{sign}[d(j)]\Phi(i, j) \quad ,$$

where $\Phi(i, j)$ is the element of the Toepliz matrix, and *d*(*n*) is the correlation signal.

2. Next, search for the fixed-codebook vector to minimize the term C^2/E , and obtain the vector index, where *C* is the correlation given by (9-1), and *E* is the energy obtained as (9-2).

The search is an iterative depth-first, tree search approach.

The sign codeword of the index, *S,* is computed according to (9-3) and the position codeword of the index, *C*, is obtained as given by (9-4).

3. Finally, the fixed-codebook vector is calculated.

Both `ippsFixedCodebookSearch_G729` and `ippsFixedCodebookSearch_G729A` functions search four signed pulses in positions given by following table:

Pulses	Positions
i_0	0,5,10,15,20,25,30,35
i_1	1,6,11,16,21,26,31,36
i_2	2,7,12,17,22,27,32,37
i_3	3,8,13,18,23,28,33,38 4,9,14,19,24,29,34,39

The function `ippsFixedCodebookSearch_G729E` is designed for G.729E codec and searches ten signed pulses in five tracks in positions shown in the following table:

Pulses	Positions
i_0, i_1	0,5,10,15,20,25,30,35
i_3, i_4	1,6,11,16,21,26,31,36
i_5, i_6	2,7,12,17,22,27,32,37
i_7, i_8	3,8,13,18,23,28,33,38
i_9, i_{10}	4,9,14,19,24,29,34,39

The two pulses for each track may overlap resulting in a single pulse with duplicated amplitude.

The function `ippsFixedCodebookSearch_G729D` is designed for G.729D codec and searches for two signed pulses in two overlapping tracks given in following table:

Pulses	Positions
i_0	1, 3, 6, 8, 11, 13, 16, 18, 21, 23, 26, 28, 31, 33, 36, 38
i_1	0, 1, 2, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 16, 17, 19, 20, 21, 22, 24, 25, 26, 27, 29, 30, 31, 32, 34, 35, 36, 37, 39

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *`subFrame`* is not 0 or 1 (for integer functions only).

GainCodebookSearch

Finds pitch and fixed gains.

Syntax

```
IppStatus ippsGainCodebookSearch_G729_32f( const Ipp32f* pSrcCorrFactors,
    Ipp32f valPredictedCodebookGain, const int*  pSrcCand, Ipp32f*  pDstIdxs,
    Ipp32s valTimeFlag);
IppStatus ippsGainCodebookSearch_G729D_32f (const Ipp32f* pSrcCorrFactors,
    Ipp32f valPredictedCodebookGain, const int*  pSrcCand, Ipp32f*  pDstIdxs,
    Ipp32s valTimeFlag);
```

Parameters

pSrcCorrFactors	Pointer to the correlations.
valPredictedCodebookGain	Predicted codebook gain.
pSrcCand	Pointer to the pre-searched candidates.
pDstIdxs	Pointer to the output indexes.
valTimeFlag	Taming process indicator.

Description

As part of LSP quantization algorithm (refer to GainQuant_G729 for details), these functions operate in floating-point arithmetic and perform selection of the gain vector in the codebook *GA* and *GB* in the range limited by *pSrcCand[0]* and *pSrcCand[1].* The search algorithm consists of minimizing the following expression:

$$E = pSrcCorrFactors[0]\times G_p^2 + pSrcCorrFactors[2]\times G_c^2 + pSrcCorrFactors[1]\times G_p$$
$$+ pSrcCorrFactors[3]\times G_c + pSrcCorrFactors[4]\times G_p \times G_c$$

where G_p and G_c are optimum pitch and code gains, respectively, in selected domain.

The ippsGainCodebookSearch_G729_32f function uses the 7-bit codebook, whereas the function ippsGainCodebookSearch_G729D_32f uses the 6-bit codebook.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrcCorrFactors*, *pSrcCand*, or *pDstIdxs* pointer is NULL.
ippStsRangeErr	Indicates an error when *valTimeFlag* is not 0 or 1.
ippStsRangeErr	*For the G729 function flavor:* Indicates an error when *pSrcCand[0]* is not in the range [0,4) or *pSrcCand[0]* is not in the range [0,8). *For the G729D function flavor:* Indicates an error when *pSrcCand[0]* or *pSrcCand[1]* is not in the range [0,2).

ToeplizMatrix_G729

Calculates elements of the Toepliz matrix for the fixed codebook search.

Syntax

```
IppStatus ippsToeplizMatrix_G729_16s(const Ipp16s*  pSrcImpulseResponse,
    Ipp16s*  pDstMatrix);
IppStatus ippsToeplizMatrix_G729_16s32s(const Ipp16s*  pSrcImpulseResponse,
    Ipp32s*  pDstMatrix);
IppStatus ippsToeplizMatrix_G729_32f(const Ipp32f*  pSrcImpulseResponse,,
    Ipp32f*  pDstMatrix);
IppStatus ippsToeplizMatrix_G729D_32f(const Ipp32f*  pSrcImpulseResponse,,
    Ipp32f*  pDstMatrix);
```

Parameters

pSrcImpulseResponse	Pointer to the impulse response vector [40], in Q12 for 16s data.
pDstMatrix	Pointer to the elements of Toepliz matrix of length 616, (of Ipp16s type in Q9 or Ipp32s type in Q25).

Description

The function ippsToeplizMatrix_G729 is declared in the ippsc.h file. This function calculates 616 elements in Toepliz matrix for the fixed-codebook search. These elements can be expressed as:

$$\Phi(i,\ j) = \sum_{n=j}^{39} h(n-i) \times h(n-j)\ ,\ 0 \le i \le 39\ ,\ \ 0 \le j \le 39$$

where $h(i)$, i = 0, 1, ..., 39, is the impulse response.

The function stores the calculated 616 elements in *pDstMatrix* in the following order:

1. $\Phi(m_i,\ m_i)$ (i = 0, 1, 2, 3), 3x8+16 = 40 elements; starting location: 0.

   $\Phi(0,\ 0), \Phi(5,\ 5),\ \ldots,\ \Phi(35,\ 35)$, $\Phi(1,\ 1), \Phi(6,\ 6),\ \ldots,\ \Phi(36,\ 36)$,

   $\Phi(2,\ 2), \Phi(7,\ 7),\ \ldots,\ \Phi(37,\ 37)$, $\Phi(3,\ 3), \Phi(8,\ 8),\ \ldots,\ \Phi(39,\ 39)$.

2. $\Phi(m_0,\ m_1)$, 8x8 = 64 elements; starting location: 40.

   $\Phi(0,\ 1),\ \ldots,\ \Phi(0,\ 36)$, $\Phi(5,\ 1),\ \ldots,\ \Phi(5,\ 36)$, $\Phi(10,\ 1),\ \ldots,\ \Phi(35,\ 36)$.

3. $\Phi(m_0,\ m_2)$, 8x8 = 64 elements; starting location: 104.

   $\Phi(0,\ 2),\ \ldots,\ \Phi(0,\ 37)$, $\Phi(5,\ 2),\ \ldots,\ \Phi(5,\ 37)$, $\Phi(10,\ 2),\ \ldots,\ \Phi(35,\ 37)$.

4. $\Phi(m_0,\ m_3)$, 8x16 = 128 elements; starting location: 168.

   $\Phi(0,\ 3),\ \ldots,\ \Phi(0,\ 38)$, $\Phi(5,\ 3),\ \ldots,\ \Phi(5,\ 38)$, $\Phi(10,\ 3),\ \ldots,\ \Phi(35,\ 38)$.

   $\Phi(0,\ 4),\ \ldots,\ \Phi(0,\ 39)$, $\Phi(5,\ 4),\ \ldots,\ \Phi(5,\ 39)$, $\Phi(10,\ 4),\ \ldots,\ \Phi(35,\ 39)$.

5. $\Phi(m_1,\ m_2)$, 8x8 = 64 elements; starting location: 296.

   $\Phi(1,\ 2),\ \ldots,\ \Phi(1,\ 37)$, $\Phi(6,\ 2),\ \ldots,\ \Phi(6,\ 37)$, $\Phi(11,\ 2),\ \ldots,\ \Phi(36,\ 37)$.

6. $\Phi(m_1,\ m_3)$, 8x16 = 128 elements; starting location: 360.

   $\Phi(1,\ 3),\ \ldots,\ \Phi(1,\ 38)$, $\Phi(6,\ 3),\ \ldots,\ \Phi(6,\ 38)$, $\Phi(11,\ 3),\ \ldots,\ \Phi(36,\ 38)$.

   $\Phi(1,\ 4),\ \ldots,\ \Phi(1,\ 39)$, $\Phi(6,\ 4),\ \ldots,\ \Phi(6,\ 39)$, $\Phi(11,\ 4),\ \ldots,\ \Phi(36,\ 39)$.

7. $\Phi(m_2,\ m_3)$, 8x16 = 128 elements; starting location: 488.

   $\Phi(2,\ 3),\ \ldots,\ \Phi(2,\ 38)$, $\Phi(7,\ 3),\ \ldots,\ \Phi(7,\ 38)$, $\Phi(12,\ 3),\ \ldots,\ \Phi(37,\ 38)$.

   $\Phi(2,\ 4),\ \ldots,\ \Phi(2,\ 39)$, $\Phi(7,\ 4),\ \ldots,\ \Phi(7,\ 39)$, $\Phi(12,\ 4),\ \ldots,\ \Phi(37,\ 39)$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcImpulseResponse* or *pDstMatrix* pointer is NULL.

Codebook Gain Functions

These functions can be used to predict, quantize, decode, and control fixed- and adaptive-codebook gains both in G.729 encoding and decoding procedures.

DecodeGain_G729

Decodes the adaptive and fixed-codebook gains.

Syntax

```
IppStatus ippsDecodeGain_G729_16s (Ipp32s energy, Ipp16s* pPastEnergy, const
    Ipp16s* pQuaIndex, Ipp16s* pGain);
IppStatus ippsDecodeGain_G729I_16s(Ipp32s energy,  Ipp16s valGainAttenuation,
    Ipp16s* pPastEnergy, const Ipp16s* pQuaIndex, Ipp16s* pGain);
```

Parameters

energy	Input energy of the codeword.
pPastEnergy	Pointer to the input/output vector (in Q14) of the log-energies for fixed codebook contributions of the 4 previous subframes.
pQuaIndex	NULL for frame erasure; otherwise, pointer to the vector of codebook indices: *pQuaIndex* [0] - first stage codebook index , *pQuaIndex* [1] - second stage codebook index .
valGainAttenuation	Attenuation factor for the gains in case of frame erasuse (*pQuaIndex* = NULL).
pGain	Pointer to the input/output decoded gain: *pGain*[0] - adaptive (pitch) gain, *pGain*[1] - fixed codebook gain.

Description

The function ippsDecodeGain_G729 is declared in the ippsc.h file. This function decodes the adaptive and fixed-codebook gains. The fixed codebook gain g_c can be expressed as follows:

$g_c = \gamma \; g_c' \; ,$

where g_c' is a predicted gain based on previous fixed codebook energies, and γ is a correction factor. The predicted gain g_c' is obtained by predicting the log-energy of the current fixed-codebook contribution from the log-energy of the previous fixed-codebook contribution, using the 4th order MA predictor with coefficients [0.68, 0.58, 0.34, 0.19].

The adaptive codebook gain and the factor are vector-quantized using conjugate structured codebooks. The first element in each codebook represents the quantized adaptive-codebook gain and the second element represents the quantized fixed-codebook. In case of frame erasure, the gains are the attenuated versions of the previous gains.

ippsDecodeGain_G729_16s. The two-stage conjugate structured two-dimensional codebook is used: first stage in 3 bits and second stage in 4 bits. In case of frame erasure, factors 0.9 and 0.98 in Q15 are used for attenuation of the adaptive and fixed codebook gains, respectively.

ippsDecodeGain_G729I_16s. The new 6-bit conjugate structured codebook is used. In case of frame erasure, the factor given by *valGainAttenuation* is used for attenuation of both the adaptive and fixed codebook gains.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pPastEnergy* or *pGain* pointer is NULL.

GainControl_G729

Calculates adaptive gain control.

Syntax

```
IppStatus ippsGainControl_G729_16s_I (const Ipp16s* pSrc, Ipp16s* pSrcDst,
    Ipp16s* pGain);
IppStatus ippsGainControl_G729A_16s_I (const Ipp16s* pSrc, Ipp16s* pSrcDst,
    Ipp16s* pGain);
IppStatus ippsGainControl_G729_32f_I (Ipp32f gainScalingFactor, Ipp32f
    gainFactor, Ipp32f* pSrcDst, Ipp32f* pGain);
```

Parameters

pSrc	Pointer to the source reconstructed speech vector [40].
gainScalingFactor	Gain scaling factor, which is the ratio of energies of the reconstructed speech signal and post-filtered output signal.
gainFactor	Gain adjustment factor.
pSrcDst	Pointer to the input post-filtered signal and output gain-compensated signal vector [40].
pGain	Pointer to the output adaptive gain.

Description

These functions are declared in the `ippsc.h` file. The function `ippsGainControl_G729` compensates the gain difference between the reconstructed speech signal *sr*, given by *pSrc* , and the filtered signal *sf*, given by *pSrcDst*. First, the gain factor *G* is calculated as follows:

$$G = \frac{\sum_{i=0}^{39} |sr[i]|}{\sum_{i=0}^{39} |sf[i]|}$$

The output gain *spf* for the post-filtered signal *pSrcDst* is calculated as:

$$spf(n) = g^{(n)} \cdot sf(n), \quad n = 0,\ldots, 39 ,$$

where

$$g^{(n)} = 0.85 \cdot g^{(n-1)} + 0.15 \cdot G , \quad n = 0,\ldots, 39 , \quad g^{(-1)} = 1.0 .$$

The function returns $g^{(39)}$ in *pGain*.

For the function `ippsGainControl_G729A_16s`, the gain factor *G* and factors $g^{(n)}$ are calculated differently as given by:

$$G = \sqrt{\frac{\sum_{i=0}^{39} |sr[i]^2|}{\sum_{i=0}^{39} |spf[i]^2|}}$$

$$g^{(n)} = 0.9 \cdot g^{(n-1)} + 0.1 \cdot G$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSrcDst* or *pGain* pointer is NULL.

GainQuant_G729

Quantizes the codebook gain using a two-stage conjugate-structured codebook.

Syntax

```
IppStatus ippsGainQuant_G729_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
    pSrcFltAdptVector, const Ipp16s* pSrcFixedVector, const Ipp16s*
    pSrcFltFixedVector, Ipp16s* pSrcDstEnergyErr, Ipp16s* pDstQGain, Ipp16s*
    pDstQGainIndex, Ipp16s tameProcess);
IppStatus ippsGainQuant_G729D_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
    pSrcFltAdptVector, const Ipp16s* pSrcFixedVector, const Ipp16s*
    pSrcFltFixedVector, Ipp16s* pSrcDstEnergyErr, Ipp16s* pDstQGain, Ipp16s*
    pDstQGainIndex, Ipp16s tameProcess)
```

Parameters

pSrcAdptTarget	Pointer to the target signal for adaptive codebook search vector [40], in Q11.
pSrcFltAdptVector	Pointer to the filtered adaptive codebook vector [40], in Q11.
pSrcFixedVector	Pointer to the pre-weighted fixed-codebook vector [40], in Q12.
pSrcFltFixedVector	Pointer to the filtered fixed codebook vector [40], in Q12.
pSrcDstEnergyErr	Pointer to the previous predicted energy error vector [4], in Q10.
pDstQGain	Pointer to the quantized gain (G_p and G_c).
pDstQGainIndex	Pointer to the gain codebook indices, *GA* and *GB*.
tameProcess	Taming process indicator.

Description

The functions `ippsGainQuant_G729` and `ippsGainQuant_G729D` are declared in the `ippsc.h` file. These functions use a two-stage conjugate-structured codebook. The function `ippsGainQuant_G729_16s` quantizes the gain using the 7-bit codebook, whereas the function `ippsGainQuant_G729D_16s` uses the 6-bit codebook. The gain codebook search is done by minimizing the mean-squared weighted error between original and reconstructed speech:

$$E = \boldsymbol{x}^t\boldsymbol{x} + g_p^2\boldsymbol{y}^t\boldsymbol{y} + g_c^2\boldsymbol{z}^t\boldsymbol{z} - 2g_p\boldsymbol{x}^t\boldsymbol{y} - 2g_c\boldsymbol{x}^t\boldsymbol{z} + 2g_pg_c\boldsymbol{y}^t\boldsymbol{z} \quad ,$$

where $\boldsymbol{x}$ is the adaptive target, $\boldsymbol{y}$ is the filtered adaptive vector, and $\boldsymbol{z}$ is the filtered fixed vector; g_p is the pitch gain, g_c is the fixed gain.

The functionality is as follows:

1. Calculate the average energy minus the mean energy of the fixed-codebook contribution:

$$E = 10\log\left(\frac{1}{40}\sum_{n=0}^{39} c(n)^2\right)$$

Then calculate $Eb - E = 30(dB) - E$.

2. Calculate the predicted energy from prediction error of the previous subframes:

$$\tilde{E}^m = \sum_{i=1}^{4} b_i \hat{U}^{(m-i)}$$

$\hat{U}^m$ is calculated by the equation: $U^m = E - \tilde{E}^m$.

3. Calculate g_c by the equation: $g_c = 10^{(\tilde{E}^m + Eb - E)/20}$.

4. Calculate the following elements:

$G0 = \boldsymbol{y}^t\boldsymbol{y}$, $G1 = -2\boldsymbol{x}^t\boldsymbol{y}$, $G2 = \boldsymbol{z}^t\boldsymbol{z}$, $G3 = -2\boldsymbol{x}^t\boldsymbol{z}$, $G4 = 2\boldsymbol{y}^t\boldsymbol{z}$.

5. Calculate optimum gain G_p and G_c as:

$$G_p = -\frac{2\times G1\times G2 - G3\times G4}{4\times G0\times G2 - G4\times G4} \quad , \quad G_c = -\frac{2\times G0\times G3 - G1\times G4}{4\times G0\times G2 - G4\times G4}$$

6. Pre-select a domain in which G_p and G_c are most possibly located. The output is *index*0 and *index*1.

7. Select the gain vector in the codebook *GA* and *GB*, the range is limited by *index*0 and *index*1. The search criteria is minimizing the following expression:

$$E = G0 \times G_p^2 + G2 \times G_c^2 + G1 \times G_p + G3 \times G_c + G4 \times G_p \times G_c .$$

8. Update U^m.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcAdptTarget*, *pSrcFltAdptVector*, *pSrcFixedVector*, *pSrcFltFixedVector*, *pSrcDstEnergyErr*, *pDstQGain* or *pDstQGainIndex* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *tameProcess* is not 0 or 1.

AdaptiveCodebookContribution_G729

Updates target vector for codebook search by subtracting adaptive codebook contribution.

Syntax

```
IppStatus ippsAdaptiveCodebookContribution_G729_16s (Ipp16s gain, const Ipp16s*
    pFltAdptVector, const Ipp16s* pSrcAdptTarget, Ipp16s* pDstAdptTarget);
IppStatus ippsAdaptiveCodebookContribution_G729_32f (Ipp32f gain, const Ipp32f*
    pFltAdptVector, const Ipp32f* pSrcAdptTarget, Ipp32f* pDstAdptTarget);
```

Parameters

gain	Adaptive codebook gain g_p.
pFltAdptVector	Pointer to the input filtered adaptive codebook vector $y(n)$.
pSrcAdptTarget	Pointer to the input target vector $x(n)$.
pDstAdptTarget	Pointer to the output updated target vector $x'(n)$.

Description

The function `ippsAdaptiveCodebookContribution_G729` is declared in the `ippsc.h` file. This function subtracts the adaptive codebook contribution from the target signal and stores updated target vector as follows:

$x'(n) = x(n) - g_p \cdot y(n)$, $n = 0,\ldots, 39$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFltAdptVector* or *pSrcAdptTarget* or *pDstAdptTarget* pointer is NULL.

AdaptiveCodebookGain_G729

Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.

Syntax

```
IppStatus ippsAdaptiveCodebookGain_G729_16s(const Ipp16s* pSrcAdptTarget,
    const Ipp16s* pSrcImpulseResponse, const Ipp16s* pSrcAdptVector, Ipp16s*
    pDstFltAdptVector, Ipp16s* pResultAdptGain);
IppStatus ippsAdaptiveCodebookGain_G729A_16s(const Ipp16s* pSrcAdptTarget,
    const Ipp16s* pSrcLPC, const Ipp16s* pSrcAdptVector, Ipp16s*
    pDstFltAdptVector, Ipp16s* pResultAdptGain);
```

Parameters

pSrcAdptTarget	Pointer to the adaptive target signal vector [40].
pSrcImpulseResponse	Pointer to the impulse response of the perceptual weighting filter vector [40], in Q12.
pSrcAdptVector	Pointer to the adaptive-codebook vector [40], in Q12 (for ippsAdaptiveCodebookGain_G729_16s) or in Q10 (for ippsAdaptiveCodebookGain_G729A_16s).
pSrcLPC	Pointer to the LPC coefficients of the synthesis filter vector [11], in Q12.
pDstFltAdptVector	Pointer to the filtered adaptive-codebook vector [40].
pResultAdptGain	Pointer to the adaptive-codebook gain, in Q14.

Description

These functions are declared in the `ippsc.h` file. They compute the gain of the adaptive-codebook vector, and calculate the filtered adaptive-codebook vector, respectively. Both functions are applied in subframes.
The first function `ippsAdaptiveCodebookGain_G729` is designed for G.729/B codec and implements the following functionality:

1. First, convolve the adaptive-codebook vector, $v(n)$, with the impulse response, $h(n)$, to obtain the filtered adaptive-codebook vector, $y(n)$, using the formula:

$$y(n) = \sum_{i=0}^{n} v(i)h(n-i) \ , \ n = 0,1,\ldots,39 \ .$$

2. Next, calculate the adaptive-codebook gain g_p as given by:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)} \ ,$$

where $x(n)$ is the adaptive target signal.

3. Finally, bound the adaptive-codebook gain g_p to the range [0, 1.2].

The second function `ippsAdaptiveCodebookGain_G729A` is designed for G.729A codec and implements the following functionality:

1. First, pass the adaptive-codebook vector, $v(n)$, through the synthesis filter, $1/A_q(z/\gamma)$, to obtain the filtered adaptive-codebook vector, $y(n)$, using the formula :

$$y(n) = v(n) - \sum_{i=1}^{10} a_i y(n-i) \ , \ n = 0,1,\ldots,39 \ .$$

2. Next, calculate the adaptive-codebook gain g_p as:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)},$$

where $x(n)$ is the adaptive target signal.

3. Finally, bound the adaptive-codebook gain g_p to the range [0, 1.2].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcAdptTarget*, *pSrcImpulseResponse*, *pSrcAdptVector*, *pSrcLPC*, *pDstFltAdptVector*, or *pResultAdptGain* pointer is `NULL`.

Filter Functions

These functions are used to perform different types of filtering, including high pass filtering on pre-processing stage of encoding and post-processing stage of decoding, as well as the long-term and short-term postfilter in decoding.
The residual (FIR), simple synthesis (IIR), harmonic, and preemphasize filter functions may be combined to perform more complex synthesis filtering and may be used in different encode and decode stages.

ResidualFilter_G729

Implements an inverse LP filter and obtains the residual signal.

Syntax

```
IppStatus ippsResidualFilter_G729_16s(const Ipp16s*  pSrcSpch, const Ipp16s*
    pSrcLPC, Ipp16s*  pDstResidual);
```

```
IppStatus ippsResidualFilter_G729E_16s(const Ipp16s* pCoeffs, int order, const
    Ipp16s* pSrc, Ipp16s* pDst, int len)
```

Parameters

pSrcSpch	Pointer to the input speech signal. Elements *pSrcSpch*[0...39] are the present speech signals, whereas *pSrcSpch*[-10... -1] are the history to be used.
pSrcLPC	Pointer to the LP coefficients vector [11], in Q12.
pDstResidual	Pointer to the LP residual.
pCoeffs	Pointer to the vector [*order*+1] of filter coefficients.
order	The filter order.
pSrc	Pointer to the source vector [*len*]. Values *pSrc* [-*order*,..,-1] must also be supplied.
pDst	Pointer to the output filtered residual vector [len].
len	Length of the source and destination vectors.

Description

The functions `ippsResidualFilter_G729` and `ippsResidualFilter_G729E` are declared in the `ippsc.h` file.

`ippsResidualFilter_G729`. This function filters the input speech signal through an inverse LP filter and gets the residual signal as:

$$r(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i \times s(n-i) \ , \ n = 0,1,\ldots,39 \ .$$

`ippsResidualFilter_G729E`. This function implements the same operation as the previous function, but accepts the variable filter order.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcSpch*, *pSrcLPC*, *pDstResidual*, *pSrC*, *pCoeffs*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *order* is less than or equal to 0, or when *len* is less than or equal to 0 and `valLPCOrder` > *len* .

SynthesisFilter_G729

Reconstructs the speech signal from LP coefficients and residuals.

Syntax

```
IppStatus ippsSynthesisFilter_G729_16s(const Ipp16s*  pSrcResidual, const
    Ipp16s*  pSrcLPC, Ipp16s*  pSrcDstSpch);
IppStatus ippsSynthesisFilterZeroStateResponse_NR_16s(const Ipp16s* pSrcLPC,
    Ipp16s* pDstImp, int len, int scaleFactor);
IppStatus ippsSynthesisFilter_G729E_16s(const Ipp16s* pLPC, int order, const
    Ipp16s* pSrc, Ipp16s* pDst, int len, const Ipp16s* pMem);
IppStatus ippsSynthesisFilter_G729E_16s_I(const Ipp16s* pLPC, int order,
    Ipp16s* pSrcDst, int len, const Ipp16s* pMem);
IppStatus ippsSynthesisFilter_G729_32f(const Ipp32f* pLpc, int order, const
    Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f* pMem);
```

Parameters

pSrcResidual	Pointer to the LP residual signal vector [40].
pSrcLPC	Pointer to the LP coefficients vector [11], in Q12 for `16s` data.
pSrcDstSpch	Pointer to the synthesized and updated speech. Elements *pSrcDstSpch*[0...39] are the present synthesized speech, *pSrcDstSpch*[-10...-1] are the history to be used.
pLPC	Pointer to the LP coefficients vector [*order*+1].
order	The filter order (*pLPC*[*order*+1] vector must be supplied on input.
pSrc	Pointer to the source vector [*len*].
pDst	Pointer to the destination speech vector [*len*].
pSrcDst	Pointer to the source and destination vector [*len*].
pDstImp	Pointer to the destination zero state impulse response vectore [*len*]
len	Length of the source and destination vectors.
pMem	Pointer to the filter memory vector [*order*] supplied for filtering.
scaleFactor	Scale factor for the input LP coefficients and output impulse response, accordingly.

Description

These functions are declared in the `ippsc.h` file.

The function `ippsSynthesisFilter_G729_16s` uses the default LP filter of 10-th order to reconstruct speech signal from the residual signal as:

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) \quad , n = 0,1,\ldots,39 .$$

The function `ippsSynthesisFilterZeroStateResponse_NR_16s` performs the same operation as `ippsSynthesisFilter_NR_16s` for the same parameters, but uses no memory and processes the impulse input vector *pSrc* which is set as

$pSrc[0] = 2^{scaleFactor}$,
$pSrc[i] = 0$, $i = 1,\ldots len - 1$.

The functions `ippsSynthesisFilter_G729E_16s`, `ippsSynthesisFilter_G729E_16s_I`, and `ippsSynthesisFilter_G729_32f` perform synthesis filtering of the order given by the parameter *order* and operate like `ippsSynthesisFilter_G729_16s` function for the default order 10.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the pointers is `NULL`.
`ippStsOverflow`	Indicates a warning that an overflow has occurred.
`ippStsSizeErr`	Indicates an error when *order* is less than or equal to 0.

LongTermPostFilter_G729

Restores the long-term information from the old speech signal.

Syntax

```
IppStatus ippsLongTermPostFilter_G729_16s (Ipp16s gammaFactor, int valDelay,
    const Ipp16s* pSrcDstResidual, Ipp16s* pDstFltResidual,
    Ipp16s* pResultVoice);
```

```
IppStatus ippsLongTermPostFilter_G729A_16s(Ipp16s valDelay, const Ipp16s*
    pSrcSpch, const Ipp16s*  pSrcLPC, Ipp16s*  pSrcDstResidual, Ipp16s*
    pDstFltResidual);
IppStatus ippsLongTermPostFilter_G729B_16s(Ipp16s valDelay, const Ipp16s*
    pSrcSpch, const Ipp16s*  pSrcLPC, Ipp16s*  pSrcDstResidual, Ipp16s*
    pDstFltResidual, Ipp16s*  pResultVoice, Ipp16s frameType);
```

Parameters

gammaFactor	Post filter coefficient y_p in Q15.
valDelay	Pitch delay.
pSrcSpch	Pointer to the reconstructed speech $\hat{s}(n)$, in Q15. Elements *pSrcSpch*[0...39] are the present speech signals, elements *pSrcSpch*[-10...-1] are the history that will be used.
pSrcLPC	Pointer to the LP coefficients $\hat{a}_i$ vector [11], in Q12.
pSrcDstResidual	Pointer to the residual after long term prediction vector [40], elements [-152...-1] are the long term prediction history that will be used.
pDstFltResidual	Pointer to the output filtered residual vector [40].
pResultVoice	Pointer to the voice information.
frameType	The type of the frame: 0 - stands for untransmitted frame, 1 - stands for normal speech frame, 2 - stands for SID frame.

Description

These functions are declared in the `ippsc.h` file. These FIR filters are used to restore the long-term relationship from old speech at the length of pitch. They are applied in subframes.

The first function **ippsLongTermPostFilter_G729** has the following functionality:

1. First, calculate autocorrelation $R(k)$ of $\hat{r}(n)$ and find the best integer $T0$, in the range of [int($T1$)-1, int($T1$)+1], which maximizes $R(k)$ using the following formula:

$$R(k) = \sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}(n-k) \tag{9-6}$$

Here int($T1$) is the integer part of the pitch delay $T1$ in the first subframe.

2. Second, seek for the best fractional delay *T* with resolution 1/8 around *T*0. This is done by finding the delay with the highest pseudo-normalized correlation $R'(k)$:

$$R'(k) = \frac{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}_k(n)}{\sqrt{\sum_{n=0}^{39} \hat{r}_k(n) \cdot \hat{r}_k(n)}} ,$$

where $\hat{r}_k(n)$ is the residual at delay *k*.

3. Once the optimal delay *T* is found, determine whether the long-term filter should be disabled or not, using the following condition:

$$\text{if } \frac{R'(T)^2}{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}(n)} < 0.5 \text{ , let } g_1 = 0 ; \tag{9-7}$$

$$\text{otherwise, let } g_1 = \frac{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}_k(n)}{\sum_{n=0}^{39} \hat{r}_k(n) \cdot \hat{r}_k(n)} \text{ , and bound it to [0, 1.0]} \tag{9-8}$$

4. Finally, use the following FIR filter:

$$y(n) = \frac{1}{1+\gamma_p \cdot g_1}(\hat{r}(n) + \gamma_p \cdot g_1 \cdot \hat{r}(n-T)) , \tag{9-9}$$

where $\hat{r}(n)$ is the residual signal and the factor γ_p controls the amount of long-term post-filtering.

The second function **ippsLongTermPostFilter_G729A** is designed for G.729A codec, and performs the following operations:

1. First, get the residual $\hat{r}(n)$ from synthesized speech $\hat{s}(n)$ using the formula:

$$\hat{r}(n) = \hat{s}(n) + \sum_{i=1}^{10} \gamma_n^i \cdot \hat{a}_i \cdot \hat{s}(n-i) , \tag{9-5}$$

where $\hat{s}(n)$ is the synthesized speech, $\hat{a}_i$ are the quantized LP coefficients and

$\gamma_n = 0.55$.

2. Second, calculate autocorrelation $R(k)$ of $\hat{r}(n)$ and find the best integer *T* in the range [int(*T*1)-3, int(*T*1)+3], which maximizes $R(k)$ using the formula (9-6).

Here int(*T*1) is the integer part of the pitch delay *T*1 in the first subframe.

3. Once the optimal delay *T* is found, determine whether the long-term filter should be disabled or not, using the relations (9-7) and (9-8).

6. Finally, use the FIR filter (9-9).

The third function **ippsLongTermPostFilter_G729B** is actually the combination of other functions as demonstrated by the code below:

```
{
 const short g_pst[11] = {32767,18022,9912,5451,2998,1649,907,499,274,151,83};
    /* yn = 0.55 powered by i=0,1,…,10 in Q15 */
 short coeffs[LP_ORDER+1];          /* temporary nominator coefficients  */
 short vc;

 ippsMul_NR_16s_Sfs(g_pst, pSrcLpc, coeffs, LP_11, 15);
 ippsResidualFilter_G729_16s(pSrcSpch, coeffs, pSrcDstResidual+154);
 if (1 == frameType){
    ippsLongTermPostFilter_G729_16s (16384,valDelay, pSrcDstResidual + 154,
                pDstFltResidual, &vc); /* Harmonic filtering : gp=0.5 in Q15*/
    *pResultVoice = (vc != 0);
 }else{
    ippsCopy_16s(pSrcDstResidual + 154,pDstFltResidual,40);
    *pResultVoice = 0;
 }
}
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcSpch*, *pSrcLPC*, *pSrcDstResidual*, *pDstFltResidual* or *pResultVoice* pointer is NULL.

ippStsRangeErr	Indicates an error when *valDelay* is not in the range [18, 145] for ippsLongTermPostFilter_G729A function, or *valDelay* is not in the range [0, 143] for ippsLongTermPostFilter_G729B function, or *frameType* is not in the range [0, 2].

ShortTermPostFilter_G729

Restores speech signal from the residuals.

Syntax

```
IppStatus ippsShortTermPostFilter_G729_16s(const Ipp16s* pSrcLPC, const
    Ipp16s* pSrcFltResidual, Ipp16s* pSrcDstSpch, Ipp16s*
    pDstImpulseResponse);
IppStatus ippsShortTermPostFilter_G729A_16s(const Ipp16s* pSrcLPC, const
    Ipp16s* pSrcFltResidual, Ipp16s* pSrcDstSpch);
```

Parameters

pSrcLPC	Pointer to the quantized LP coefficients vector [11], in Q12.
pSrcFltResidual	Pointer to the residual signal $x(n)$ vector [40], in Q15.
pSrcDstSpch	Pointer to the short-term filtered speech $y(n)$, in Q15. Elements *pSrcDstSpch*[0...39]are the present short-term filtered speech signals, elements *pSrcDstSpch*[-10...-1] are the history that will be used.
pDstImpulseResponse	Pointer to the generated impulse response $h_f(n)$ vector [20], in Q12.

Description

These functions are declared in the ippsc.h file. These two IIR filters restore the speech from the residual. They both are applied in subframes.

The first function ippsShortTermPostFilter_G729 is designed for G.729/B codec, and has the following functionality:

1. First, compute the impulse response $h_f(n)$ of the inverse filter. $\hat{A}(z)$.

2. Next, calculate the gain term g_f using the formula:

$$g_f = \sum_{0}^{10} |h_f(n)|$$

3. Finally, filter the residual through the inverse filter $\hat{A}(z)$, as follows:

$$y(n) = \frac{1}{g_f} x(n) - \sum_{i=1}^{10} \hat{a}_i y(n-i) \ , \ n = 0,1,\ldots,39 \ ,$$

where $x(n)$ is the residual and $\hat{a}_i$ are the weighted quantized LP coefficients.

The second function `ippsShortTermPostFilter_G729A` is designed for G.729A codec. This function filters the residual through the inverse filter $\hat{A}(z)$, using the formula:

$$y(n) = x(n) - \sum_{i=1}^{10} \hat{a}_i y(n-i) \ , \ n = 0,1,\ldots,39 \ ,$$

where $x(n)$ is the residual and $\hat{a}_i$ are the weighted quantized LP coefficients.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLPC*, *pSrcFltResidual*, *pSrcDstSpch*, or *pDstImpulseResponse* pointer is `NULL`.

TiltCompensation_G729

Compensates for the tilt in the short-term filter.

Syntax

```
IppStatus ippsTiltCompensation_G729_16s(const Ipp16s*  pSrcImpulseResponse,
    Ipp16s*  pSrcDstSpch);
IppStatus ippsTiltCompensation_G729A_16s(const Ipp16s*  pSrcLPC, Ipp16s*
    pSrcDstFltResidual);
IppStatus ippsTiltCompensation_G729E_16s(Ipp16s val, const Ipp16s* pSrc,
    Ipp16s* pDst);
```

Parameters

pSrcImpulseResponse	Pointer to the impulse response $h_f(n)$ vector [20], in Q12.
pSrcLPC	Pointer to the gamma weighted LP coefficients vector [22], in Q12. The first 11 elements refer to γ_n and the next 11 elements refer to γ_d.
pSrcDstSpch	Pointer to the present and tilt-compensated speech *x*(*n*), in Q15. Elements *pSrcDstSpch*[0...39] are the present speech signals, element *pSrcDstSpch*[-1] is the history that will be used.
pSrcDstFltResidual	Pointer to the long-term filtered LP residual, in Q15. Elements *pSrcDstFltResidual*[0...39] are the present long-term filtered LP residual signals, element *pSrcDstFltResidual*[-1] is the history that will be used.
val	The input tilt factor.
pSrc	Pointer to the source vector [41].
pDst	Pointer to the output filtered residual vector [40]

Description

These functions are declared in the `ippsc.h` file. These FIR filters compensate for the tilt in the short-term filter. They are applied in subframes.

The function `ippsTiltCompensation_G729` is designed for G.729/B codec, and the functionality is as follows.

1. First, calculate correlation $r_h(i)$ of the impulse response $h_f(i)$ as:

$$r_h(i) = \sum_{j=0}^{19-i} h_f(j)h_f(j+i)$$

Then obtain the tilt factor k_1' using the relation:

$$k_1' = -\frac{r_h(1)}{r_h(0)}$$

2. Next, obtain the gain term g_t using the formula:

$$g_t = 1 - |\gamma_t \cdot k_1'| ,$$

where $\gamma_t = 0.9$ if k_1' is negative, and $\gamma_t = 0.2$ if k_1' is positive.

3. Finally, filter the speech using the formula:

$$y(n) = \frac{1}{g_t}(x(n) + \gamma_t \cdot k_1' \cdot x(n-1)) ,$$

where $x(n)$ is the input speech signal.

The function `ippsTiltCompensation_G729E` performs only stages 2 and 3 for a given input tilt factor *val* and filter memory specified by *pSrc*[0].

The function `ippsTiltCompensation_G729A` is designed for G.729A codec, and the functionality is as follows.

1. First, compute the impulse response $h_f(n)$ of the inverse filter. $\hat{A}(z)$.

2. Next, calculate correlation $r_h(i)$ of the impulse response $h_f(i)$ as:

$$r_h(i) = \sum_{j=0}^{21-i} h_f(j)h_f(j+i)$$

Then obtain the tilt factor k_1' , using the relation:

$$k_1' = -\frac{r_h(1)}{r_h(0)}$$

3. Finally, filter the residual using the formula:

$$y(n) = x(n) + \gamma_t \cdot k_1' \cdot x(n-1) ,$$

where $x(n)$ is the residual, and $\gamma_t = 0.8$ if k_1' is negative; or $\gamma_t = 0$ if k_1' is positive.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDstSpch*, *pSrcImpulseResponse*, *pSrcLPC*, *pSrc*, *pDst*, or *pSrcDstFltResidual* pointer is `NULL`.

HarmonicFilter

Calculates the harmonic filter.

Syntax

```
IppStatus ippsHarmonicFilter_16s_I (Ipp16s beta, int T, Ipp16s* pSrcDst,
    int len);
IppStatus ippsHarmonicFilter_NR_16s (Ipp16s beta, int T, const Ipp16s*  pSrc,
    Ipp16s*  pDst, int len);
```

```
IppStatus ippsHarmonicFilter_32f_I(Ipp32f beta, int T, Ipp32f* pSrcDst, int
    len);
```

Parameters

beta	The beta factor, in Q15 for `16s` data.
T	The integer delay.
pSrc	Pointer to the source vector. Elements *pSrc*[-*T*,...,0,...*len*-1] will be used as input.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector. Elements *pSrcDst*[-*T*,...,0,...,*len*-1] will be used as input, elements *pSrcDst*[0,...,*len*-1] will be computed.
len	Number of elements in the source and destination vectors.

Description

These functions are declared in the `ippsc.h` file.

The functions `ippsHarmonicFilter_16s_I` and `ippsHarmonicFilter_32f_I` implement the adaptive pre-filter:

$$H(z) = \frac{1}{1 - \beta \cdot z^{-T}},$$

which can be used to enhance the harmonic component of speech.

The calculation is as follows:

$$pSrcDst[n] = pSrcDst[n] + beta \cdot pSrcDst[n - T], \quad 0 \le n < len .$$

The function `ippsHarmonicFilter_NR_16s` implements the harmonic noise shaping filter:

$$H(z) = 1 + \beta \cdot z^{-T}$$

The calculation is as follows:

$$pDst[n] = pSrc[n] + beta \cdot pSrc[n - T], \quad 0 \le n < len .$$

Both filters can be used to enhance the harmonic component of the input signal.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

HighPassFilterSize_G729

Returns the G729 high-pass filter size.

Syntax

```
IppStatus ippsHighPassFilterSize_G729 (int* pSize);
```

Parameters

pSize	Pointer to the output value of the memory size.

Discussion

The function `ippsHighPassFilterSize_G729` is declared in the `ippsc.h` file. This function returns the size of memory required for the `ippsHighPassFilterInit_G729` function to operate.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSize* pointer is `NULL`.

HighPassFilterInit_G729

Initializes the high-pass filter.

Syntax

```
IppStatus ippsHighPassFilterInit_G729 (Ipp16s* pCoeff, char* pMemUpdated);
```

Parameters

pCoeff	Pointer to the array of length 6 containing values a_0, a_1, a_2, b_0, b_1, b_2, in Q12 or Q13.
pMemUpdated	Pointer to the memory allocated for the filter.

Description

The function `ippsHighPassFilterInit_G729` is declared in the `ippsc.h` file. This function initializes the internal data of the high-pass filter: a_0 should be equal to scaled 1 (in Q12 or Q13). The filter history data x_{-2}, x_{-1}, y_{-2}, y_{-1} are set to zero. The `ippsHighPassFilter_G729_16s_ISfs` function uses this memory for filtering.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pCoeff* or *pMemUpdated* pointer is NULL.

HighPassFilter_G729

Performs G729 high-pass filtering.

Syntax

```
IppStatus ippsHighPassFilter_G729_16s_ISfs (Ipp16s* pSrcDst, int len, int
    scaleFactor, char* pMemUpdated);
```

Parameters

pSrcDst	Pointer to the source and destination vector [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Scale factor for output data scaling.
pMemUpdated	Pointer to the memory allocated for the filter.

Description

The function `ippsHighPassFilter_G729` is declared in the `ippsc.h` file. This function performs the input signal pre-processing or the output signal post-processing using the high-pass filter:

$$H_h(z) = \frac{b_0 + b_1 \cdot z^{-1} + b_2 \cdot z^{-2}}{a_0 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2}}$$

Currently, only $a_0 = 1$ (in Q12 or Q13) is supported. The function uses the scale factor value equal to 12 for input data pre-filtering, and equal to 13 for output data post-filtering.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* or *pMemUpdated* pointer is NULL.
ippStsScaleRangeErr	Indicates an error when *scaleFactor* is not equal to 12 or 13.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

IIR16s_G729

Performs IIR filtering.

Syntax

```
IppStatus ippsIIR16sLow_G729_16s(const Ipp16s* pCoeffs, const Ipp16s* pSrc,
    Ipp16s* pDst, Ipp16s* pMem);
IppStatus ippsIIR16s_G729_16s(const Ipp16s* pCoeffs, const Ipp16s* pSrc,
    Ipp16s* pDst, Ipp16s* pMem);
```

Parameters

pCoeffs	Pointer to the input vector of filter coefficients vector [20]: $b_1,\ldots, b_{10}, a_1, \ldots,a_{10}$.
pSrc	Pointer to the input signal vector [40].
pDst	Pointer to the output filtered vector [40].
pMem	Pointer to the filter memory vector [20]. This vector must be initially filled with zeroes.

Description

The functions ippsIIR16sLow_G729s_16s and ippsIIR16s_G729_16s are declared in the ippsc.h file.

These functions perform infinite impulse response (IIR) filtering using the following transfer function:

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The function `ippsIIR16sLow_G729_16s` does not check for overflow and result saturation, whereas the function `ippsIIR16s_G729_16s` makes these checks.

The filter memory is updated.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pCoeffs*, *pSrc*, *pDst*, or *pMem* pointer is `NULL`.

PhaseDispersionGetStateSize_G729D

Queries the memory length of the phase dispersion filter.

Syntax

```
IppStatus ippsPhaseDispersionGetStateSize_G729D_16s (int* pSize);
```

Parameters

pSize	Pointer to the output phase dispersion filter memory length in bytes.

Description

The function `ippsPhaseDispersionGetStateSize_G729D_16s` reports the size of memory that the phase dispersion filter needs for proper operation.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSize* pointer is `NULL`.

PhaseDispersionInit_G729D

Initializes the phase dispersion filter memory.

Syntax

```
IppStatus ippsPhaseDispersionInit_G729D_16s
    (IppsPhaseDispersion16s_State_G729D* pPhDMem);
```

Parameters

pPhDMem	Pointer to the memory of the phase dispersion filter.

Description

The function `ippsPhaseDispersionInit_G729D_16s` sets the contents of the given buffer as memory of the phase dispersion filter in initial state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pPhDMem* pointer is `NULL`.

PhaseDispersionUpdate_G729D

Updates the phase dispersion filter state.

Syntax

```
IppStatus ippsPhaseDispersionUpdate_G729D_16s(Ipp16s valPitchGain,
    Ipp16s valCodebookGain, IppsPhaseDispersion16s_State_G729D* pPhDMem);
```

Parameters

valPitchGain	Long term pitch gain.
valCodebookGain	Codebook gain
pPhDMem	Pointer to the memory of phase dispersion filter.

Description

The function `ippsPhaseDispersionUpdate_G729D_16s` updates the state of phase dispersion filter memory with given pitch and codebook gains.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pPhDMem* pointer is `NULL`.

PhaseDispersion_G729D

Performs the phase dispersion filtering.

Syntax

```
IppStatus ippsPhaseDispersion_G729D_16s(Ipp16s valCodebookGain, Ipp16s
    valPitchGain,  const Ipp16s* pSrcExcSignal, Ipp16s* pDstFltExcSignal,
    Ipp16s* pSrcDstInnovation, IppsPhaseDispersion16s_State_G729D* pPhDMem)
```

Parameters

valPitchGain	Input long term pitch gain.
valCodebookGain	Input codebook gain
pSrcExcSignal	Pointer to the input excitation vector [40].
pDstFltExcSignal	Pointer to the output filtered excitation vector [40].
pSrcDstInnovation	Pointer to the input/output innovative codebook vector [40].
pPhDMem	Pointer to the memory of phase dispersion filter.

Description

The function `ippsPhaseDispersion_G729D_16s` performs the phase dispersion filtering for the given pitch and codebook gains. The filter alters the innovation signal (mainly its phase), such that a new innovation signal is created with the energy being more spread over the subframe. The filtering is performed by a circular convolution using one of the three "semi-random" impulse responses according to the different amounts of spreading.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrcExcSignal*, *pDstFltExcSignal*, *pSrcDstInnovation* or *pPhDMem* pointer is NULL.

Preemphasize_G729A

Computes pre-emphasis of a post filter.

Syntax

```
IppStatus ippsPreemphasize_G729A_16s (Ipp16s gamma, const Ipp16s* pSrc, Ipp16s*
    pDst, int len, Ipp16s* pMem);
IppStatus ippsPreemphasize_G729A_16s_I (Ipp16s gamma, Ipp16s* pSrcDst, int len,
    Ipp16s* pMem);
IppStatus ippsPreemphasize_32f_I(Ipp32f gamma,  Ipp32f* pSrcDst,  int len,
    Ipp32f* pMem);
```

Parameters

gamma	The filter coefficient, in Q15 for 16s data.
pSrc	Pointer to the source vector, in Q0.
pDst	Pointer to the destination vector, in Q0.
pSrcDst	Pointer to the source and destination vector, in Q0.
len	Number of elements in the source and destination vectors.
pMem	Pointer to the filter memory [1].

Description

These functions are declared in the ippsc.h file and compute pre-emphasis of a post filter.

The computation is performed according to the difference signal pre-emphasis equation:

$$H(z) = 1 - gamma \cdot z^{-1}$$

The filter memory *pMem* should be initialized to zero and then be constantly updated while filtering.

The functions ippsPreemphasize_G729A perform no rounding (clipping).

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *pSrcDst*, or *pMem* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

WinHybridGetStateSize_G729E

Queries the length of the hybrid windowing module memory.

Syntax

```
IppStatus ippsWinHybridGetStateSize_G729E_16s (int* pSize);
IppStatus ippsWinHybridGetStateSize_G729E_32f (int* pSize);
```

Parameters

pSize	Pointer to the output length of hybrid windowing module memory in bytes.

Description

The function ippsWinHybridGetStateSize_G729E reports the size of memory that the hybrid windowing module needs for proper operation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSize* pointer is NULL.

WinHybridInit_G729E

Initializes the hybrid windowing module memory.

Syntax

```
IppStatus ippsWinHybridInit_G729E_16s (IppsWinHybridState_G729E_16s* pMem);
IppStatus ippsWinHybridInit_G729E_32f(IppsWinHybridState_G729E_32f* pMem);
```

Parameters

pMem	Pointer to the memory of hybrid windowing module.

Description

The function `ippsWinHybridInit_G729E` sets the contents of the given buffer as the memory of the hybrid windowing module in initial state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMem* pointer is `NULL`.

WinHybrid_G729E

Applies the hybrid window and computes autocorrelation coefficients.

Syntax

```
IppStatus ippsWinHybrid_G729E_16s32s (const Ipp16s* pSrcSynthSpeech, Ipp32s*
    pDstInvAutoCorr, IppsWinHybridState_G729E_16s* pMem);
IppStatus ippsWinHybrid_G729E_32f(const Ipp32f* pSrcSynthSpeech, Ipp32f*
    pDstInvAutoCorr, IppsWinHybridState_G729E_32f* pMem);
```

Parameters

pSrcSynthSpeech	Pointer to the input synthesized speech vector [144].
pDstInvAutoCorr	Pointer to the output autocorrelation vector [31].
pMem	Pointer to the memory of hybrid windowing module.

Description

The function `ippsWinHybrid_G729E` applies the hybrid window to input synthesized speech and computes autocorrelation coefficients as follows.

1. First, the input speech vector is multiplied by the hybrid window:

$$s_{w+I}(k) = s_w(k) \cdot w(144 - k), k = 0,\ldots,144$$

where

$$w(k) = \begin{cases} \sin((k+1)*0.047783), k = 0,\ldots,34 \\ \sin(36*0.047783)*a^{(35-k)}, k = 35,\ldots,144 \end{cases}$$

Here a = 0.9928337491 so that a^{40}=0.75 and a^{160} = 0.75^4 = 0.31640625 .

2. Second, the autocorrelation of the windowed speech is calculated by the formulae:

$$R_{rec}(n) = \sum_{k=0}^{k=79} s(k+30) \cdot s(k+30-n), n = 0,\ldots,30$$

$$R(n) = \sum_{k=0}^{k=34} s(k+110) \cdot s(k+110-n), n = 0,\ldots,30$$

3. Third, the output autocorrelation is computed and the memory is updated as follows:

$$\begin{cases} mem(k) = 0.31640625*mem(k) + R_{rec}(k) \\ pDstAutoCorr(k) = mem(k) + R(k) \end{cases}, k = 0,\ldots,30$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcSynthSpeech*, *pDstInvAutoCorr* or *pMem* pointer is `NULL`.

RandomNoiseExcitation_G729B

Initializes a random vector with a Gaussian distribution.

Syntax

```
IppStatus ippsRandomNoiseExcitation_G729B_16s (Ipp16s* pSeed, Ipp16s* pExc, int
    len);
IppStatus ippsRandomNoiseExcitation_G729B_16s32f(Ipp16s* pSeed, Ipp32f* pExc,
    int len);
```

Parameters

pSeed	Pointer to the input/output seed of random generator.
pExc	Pointer to the destination vector.
len	Length of the destination vector.

Description

The function `ippsRandomNoiseExcitation_G729B_16s` is declared in the `ippsc.h` file. This function generates random excitations as follows:

$$pEcx[n] = \frac{\sum_{i=1}^{12} \xi_{(12n+i)}}{128}, \quad n = 0,\ldots, len-1$$

where $\xi_{k+1} = 31821 \cdot \xi_k + 13849$, $\xi_0 = *pSeed$

The seed of the random generator ξ_k is updated.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pExc* or *pSeed* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

FilteredExcitation_G729

Computes filtered exitation.

Syntax

```
IppStatus ippsFilteredExcitation_G729_32f(const Ipp32f* pSrcImpulseResponse,
    Ipp32f* pSrcDstPastExc, Ipp32f valExc, Ipp32s len);
```

Parameters

pSrcImpulseResponse	Impulse response [*len*].
pSrcDstPastExc	Pointer to the previous and updated excitation vector [*len*].
valExc	Excitation.
len	Length

Description

The function computes filtered excitation using following formula:

pSrcDstPastExc[i] = *SrcDstPastExc*[i-1] + *valExc** *pSrcImpulseResponse*[i],
pSrcDstPastExc[0] = *valExc* ,

where i=*len* - 1, ..., 1.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcImpulseResponse* or *pSrcDstPastExc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.

G.723.1 Related Functions

Intel IPP functions described in this section implement building blocks that can be used to create speech codecs compliant with the ITU-T Recommendation G.723.1 (see [ITU723], [ITU723A]).
The list of these functions is given in the following table:

Table 9-3 Intel IPP G.723.1 Related Functions

Function Base Name	Operation
Linear Prediction Analysis Functions	
AutoCorr_G723	Estimates the auto-correlation of a vector.
AutoCorr_NormE_G723	Estimates normal auto-correlation of a vector.
LevinsonDurbin_G723	Calculates the LP coefficients from the autocorrelation coefficients.
LPCToLSF_G723	Converts LP coefficients to LSF coefficients.
LSFToLPC_G723	Converts LSF coefficients to the LP coefficients.
LSFDecode_G723	Performs inverse quantization of LSFs.
LSFQuant_G723	Quantizes LSF coefficients.
Codebook Search Functions	
OpenLoopPitchSearch_G723	Searches for an optimal pitch value.
ACELPFixedCodebookSearch_G723	Searches the ACELP fixed codebook for the excitation.

Table 9-3 Intel IPP G.723.1 Related Functions (continued)

Function Base Name	Operation
AdaptiveCodebookSearch_G723	Searches for the close loop pitch and the adaptive gain index.
MPMLQFixedCodebookSearch_G723	Searches the MP-MLQ fixed codebook for the excitation.
ToeplizMatrix_G723	Calculates 416 elements of the Toepliz matrix for fixed codebook search.
Gain Quantization Functions	
GainQuant_G723	Implements MP-MLQ gain estimation and quantization.
GainControl_G723	Extracts delayed pitch contribution.
Filter Functions	
HighPassFilter_G723	Performs high-pass filtering of the input signal.
IIR16s_G723	Performs IIR filtering.
SynthesisFilter_G723	Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).
TiltCompensation_G723	Computes tilt compensation filter.
HarmonicSearch_G723	Searches for the harmonic delay and gain for the harmonic noise shaping filter.
HarmonicNoiseSubtract_G723	Performs harmonic noise shaping.
DecodeAdaptiveVector_G723	Restores the adaptive codebook vector from excitation, pitch, and adaptive gain.
PitchPostFilter_G723	Calculates coefficients of the pitch post filter.

These functions are used in the Intel IPP *G.723.1 Speech Encoder-Decoder* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

Linear Prediction Analysis Functions

These functions perform LPC analysis, LSP coding (quantization) and decoding, as well as transformation between LPC and LSF coefficients.

AutoCorr_G723

Estimates the auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorr_G723_16s(const Ipp16s* pSrcSpch,  Ipp16s*
    pResultAutoCorrExp, Ipp16s* pDstAutoCorr);
```

Parameters

pSrcSpch	Pointer to the input speech signal vector [180].
pResultAutoCorrExp	Pointer to the exponent for autocorrelation coefficients.
pDstAutoCorr	Pointer to the autocorrelation coefficients vector [11].

Description

The function `ippsAutoCorr_G723` is declared in the `ippsc.h` file. This function calculates the first 11 autocorrelation coefficients of the input speech signal. This function is applied to the 180 speech samples centered on the current subframe. The functionality is as follows.

1. First, apply to speech samples the Hamming windows, given by the following formula.

$$w(i) = 0.54 - 0.46\cos\left[\frac{(2i-1)}{399}\pi\right] \quad , i = 0,1,\ldots,179$$

2. Next, calculate the autocorrelations as follows:

$$r(k) = \sum_{i=k}^{179} s(i) \times s(i-k) \quad , k = 0,1,\ldots,10$$

3. Finally, add the autocorrelation with binomial window, given by:

$$b(0) = 1025/\ 1024$$

$$b(i) = \exp\left[-0.5\left(\frac{2\pi f_0 i}{f_s}\right)^2\right] \quad , i = 1,\ldots,10 \ .$$

NOTE. The function `ippsAutoCorr_G723` is actually a combination of `ippsAutoScale`, `ippsMul_NR` and `ippsAutoCorr_NormE_G723` functions. The following code details the correspondence.

```
{
        int autoScale=3, corrScale;
        short vect[180];
        ippsAutoScale_16s(pSrcSpch, Vect,180, &autoScale );
        /* Apply the Hamming window */
        ippsMul_NR_16s_ISfs(HammWindow,Vect,180,15);
        /* Compute the autocorrelation coefficients */
        ippsAutoCorr_NormE_G723_16s(Vect,pDstAutoCorr,&corrScale);
        *pResultAutoCorrExp = corrScale+(autoScale<<1);
}
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrcSpch* or *pDstAutoCorr* or *pResultAutoCorrExp* pointer is `NULL`.

AutoCorr_NormE_G723

Estimates normal auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorr_NormE_G723_16s(const Ipp16s* pSrc, Ipp16s* pDst, int*
    pNorm);
```

Parameters

pSrc	Pointer to the source vector [180].
pDst	Pointer to the destination vector, which stores the estimated auto-correlation results of the source vector.

pNorm	Pointer to the output normalization scale factor.

Description

The function `ippsAutoCorr_NormE_G723_16s` is declared in the `ippsc.h` file. This function computes 11 auto-correlation coefficients from the input signal. The first correlation coefficient (energy) is multiplied by a white noise correction factor $b(0) = 1025/1024$, and the remaining 10 correlation coefficients are multiplied by the binomial window coefficients
b_n, $n = 1,\ldots,10$ defined in the reference C-code (see [ITU723]).

The auto-correlation coefficients are additionally multiplied by the factor 2^{norm0}, where $norm0 \geq 0$ is calculated so as to make the first coefficient (energy) normalized. Thus, the resulting vector *pDst* is computed as follows:

$$pDst[n] = b_n \cdot 2^{norm0} \cdot \sum_{i=0}^{179-n} pSrc[i] \cdot pSrc[i+n] \quad , 0 \leq n \leq 10 .$$

The function `ippsAutoCorr_NormE` stores the normalization scale factor *norm*0 in *pNorm*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst* or *pNorm* pointer is `NULL`.

LevinsonDurbin_G723

Calculates the LP coefficients from the autocorrelation coefficients.

Syntax

```
IppStatus ippsLevinsonDurbin_G723_16s(const Ipp16s*  pSrcAutoCorr, Ipp16s*
    pValResultSineDtct, Ipp16s* pResultResidualEnergy, Ipp16s* pDstLPC);
```

Parameters

pSrcAutoCorr	Pointer to the autocorrelation coefficients vector [11].
pValResultSineDtct	Pointer to the sine detector input/output parameter.
pResultResidualEnergy	Pointer to the output residual energy, in Q15.

pDstLPC	Pointer to the output LP coefficients vector [10], in Q13.

Description

The function `ippsLevinsonDurbin_G723` is declared in the `ippsc.h` file. This function calculates the 10th order LP coefficients from the autocorrelation coefficients using Levinson-Durbin algorithm. This function also performs sine detection.

To obtain LP coefficients a_i, i = 1, 2, ..., 10, the following set of equations is to be solved:

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k) \ , \ k = 1,2,\ldots,10 \ .$$

The function `ippsLevinsonDurbin_G723` performs the following steps:

1. Levinson-Durbin algorithm is applied to solve the above set of equations. This algorithm uses the recursion detailed in `ippsLevinsonDurbin_G729` function.

2. If the LPC filter used in this algorithm is unstable, that is $|k_i|$ is very close to 1.0 during recursion, just set the remaining LP coefficients to zero.

3. The sine detector parameter is updated when i = 1 in the recursion, as follows:

$SineDtct = SineDtct << 1$

If k > 0.95, then $SineDtct = SineDtct + 1$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcAutoCorr*, *pDstLPC*, *pValResultSineDtct*, or *pResultResidualEnergy* pointer is NULL.

LPCToLSF_G723

Converts LP coefficients to LSF coefficients.

Syntax

```
IppStatus ippsLPCToLSF_G723_16s (const Ipp16s* pSrcLPC, const Ipp16s*
    pSrcPrevLSF, Ipp16s* pDstLSF);
```

Parameters

pSrcLPC	Pointer to the LPC input vector [10]: $a_1,\ldots,a_{10}$, in Q13.
pSrcPrevLSF	Pointer to the previous normalized LSF coefficients vector [10], in Q15.
pDstLSF	Pointer to the normalized LSF coefficients vector [10], in Q15.

Description

The function `ippsLPCToLSF_G723` is declared in the `ippsc.h` file. This function converts a set of 10th order LP coefficients to LSF coefficients by implementing the following operations:

1. Apply the 7.5Hz bandwidth expansion.

2. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursion:

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i)$$

$i = 0\ldots4$,

where $f_1(0) = f_2(0) = 1.0$

3. Use Chebyshev polynomials to find the roots of $F_1(z)$ and $F_2(z)$, and obtain the LSF coefficients:

$$c_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2$$

$$c_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2$$

3. If all 10 roots needed to determine LSF coefficients are not found, use the set of previous LSF coefficients.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLPC*, *pSrcPrevLSF or pDstLSF* pointer is `NULL`.

LSFToLPC_G723

Converts LSF coefficients to the LP coefficients.

Syntax

```
IppStatus ippsLSFToLPC_G723_16s(const Ipp16s* pSrcLSF, Ipp16s* pDstLPC);
IppStatus ippsLSFToLPC_G723_16s_I (Ipp16s* pSrcLSFDstLPC);
```

Parameters

pSrcLSF	Pointer to the input LSF coefficients vector [10], in Q15.
pDstLPC	Pointer to the output LP coefficients vector [10], in Q13.
pSrcLSFDstLPC	Pointer to the input LSF and output LP coefficients vector [10], in Q15 and Q13, respectively.

Description

The function `ippsLSFToLPC_G723` is declared in the `ippsc.h` file. This function converts a set of 10th order LSF coefficients to the LP coefficients as follows:

1. Convert LSF coefficients to the LSP coefficients as:
$q_i = \cos(\omega_i)$, $i = 0...9$

2. Calculate the LP coefficients using the same algorithm as in the function ippsLSFToLPC_G729 with the only difference in rounding mode required for G.723.1 bit-exact compliance.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLSF*, *pDstLPC*, *pSrcLSFDstLPC* pointer is `NULL`.

LSFDecode_G723

Performs inverse quantization of LSFs.

Syntax

```
IppStatus ippsLSFDecode_G723_16s (const Ipp16s* quantIndex, const Ipp16s*
    pPrevLSF, int erase, Ipp16s* pQuantLSF);
```

Parameters

quantIndex	Pointer to the input LSP VQ indices vector [3].
pPrevLSF	Pointer to the input quantized LSF for previous subframes.
erase	Indicates frame erasure.
pQuantLSF	Pointer to the output quantized LSF for previous subframes.

Description

The function `ippsLSFDecode_G723_16s` is declared in the `ippsc.h` file. This function performs inverse quantization of LSFs (in other words, decodes LSP VQ indices). First three VQ table entries corresponding to the transmitted index are found. The predicted vector is added to the found vector and DC vector to form the decoded vectors in three bands separately. A stability check is performed to ensure the difference between them not is less then 31.25 Hz.

For erased frame, zero predicted vectors are chosen and stability check for the difference in 62.5 Hz is applied.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *quantIndex*, *pPrevLSF* or *pQuantLSF* pointer is `NULL`.
`ippStsLSFLow`	Indicates a warning when a stability condition is not met.

LSFQuant_G723

Quantizes LSF coefficients.

Syntax

```
IppStatus ippsLSFQuant_G723_16s32s(const Ipp16s* pSrcLSF, const Ipp16s*
    pSrcPrevLSF, Ipp32s* pResultQLSFIndex);
```

Parameters

pSrcLSF	Pointer to the LSF coefficients vector [10], in Q15.
pSrcPrevLSF	Pointer to the previous LSF coefficients vector [10], in Q15.
pResultQLSFIndex	Pointer to the combined index of quantized LSF coefficients. The combination is built by left-shifting the first codebook index by 16 bits and left-shifting the second codebook index by 8 bits, and then adding together the two shifted indices and the third codebook index.

Description

The function `ippsLSFQuant_G723` is declared in the `ippsc.h` file. This function quantizes the LSF coefficients to obtain the codebook indices using PSVQ. It implements the following operations:

1. Calculate the diagonal weighting matrix W, determined from the unquantized LSF coefficients, as:

$$w_{1,\ 1} = 1/\ (\omega_2 - \omega_1)$$

$$w_{10,\ 10} = 1/\ (\omega_{10} - \omega_9)$$

$$w_{j,\ j} = 1/\ \min(\omega_j - \omega_{j-1},\ \omega_{j+1} - \omega_j)\ ,\ j = 2\ldots9$$

2. Calculate the prediction LSF coefficients as given by:

$$\omega_p = (\omega - \omega_{DC}) - 0.375(\omega_{-1} - \omega_{DC})$$

where ω are the current LSF coefficients, ω_{-1} are the previous LSF coefficients, and ω_{DC} are the DC LSF coefficients.

3. Search the codebook vector to minimize the following error:

$$E_l = (\omega_p - \omega_l)^T W (\omega_p - \omega_l)\ ,$$

where ω_l is the l-th code vector in the codebook. The quantization uses 3-3-4 split.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcLSF*, *pSrcPrevLSF*, or *pResultQLSFIndex* pointer is NULL.

Codebook Search Functions

These functions perform open loop pitch estimation using the adaptive codebook, and search in ACELP excitation (fixed) codebook for optimal signs and positions of the pulses.

OpenLoopPitchSearch_G723

Searches for an optimal pitch value.

Syntax

```
IppStatus ippsOpenLoopPitchSearch_G723_16s(const Ipp16s* pSrcWgtSpch, Ipp16s*
    pResultOpenDelay);
```

Parameters

pSrcWgtSpch	Pointer to the perceptually weighted speech signal. The signal length is 265, and the pointer *pSrcWgtSpch* points to its 146th element, in Q12.
pResultOpenDelay	Pointer to the open-loop pitch search result.

Description

The function ippsOpenLoopPitchSearch_G723 is declared in the ippsc.h file. This function extracts the open loop pitch from the weighted speech signal. It is applied in half-frames as follows:

1. The open loop pitch is chosen such that the cross-correlation is maximized:

$$C_{ol}(j) = \frac{\left[\sum_{i=0}^{119} s(i)s(i-j)\right]^2}{\sum_{i=0}^{119} s(i-j)s(i-j)}, \quad j = 18,\ldots,142$$

2. During the search, preference is given to smaller pitch periods to avoid choosing pitch multiples. Every found pitch value is compared with the previous best. If the difference is less than 18 and $C_{o1}(j) > C_{o1}(j')$, the process is over. Otherwise, the pitch is chosen only if $C_{o1}(j)$ is greater than $C_{o1}(j')$ by 1.25db.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcWgtSpch* or *pResultOpenDelay* pointer is NULL.

ACELPFixedCodebookSearch_G723

Searches the ACELP fixed codebook for the excitation.

Syntax

```
IppStatus ippsACELPFixedCodebookSearch_G723_16s(const Ipp16s* pSrcFixedCorr,
    const Ipp16s* pSrcMatrix, Ipp16s* pDstFixedSign, Ipp16s* pDstFixedPosition,
    Ipp16s* pResultGrid, Ipp16s* pDstFixedVector, Ipp16s* pSearchTimes);
IppStatus ippsACELPFixedCodebookSearch_G723_32s16s(const Ipp16s* pSrcFixedCorr,
    Ipp32s* pSrcDstMatrix, Ipp16s* pDstFixedSign, Ipp16s* pDstFixedPosition,
    Ipp16s* pResultGrid, Ipp16s* pDstFixedVector, Ipp16s* pSearchTimes);
```

Parameters

pSrcFixedCorr	Pointer to the correlation between residual and impulse response vector [60]
pSrcMatrix	Pointer to the elements of Toepliz matrix [416].
pSrcDstMatrix	Pointer to the input and output elements of Toepliz matrix [416].
pDstFixedSign	Pointer to the signs of the fixed vector [4].
pDstFixedPosition	Pointer to the positions of the fixed vector [4].
pResultGrid	Pointer to the beginning grid location.
pDstFixedVector	Pointer to the fixed vector [60].
pSearchTimes	Pointer to the input and output maximum search time

Description

The function `ippsACELPFixedCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches the ACELP fixed codebook for the excitation in the 5.3Kbps encoder. It is applied in subframes as follows:

1. There are four non-zero pulses in the fixed vector as given by:

$$c(n) = \sum_{k=0}^{3} \alpha_k \delta(n - m_k) \quad , \quad n = 0,\ldots,59$$

where α_k, $k = 0..3$ and m_k, $k = 0..3$ are the signs and positions of the fixed vector respectively.

2. Search for the parameters α_k, $k = 0..3$ and m_k, $k = 0..3$, that minimize the following error:

$$\delta = \Phi(m_0, m_0) + \Phi(m_1, m_1) + 2\alpha_0\alpha_1\Phi(m_0, m_1) +$$
$$\Phi(m_2, m_2) + 2[\alpha_0\alpha_2\Phi(m_0, m_2) + \alpha_1\alpha_2\Phi(m_1, m_2)] +$$
$$\Phi(m_3, m_3) + 2[\alpha_0\alpha_3\Phi(m_0, m_3) + \alpha_1\alpha_3\Phi(m_1, m_3) + \alpha_2\alpha_3\Phi(m_2, m_3)]$$

where $\Phi(i, j)$ is the Toepliz matrix.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcFixedCorr*, *pSrcMatrix*, *pSrcDstMatrix*, *pSearchTimes*, *pDstFixedSign*, *pDstFixedPosition*, *pResultGrid*, or *pDstFixedVector* pointer is `NULL`.

AdaptiveCodebookSearch_G723

Searches for the close loop pitch and the adaptive gain index.

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_G723(Ipp16s valBaseDelay,
    const Ipp16s*  pSrcAdptTarget, const Ipp16s*  pSrcImpulseResponse,
    Ipp16s*  pSrcPrevExcitation, const Ipp32s* pSrcPrevError,
    Ipp16s*  pResultCloseLag, Ipp16s*  pResultAdptGainIndex, Ipp16s subFrame,
    Ipp16s sineDtct, IppSpchBitRate bitRate);
```

Parameters

valBaseDelay	Base delay, in the range [18,145].
pSrcAdptTarget	Pointer to the adaptive target signal vector [60].
pSrcImpulseResponse	Pointer to the impulse response vector [60].
pSrcPrevExcitation	Pointer to the previous excitation vector [145].
pSrcPrevError	Pointer to the previous error vector [5].
subFrame	Subframe number, from 0 to 3.
bitRate	Transmit bit rate, equal to either `IPP_SPCHBR_6300` or `IPP_SPCHBR_5300`.
sineDtct	Sine detector parameter.
pResultCloseLag	Pointer to the lag of close pitch.
pResultAdptGainIndex	Pointer to the index of the adaptive gain.

Description

The function `ippsAdaptiveCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches for the close loop pitch and the adaptive gain index. It is applied in subframes as follows:

1. For subframe 0 and 2, the close loop pitch lag is chosen in the range of [-1,1] around the appropriate open loop pitch. For subframe 1 and 3, the range is [-1, 3] around the open loop pitch.

2. Denote the close loop pitch as L_i , i = 0...3. The pitch predictor gains are vector quantized. For 6.3Kbps bit rate, two codebooks are used with 85 entries and 170 entries, respectively. If L_0 is less than 58 for subframe 0 and 1, or if L_2 is less than 58 for subframe

2 and 3, then the 85-entry codebook is used for pitch gain quantization. Otherwise, the 170-entry codebook is used. For 5.3Kbps bit rate, the quantization is against a 170-entry codebook.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcAdptTarget*, *pSrcImpulseResponse*, *pSrcPrevExcitation*, *pSrcPrevError*, *pResultCloseLag*, or *pResultAdptGainIndex* pointer is NULL.
ippStsRangeErr	Indicates an error when *valBaseDelay* is not in the range [18, 145], or *subFrame* is not in the range [0, 3], or *bitRate* is not a valid element of the enumerated type IppSpchBitRate.

MPMLQFixedCodebookSearch_G723

Searches the MP-MLQ fixed codebook for the excitation.

Syntax

```
ppStatus ippsMPMLQFixedCodebookSearch_G723(Ipp16s valBaseDelay, const Ipp16s*
    pSrcImpulseResponse, const Ipp16s* pSrcResidualTarget, Ipp16s*
    pDstFixedVector, Ipp16s* pResultGrid, Ipp16s* pResultTrainDirac, Ipp16s*
    pResultAmpIndex, Ipp16s* pResultAmplitude, Ipp32s* pResultPosition, Ipp16s
    subFrame);
```

Parameters

valBaseDelay	Base delay, in the range [18,145].
pSrcImpulseResponse	Pointer to the impulse response vector [60].
pSrcResidualTarget	Pointer to the residual target signal vector [60].
pDstFixedVector	Pointer to the fixed codebook vector [60].
pResultGrid	Pointer to the beginning grid location, 0 or 1.
pResultTrainDirac	Pointer to the flag to show if train Dirac function is used: 0-unused, 1-used.
pResultAmpIndex	Pointer to the index of the quantized amplitude.
pResultAmplitude	Pointer to the amplitude of the fixed codebook vector.

pResultPosition	Pointer to the position of the fixed codebook vector, whose amplitude has non-zero value.
subFrame	Subframe number, from 0 to 3.

Description

The function `ippsMPMLQFixedCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches the MP-MLQ fixed codebook for the excitation in the 6.3Kbps encoder. It is applied in subframes as follows:

1. The error is obtained by the following calculation:

$$err(n) = res(n) - G\sum_{k=0}^{M-1}\alpha_k h(n-m_k)$$

where G is the gain factor, α_k and m_k are the signs and positions of the fixed vector, respectively.

2. Search for the parameters G, α_k and m_k that minimize the mean square of the error signal $err(n)$.

3. The fixed codebook gain is obtained using the following formulae:

$$d(j) = \sum_{n=j}^{59} c(n)\cdot h(n-j)$$

$$Gm = \frac{\max\{|d(j)|\}_{j=0..59}}{\sum_{n=0}^{59} h(n)\cdot h(n)},$$

where $c(n)$ is the fixed vector.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcImpulseResponse*, *pSrcResidualTarget*, *pDstFixedVector*, *pResultGrid*, *pResultTrainDirac*, *pResultAmpIndex*, *pResultAmplitude*, *or pResultPosition* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *subFrame* is not one of 0, 1, 2, or 3; or when *valBaseDelay* is not in the range [18,145].

ToeplizMatrix_G723

Calculates 416 elements of the Toepliz matrix for fixed codebook search.

Syntax

```
IppStatus ippsToeplizMatrix_G723_16s(const Ipp16s*  pSrcImpulseResponse,
    Ipp16s*  pDstMatrix);
IppStatus ippsToeplizMatrix_G723_16s32s(const Ipp16s*  pSrcImpulseResponse,
    Ipp32s*  pDstMatrix);
```

Parameters

pSrcImpulseResponse	Pointer to the impulse response vector [60].
pDstMatrix	Pointer to the elements of Toepliz matrix vector [416].

Description

The function `ippsToeplizMatrix_G723` is declared in the `ippsc.h` file. This function calculates the 416 elements in Toepliz matrix for the fixed-codebook search. Elements of the Toepliz matrix can be expressed as follows:

$$\Phi(i,\ j) = \sum_{n=j}^{59} h(n-i) \times h(n-j)\ ,\ i \le j\ ,\ 0 \le i \le 59\ ,$$

where $h(i)$, i = 0, 1, …, 59, is the impulse response.

The function stores the calculated 416 elements in *pDstMatrix* in the following order:

1. $\Phi(m_i,\ m_i)$, (i = 0, 1, 2, 3), 4x8 =32 elements; starting location: 0.
2. $\Phi(m_0,\ m_1)$, 8x8 = 64 elements; starting location: 32.
3. $\Phi(m_0,\ m_2)$, 8x8 = 64 elements; starting location: 96.
4. $\Phi(m_0,\ m_3)$, 8x8 = 64 elements; starting location: 160.
5. $\Phi(m_1,\ m_2)$, 8x8 = 64 elements; starting location: 224.
6. $\Phi(m_1,\ m_3)$, 8x8 = 64 elements; starting location: 288.
7. $\Phi(m_2,\ m_3)$, 8x8 = 64 elements; starting location: 352.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcImpulseResponse* or *pDstMatrix* pointer is NULL.

Gain Quantization

These functions perform fixed codebook gain estimation and gain adjustment of the postfiltered signal.

GainQuant_G723

Implements MP-MLQ gain estimation and quantization.

Syntax

```
IppStatus ippsGainQuant_G723_16s (const Ipp16s* pImp, const Ipp16s* pSrc,
    Ipp16s* pDstLoc, Ipp16s* pDstAmp, Ipp32s* pMaxErr, Ipp16s* pGrid, Ipp16s*
    pAmp, int Np, int* isBest);
```

Parameters

pImp	Pointer to the input impulse response *h* vector [60].
pSrc	Pointer to the input target signal *r* vector [60].
pDstLoc	Pointer to the output pulse locations.
pDstAmp	Pointer to the output pulse amplitudes.
pMaxErr	Pointer to the input/output quantization error.
pGrid	Pointer to the output grid: 0 - if pulses are in even positions, 1 - if pulses are in odd positions.
pAmp	Pointer to the output index of maximum amplitude.
Np	Number of pulses: equal to 6 for even subframe, equal to 5 for odd subframe.
isBest	Indicates quantization result. Set to 0, if no pulses are found with quantization error better (less) than the input error.

Description

The function `ippsGainQuant_G723_16s` is declared in the `ippsc.h` file. This function estimates the unknown parameters, G, $\{\alpha_k\}$, $\{m_k\}$, $k = 0,\ldots,Np-1$, that minimize the mean square of the error signal:

$$err[n] = r[n] - G \cdot \sum_{k=0}^{Np-1} a_k \cdot h[n - m_k]$$

The estimation is done as follows.

First, cross correlation of the impulse response and the target vector is computed:

$$d[j] = \sum_{n=j}^{59} r[n] \cdot h[n-j] \ , \ 0 \le n \le 59$$

The estimated maximum gain is computed as:

$$G_{max} = \frac{\max\{|d[j]|\}_{j=0..59}}{\sum_{n=0}^{59} h[n] \cdot h[n]}$$

and then is quantized by the logarithmic quantizer. The gain values selected around this quantized gain are then used to optimize the signs and locations of the pulses for both even and odd grids that yield the minimum mean square of the error signal.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pImp*, *pSrc*, *pDstLoc*, *pDstAmp*, *pMaxErr*, *pGrid*, *pAmp*, or *isBest* pointer is `NULL`.

GainControl_G723

Extracts delayed pitch contribution.

Syntax

```
IppStatus ippsGainControl_G723_16s_I (Ipp32s energy, Ipp16s* pSrcDst, Ipp16s*
    pGain);
```

Parameters

energy	The input energy coefficient.
pSrcDst	Pointer to the input/output post-filtered signal.
pGain	Pointer to the input/output gain.

Description

The function `ippsGainControl_G723_16s_I` is declared in the `ippsc.h` file. This function first computes the amplitude ratio g_s as:

$$g_s = \sqrt{\frac{energy}{\sum_{n=0}^{59} pSrcDst[n] \cdot pSrcDst[n]}}$$

If denominator in this expression is equal to 0, g_s is set to 1.

Then the input postfiltered signal is scaled as follows:

$pSrcDst[n] = pSrcDst[n] \cdot g_n \cdot (1+\alpha)$, $n = 0,\ldots, 59$,

where g_n is updated using the following expression, respectively:

$g_n = (1-\alpha) \cdot g_{n-1} + \alpha \cdot g_s$, $n = 0,\ldots, 59$, $g_{-1} = 0$

and α is equal to 1/16.

The output gain is equal to g_{59} .

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst* or *pGain* pointer is `NULL`.

Filter Functions

These functions implement different types of filtering, including high pass filtering on pre-processing stage of encoding and post-processing stage of decoding, as well as the postfilter in final decode stage.
Different kinds of synthesis (IIR), harmonic, and preemphasize filter functions can be combined to perform more complex filtering and may be used in different encode and decode stages.

HighPassFilter_G723

Performs high-pass filtering of the input signal.

Syntax

```
IppStatus ippsHighPassFilter_G723_16s (const Ipp16s* pSrc, Ipp16s* pDst, int*
    pMem);
```

Parameters

pSrc	Pointer to the source vector [240].
pDst	Pointer to the destination vector [240].
pMem	Pointer to the filter memory vector [2]. This vector must be initially filled with zeroes.

Description

The function `ippsHighPassFilter_G723` is declared in the `ippsc.h` file. This function pre-processes the input signal using the following filter transfer function:

$$H_h(z) = \frac{1 - z^{-1}}{1 - \frac{127}{128} z^{-1}}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst* or *pMem* pointer is `NULL`.

IIR16s_G723

Performs IIR filtering.

Syntax

```
IppStatus ippsIIR16s_G723_16s32s (const Ipp16s* pCoeffs, const Ipp16s* pSrc,
    Ipp32s* pDst, Ipp16s* pMem);
IppStatus ippsIIR16s_G723_16s_I (const Ipp16s* pCoeffs, Ipp16s* pSrcDst,
    Ipp16s* pMem);
IppStatus ippsIIR16s_G723_32s16s_Sfs (const Ipp16s* pCoeffs, const Ipp32s*
    pSrc, int scaleFactor, Ipp16s* pDst, Ipp16s* pMem);
```

Parameters

pCoeffs	Pointer to the input vector of filter coefficients vector [20]: $b_1,\ldots, b_{10}, a_1, \ldots,a_{10}$.
pSrc	Pointer to the input signal vector [60].
pSrcDst	Pointer to the input/output signal vector [60].
scaleFactor	Scale factor.
pDst	Pointer to the output filtered vector [60].
pMem	Pointer to the filter memory vector [20]. This vector must be initially filled with zeroes.

Description

These functions are declared in the `ippsc.h` file.

The functions `ippsIIR16s_G723_16s32s` and `ippsIIR16s_G723_16s_I` perform infinite impulse response (IIR) filtering using the following transfer function:

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The function `ippsIIR16s_G723_32s16s_Sfs` performs IIR filtering using the transfer function:

$$H(z) = 2^{sFs} \cdot \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The filter memory is updated.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pCoeffs*, *pSrc*, *pSrcDst*, *pDst*, or *pMem* pointer is NULL.

SynthesisFilter_G723

Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).

Syntax

```
IppStatus ippsSynthesisFilter_G723_16s32s (const Ipp16s* pLPC, const Ipp16s*
    pSrc, Ipp32s* pDst, Ipp16s* pMem);
IppStatus ippsSynthesisFilter_G723_16s (const Ipp16s* pLPC, const Ipp16s*
    pSrc, Ipp16s*  pMem, Ipp16s* pDst);
```

Parameters

pLPC	Pointer to the input LP coefficients a_0, a_1,...,a_{10}, in Q11.
pSrc	Pointer to the source vector.
pDst	Pointer to the filtered output.
pMem	Pointer to the memory supplied for filtering: short integer vector [10], initially set to zero.

Description

The function ippsSynthesisFilter_G723 is declared in the ippsc.h file. This function computes the filter given by:

$$H(z) = \hat{A}^{-1}(z) = \frac{1}{a_0 + \sum_{i=1}^{10} a_i \cdot \hat{z}^{-i}}$$

This function is applied after the residual filter in computing the perceptually weighted speech signal:

$$pDst[n] = pSrc[n] \cdot pLPC[0] + \sum_{i=1}^{10} pLPC[i] \cdot pMem[n-i+1] \quad , \quad n = 0,\ldots,59$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pLPC*, *pDst*, or *pMem* pointer is NULL.
ippStsOverflow	Indicates a warning that at least one result value was saturated.

TiltCompensation_G723

Computes tilt compensation filter.

Syntax

```
IppStatus ippsTiltCompensation_G723_32s16s (Ipp16s val, const Ipp32s* pSrc,
    Ipp16s* pDst);
```

Parameters

val	The first-order partial correlation coefficient, in Q15.
pSrc	Pointer to the source vector [61].
pDst	Pointer to the output filtered vector [60].

Description

The function ippsTiltCompensation_G723 is declared in the ippsc.h file. This function computes the tilt compensation filter as:

$$H_t(z) = (1 - val \cdot z^{-1})$$

$$pDst[i] = pSrc[i+1] + pSrc[i] \cdot val \quad , \quad i = 0,\ldots,59$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.

HarmonicSearch_G723

Searches for the harmonic delay and gain for the harmonic noise shaping filter.

Syntax

```
IppStatus ippsHarmonicSearch_G723_16s(Ipp16s valOpenDelay, const Ipp16s*
    pSrcWgtSpch, Ipp16s* pResultHarmonicDelay, Ipp16s* pResultHarmonicGain);
```

Parameters

valOpenDelay	Open loop pitch, in the range [18,145].
pSrcWgtSpch	Pointer to the weighted speech vector [205]. The pointer points to the 146th element, in Q12.
pResultHarmonicDelay	Pointer to the output harmonic delay.
pResultHarmonicGain	Pointer to the output harmonic gain, in Q15.

Description

The function ippsHarmonicSearch_G723 is declared in the ippsc.h file. This function searches for the harmonic delay and harmonic gain for the harmonic noise shaping filter, using the weighted speech and open loop pitch as input.
This function is applied in subframes as follows:

1. Find the index *L* to optimize the following expression:

$$C_{pw}(j) = \left[\sum_{i=0}^{59} s(i)s(i-j)\right]^2 / \left(\sum_{i=0}^{59} s(i-j)s(i-j)\right), \quad j = pitch-3..pitch+3$$

2. Calculate the optimal filter gain as follows:

$$G_{opt} = \sum_{i=0}^{59} s(i)s(i-j) / \sum_{i=0}^{59} s(i-j)s(i-j)$$

G_{opt} is saturated to [0,1].

3. Calculate the energy of the weighted speech samples and the harmonic gain:

$$E = \sum_{i=0}^{59} s^2(i)$$

$$\beta = \begin{cases} 0.3125 G_{opt}, & if\ -10\log 10(1 - C_L / E) \geq 2.0 \\ 0, & otherwise \end{cases}$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcWgtSpch* , *pResultHarmonicDelay*, or *pResultHarmonicGain* pointer is NULL.
ippStsRangeErr	Indicates an error when *valOpenDelay* is not in the range [18, 145].

HarmonicNoiseSubtract_G723

Performs harmonic noise shaping.

Syntax

```
IppStatus ippsHarmonicNoiseSubtract_G723_16s_I(Ipp16s val, int T, const Ipp16s*
    pSrc, Ipp16s* pSrcDst);
```

Parameters

val	The input harmonic filter coefficient, in Q15.
T	The input harmonic filter lag.
pSrc	Pointer to the input zero impulse response vector [60] of the combined filter.
pSrcDst	Pointer to the input/output harmonic noise weighted speech vector [60].

Description

The function `ippsHarmonicNoiseSubtract_G723_16s_I` is declared in the `ippsc.h` file. This function subtracts the harmonic shaped vector *pSrc* from vector *pSrcDst*, as follows:

$$pSrcDst[n] = pSrcDst[n] - (pSrc[n] + val \cdot pSrc[n-T]) \ , \ n = 0,\ldots, 59$$

This operation is used for ringing subtraction, which is performed by subtracting the zero impulse response from the harmonic weighted speech vector to obtain the target vector:

$$t[n] = w[n] - z[n]$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pSrcDst* pointer is `NULL`.

DecodeAdaptiveVector_G723

Restores the adaptive codebook vector from excitation, pitch, and adaptive gain.

Syntax

```
IppStatus ippsDecodeAdaptiveVector_G723_16s(Ipp16s valBaseDelay, Ipp16s
    valCloseLag, Ipp16s valAdptGainIndex, const Ipp16s* pSrcPrevExcitation,
    Ipp16s* pDstAdptVector, IppSpchBitRate bitRate);
```

Parameters

valBaseDelay	Base delay, in the range [18,145].
valCloseLag	Lag of close pitch.
valAdptGainIndex	Index of adaptive gain.
pSrcPrevExcitation	Pointer to the past excitations vector [145].
bitRate	Transmit bit rate, equal to either `IPP_SPCHBR_6300` or `IPP_SPCHBR_5300`.
pDstAdptVector	Pointer to the adaptive codebook vector [60].

Description

The function `ippsDecodeAdaptiveVector_G723` is declared in the `ippsc.h` file. This function decodes the adaptive vector from excitation, close loop pitch, and adaptive gain index. It is applied in subframes as follows:

1. Obtain the pitch as:

$$L = L_o + L_c - 1 \ ,$$

where L_o is the open loop pitch and L_c is the close loop pitch lag.

2. Obtain the residual from the previous excitation using the following relations:

res(0) = *exci*(143-*L*);

res(1) = *exci*(143-*L*+1);

When decoding a second subframe, the sum (L_o+L_c) can be equal to 146, so elements *exci*(-2) and *exci*(-1) must be available.

$$res(i+2) = exci(145 - L + (i\%L)) \ , \ i = 0...61$$

3. Select the adaptive codebook, G_b, from the 85-or 170-entry codebooks.

4. Obtain the adaptive vector using the formula:

$$c(i) = \sum_{j=0}^{d} res(i+j)G_b(20k+i) \ , \ i = 0...59$$

where k is the adaptive gain index.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcPrevExcitation* or *pDstAdptVector* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *valBaseDelay* is not in the range [18, 145]; or *valCloseLag* is not in the range [0, 2]; or *valAdptGainIndex* is not in the range [0, 169], or *bitRate* is not a valid element of the enumerated type `IppSpchBitRate`.

PitchPostFilter_G723

Calculates coefficients of the pitch post filter.

Syntax

```
IppStatus ippsPitchPostFilter_G723_16s(Ipp16s valBaseDelay, const Ipp16s*
    pSrcResidual, Ipp16s* pResultDelay, Ipp16s* pResultPitchGain, Ipp16s*
    pResultScalingGain, Ipp16s subFrame, IppSpchBitRate bitRate);
```

Parameters

valBaseDelay	Base delay, in the range [18,145].
pSrcResidual	Pointer to the residual signal vector [365]. This pointer points to the 146th element.
pResultDelay	Pointer to the delay of the pitch post filter.
pResultPitchGain	Pointer to the gain of the pitch post filter, in Q15.
pResultScalingGain	Pointer to the scaling gain of the pitch post filter, in Q15.
subFrame	Subframe number, from 0 to 3.
bitRate	Transmit bit rate, equal to either `IPP_SPCHBR_6300` or `IPP_SPCHBR_5300`.

Description

The function `ippsPitchPostFilter_G723` is declared in the `ippsc.h` file. This function calculates the coefficients of the pitch post filter. It is applied in subframes as follows:

1. Search for the forward pitch lag M_f from the forward cross-correlation:

$$C_f = \sum_{i=0}^{59} res(i)res(i+M_f) \ , \ M_1 \le M_f \le M_2$$

where $M_1 = valBaseDelay - 3$ and $M_2 = valBaseDelay + 3$.

2. Search for the backward pitch lag M_b from the backward cross-correlation:

$$C_b = \sum_{i=0}^{59} res(i)res(i-M_b) \ , \ M_1 \le M_b \le M_2$$

3. If C_f or C_b is negative, then the corresponding weight and delay are set to zero.

4. Calculate the relevant residual energy as follows:

$$D_f = \sum_{i=0}^{59} res(i+M_f)res(i+M_f)$$

$$D_b = \sum_{i=0}^{59} res(i-M_b)res(i-M_b)$$

$$T_{en} = \sum_{i=0}^{59} res(i)res(i)$$

5. Obtain the optimal gain: $g = C/D$

6. Calculate the scaling gain using the following formula:

$$g_s = \sqrt{\sum_{i=0}^{59} res^2(i) / \sum_{i=0}^{59} res'^{\,2}(i)}$$

where $res'(i)$ is the post filtered residual.

7. Calculate the total gain of the post filter:

$$g_p = \gamma_{ltp} \cdot g_p \cdot g$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcResidual*, *pResultDelay*, *pResultPitchGain*, *or pResultScalingGain* pointer is NULL.
ippStsRangeErr	Indicates an error when *valBaseDelay* is not in the range [18, 145], or *subFrame* is not 0 or 3, or *bitRate* is not a valid element of the enumerated type IppSpchBitRate.

GSM-AMR Related Functions

This part of the chapter describes Intel IPP functions that can be combined to construct a bit-exact implementation of the European Telecommunications Standards Institute (ETSI) Global System for Mobile Communications (GSM) Adaptive Multi-Rate (AMR) ETSI EN 301 704 GSM 06.90 version 7.5.0 Release 2001 speech codec, more commonly known as the "GSM-AMR 06.90" codec. The primitives are primarily concerned with the well-defined, computationally expensive core operations that comprise the codec portion of the GSM-AMR system.

The GSM 06.90 AMR speech codec comprises an adaptive multi-rate algorithm that represents efficiently telephone-bandwidth digital speech using compressed data rates of 4.75, 5.15, 5.90, 6.70, 7.40, 7.95, 10.2, and 12.2 kilobits per second (kbps). The GSM-AMR system adaptively controls speech codec bit rates such that output quality is maximized for a given set of channel conditions.

Table 9-4 shows the functional grouping of the GSM-AMR primitives.

Table 9-4 Summary and Functional Groupings of the GSM-AMR Codec Primitives

Function Subset	Function Name
Basic functions	Interpolate_GSMAMR FFTFwd_RToPerm_GSMAMR
LP analysis	AutoCorr_GSMAMR LevinsonDurbin_GSMAMR LPCToLSP_GSMAMR LSPToLPC_GSMAMR LSFToLSP_GSMAMR LSPQuant_GSMAMR QuantLSPDecode_GSMAMR
Adaptive-codebook search	OpenLoopPitchSearchNonDTX_GSMAMR OpenLoopPitchSearchDTXVAD1_GSMAMR OpenLoopPitchSearchDTXVAD2_GSMAMR ImpulseResponseTarget_GSMAMR AdaptiveCodebookSearch_GSMAMR AdaptiveCodebookDecode_GSMAMR AdaptiveCodebookGain_GSMAMR

Table 9-4 Summary and Functional Groupings of the GSM-AMR Codec Primitives

Fixed-codebook search	AlgebraicCodebookSearch_GSMAMR FixedCodebookDecode_GSMAMR
Discontinuous transmission	Preemphasize_GSMAMR VAD1_GSMAMR VAD2_GSMAMR EncDTXSID_GSMAMR EncDTXHandler_GSMAMR EncDTXBuffer_GSMAMR , DecDTXBuffer_GSMAMR
Post-processing	PostFilter_GSMAMR

Sections that follow provide details on the primitives that implement each of the GSM-AMR functional blocks summarized in Table 9-4, specifically:

Section Basic Functions describes some general primitives.

Section LP Analysis and Quantization Functions describes LP analysis primitives.

Sections Adaptive Codebook Functions and Fixed Codebook Search give details on primitives provided for the adaptive and fixed codebook search procedures, respectively.

Section Discontinuous Transmission (DTX) is concerned with primitives for discontinuous transmission (DTX), including functions that implement voice activity detection (VAD) algorithms 1 and 2, as well as primitives for comfort noise generation (CNG), and DTX buffer management.

Section Post Processing describes post-processing primitives.

These functions are used in the Intel IPP *GSM/AMR Speech Encoder-Decoder* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

Basic Functions

Interpolate_GSMAMR

Computes the weighted sum of two vectors.

Syntax

```
IppStatus ippsInterpolate_GSMAMR_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
```

Parameters

pSrc1	Pointer to the first input vector[*len*].
pSrc2	Pointer to the second input vector[*len*].
pDst	Pointer to the output vector[*len*].
len	Length of the input and output vectors.

Description

The function `ippsInterpolate_GSMAMR` is declared in `ippsc.h` file. This function computes the weighted sum of two vectors as:

$$pDst[i] = (pSrc1[i]>>2) + (pSrc2[i] - (pSrc2[i]>>2)); \quad 0 \le i < len$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

FFTFwd_RToPerm_GSMAMR

Computes the forward fast Fourier transform (FFT) of a real signal.

Syntax

```
IppStatus ippsFFTFwd_RToPerm_GSMAMR_16s_I (Ipp16s* pSrcDst);
```

Parameters

pSrcDst	Pointer to the input and output vector[128].

Description

The function `ippsFFTFwd_RToPerm_GSMAMR` is declared in `ippsc.h` file. This function computes the forward FFT of a real signal. This transform is the same as done by the general signal processing FFT function `ippsFFTFwd_RToPerm_16s_Sfs` with the following settings: `order` = 7, `flag` = `IPP_FFT_DIV_FWD_BY_N`, `scale factor` = –1. The result of function operation may be different because the rounding mode used in `ippsFFTFwd_RToPerm_16s_Sfs` does not guarantee a bit-exact operation of GSM AMR codec (see Fourier Transform Functions in chapter 7).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst* pointer is `NULL`.

LP Analysis and Quantization Functions

This section describes the GSM-AMR primitives that are concerned with LP analysis, quantization, encoding, and decoding. It also provides details on primitives that accomplish the following tasks:

- Autocorrelation analysis
- Levinson-Durbin algorithm
- LPC to LSP conversion
- LSP to LPC conversion
- LSP quantization and inverse quantization
- Quantized LSP encoding and decoding.

AutoCorr_GSMAMR

Estimates the autocorrelation sequence for a block of samples.

Syntax

```
IppStatus ippsAutoCorr_GSMAMR_16s32s(const Ipp16s*  pSrcSpch, Ipp32s*
    pDstAutoCorr, IppSpchBitRate mode);
```

Parameters

pSrcSpch	Pointer to the input speech vector (240 samples), represented using Q15.0. This should be aligned on an 8-byte boundary.
pDstAutoCorr	Pointer to the autocorrelation coefficients, of length 22. For 12.2 kbps mode, elements 0 ~ 10 contain the first set of autocorrelation lags, and elements 11 ~ 21 contain the second set of autocorrelation lags. For all other modes there is only one set of autocorrelation lags contained in vector elements 0 ~ 10.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Description

The function `ippsAutoCorr_GSMAMR` is declared in `ippsc.h` file. This function estimates the autocorrelation sequence for a block of 240 samples (30 ms). For the12.2 kbps mode, the 160 samples associated with the current 20 ms frame are combined with the last 80

samples from the previous frame, and two autocorrelation sequences are estimated. For all other bit rates, 160 samples from the current frame are combined with the last 40 samples from the previous frame as well as the first 40 samples from the next frame, and only one autocorrelation sequence is estimated. In particular, the estimates are formed as follows:

1. Tapered windowing – asymmetric tapered windows are applied to the input speech. Different windows are selected depending on the bit rate. For 12.2 kbps frames, unique tapered windows are applied for each of the two autocorrelation estimates, that is

$$w_1(n) = \begin{cases} 0.54 - 0.46\cos\dfrac{\pi n}{159}, & n = 0, 1, \ldots, 159 \\ 0.54 - 0.46\cos\dfrac{(n-160)\pi}{79}, & n = 160, 161, \ldots, 239 \end{cases}$$

and

$$w_2(n) = \begin{cases} 0.54 - 0.46\cos\dfrac{2\pi n}{463}, & n = 0, 1, \ldots, 231 \\ \cos\dfrac{2\pi(n-232)}{31}, & n = 232, 233, \ldots, 239 \end{cases}$$

Neither w_1 nor w_2 has any look ahead. For all bit rates other than 12.2 kbps, a window centered on the current frame is applied, i.e.,

$$w_3(n) = \begin{cases} 0.54 - 0.46\cos\dfrac{2\pi n}{399}, & n = 0, 1, \ldots, 199 \\ \cos\dfrac{2\pi(n-232)}{31}, & n = 200, 201, \ldots, 239 \end{cases}$$

2. Estimation of autocorrelation lags – autocorrelation lags are estimated from the windowed speech samples $s(i)$, $i = 0, 1, \ldots, 239$, as follows

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k) \;, \quad k = 0, 1, \ldots, 10$$

3. Bandwidth expansion – the following binomial lag window is applied to the autocorrelation sequence(s) obtained in step 2:

$$bi(i) = \exp\left[-0.5 \times \left(\frac{2\pi f_0 i}{f_S}\right)^2\right], \quad i = 0, 1, ..., 10$$

where f_0 = 60 Hz and f_S = 8000 Hz.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcSpch* or *pDstAutoCorr* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

LevinsonDurbin_GSMAMR

Calculates the LP coefficients using the Levinson-Durbin algorithm.

Syntax

```
IppStatus ippsLevinsonDurbin_GSMAMR(const Ipp32s*  pSrcAutoCorr,
    Ipp16s*  pSrcDstLpc);
IppStatus ippsLevinsonDurbin_GSMAMR_32s16s (const Ipp32s*  pSrcAutoCorr,
    Ipp16s*  pSrcDstLpc);
```

Parameters

pSrcAutoCorr	Pointer to the autocorrelation coefficients, a vector of length 11.
pSrcDstLpc	Pointer to the LP coefficients associated with the previous frame, a vector of length 11, represented using Q3.12. On output, updated to point to the LP coefficients associated with the current frame, a vector of length 11, represented using Q3.12.

Description

The function `ippsLevinsonDurbin_GSMAMR` is declared in `ippsc.h` file. This function calculates the 10th-order LP coefficients from the autocorrelation lags using the Levinson-Durbin algorithm. The detailed steps performed by the primitive are as follows:

1. Minimization of the prediction residual in the mean-square sense yields a system of linear equations that can be solved efficiently using the Levinson-Durbin algorithm to yield a set of LP coefficients, a_i, i = 1, 2, ..., 10, that is

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k) \quad , \; k = 1,2,\ldots,10 \; .$$

2. The Levinson-Durbin algorithm performs the following recursion to obtain the prediction coefficients:

$$E^{[0]} = r(o)$$

for i = 1 to 10

$$a_0^{[i-1]} = 1$$

$$k_i = -\left[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j)\right] / \; E^{[i-1]}$$

$$a_i^{[i]} = k_i$$

for j = 1 to i-1

$$a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$$

end

$$E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$$

end

3. Unstable synthesis filter exception handling: if autocorrelation analysis yields an unstable LPC synthesis filter, i.e., if $|\, k_i \,|$ is close to or larger than 1.0, then the LP coefficients associated with the previous frame should replace the LP coefficients estimated on the current frame.

Return Values

`ippStsNoErr`	Indicates no error.

`ippStsNullPtrErr`	Indicates an error when the *pSrcAutoCorr* or *pSrcDstLpc* pointer is `NULL`.

LPCToLSP_GSMAMR

Converts LP coefficients to line spectrum pairs.

Syntax

```
IppStatus ippsLPCToLSP_GSMAMR_16s(const Ipp16s*  pSrcLpc, const Ipp16s*
    pSrcPrevLsp, Ipp16s*  pDstLsp);
```

Parameters

pSrcLpc	Pointer to the eleven-element LP coefficient vector, represented using Q3.12.
pSrcPrevLsp	Pointer to the ten-element LSP coefficient vector associated with the previous frame, represented using Q0.15.
pDstLsp	Pointer to the ten-element LSP coefficient vector, represented using Q0.15.

Description

The function `ippsLPCToLSP_GSMAMR` is declared in `ippsc.h` file. This function converts a set of 10th-order LP coefficients to an equivalent set of line spectrum pairs (LSPs). The functionality is as follows:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$ using the recursive relations

$$f_1(i+1) = a_{i+1} + a_{i-1} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{i-1} + f_1(i) \qquad i = 0, 1, ..., 4$$

where $f_1(0) = f_2(0) = 1.0$.

2. Use Chebyshev polynomials to evaluate $F_1(z)$ and $F_2(z)$. The Chebyshev polynomials are given by the following formulas;

$$C_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + \frac{f_1(5)}{2}$$

$$C_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + \frac{f_2(5)}{2}$$

3. Evaluate $F_1(z)$ and $F_2(z)$ on 60 points equally spaced between 0 and π, checking for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided 4 times to track the root.
4. If 10 roots for LSP coefficients are not found during the search, the previous set of LSPs is used.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLpc*, *pSrcPrevLsp* or *pDstLsp* pointer is `NULL`.

LSPToLPC_GSMAMR

Converts LSPs to LP coefficients.

Syntax

```
IppStatus ippsLSPToLPC_GSMAMR_16s(const Ipp16s*  pSrcLsp, Ipp16s*  pDstLpc);
```

Parameters

pSrcLsp	Pointer to the ten-element LSP coefficient vector, represented using Q0.15.
pDstLpc	Pointer to the eleven-element LP coefficient vector, represented using Q3.12.

Description

The function `ippsLSPToLPC_GSMAMR` is declared in `ippsc.h` file. This function converts a set of 10th-order LSPs to LP coefficients. The functionality is as follows:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursive relations

```
for i = 1 to 5
```

$$f_1(i) = -2q_{2i-1} \times f_1(i-1) + 2f_1(i-2)$$

```
for j = i-1 down to 1
```

$$f_1(j) = f_1(j) - 2q_{2i-1} \times f_1(j-1) + 2f_1(j-2)$$

```
end
end
```

where initial values $f_1(0) = 1$ and $f_1(-1) = 0$. The coefficients $f_2(i)$ are computed similarly by replacing q_{2i-1} by q_{2i}.

2. Multiply $F_1(z)$ and $F_2(z)$ by $1+z^{-1}$ and $1-z^{-1}$, respectively, to obtain $F_1'(z)$ and $F_2'(z)$ using the following relations:

$$f_1'(i) = f_1(i) + f_1(i-1), \quad i = 1, 2, ..., 5$$

$$f_2'(i) = f_2(i) + f_2(i-1), \quad i = 1, 2, ..., 5$$

3. Compute the LP coefficients from the polynomials $F_1'(z)$ and $F_2'(z)$ as follows

$$a_i = \begin{cases} 0.5\, f_1'(i) + 0.5\, f_2'(i), & i = 1, 2, ..., 5 \\ 0.5\, f_1'(11-i) + 0.5\, f_2'(11-i), & i = 6, 7, ..., 11 \end{cases}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLsp* or *pDstLpc* pointer is `NULL`.

LSFToLSP_GSMAMR

Converts Line Spectral Frequencies to LSP coefficients.

Syntax

```
IppStatus ippsLSFToLSP_GSMAMR_16s (const Ipp16s* pLSF, Ipp16s* pLSP) ;
```

Parameters

pLSF	Pointer to the vector [10] of LSF normalized by factor 1/π in Q14, so given in the range [0,1].
pLSP	Pointer to the LSP vector [10], in Q15 in the range [-1,1].

Description

The function `ippsLSPToLSF_GSMAMR` is declared in `ippsc.h` file. This function converts the Line Spectral Frequencies (LSF) to the LSP coefficients as given by:

$$lsp_n = \cos(lsf_n), 1 \le n \le 10$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pLSF* or *pLSP* pointer is `NULL`.

LSPQuant_GSMAMR

Quantizes the LSP coefficient vector.

Syntax

```
IppStatus ippsLSPQuant_GSMAMR_16s(const Ipp16s*  pSrcLsp, Ipp16s*
    pSrcDstPrevQLsfResidual, Ipp16s*  pDstQLsp, Ipp16s*  pDstQLspIndex,
    IppSpchBitRate  mode);
```

Parameters

pSrcLsp	Pointer to the unquantized 20-element LSP vector, represented using Q0.15. For 12.2 kbps frames, the first LSP set is contained in vector elements 0 ~ 9, and the second LSP set is contained in vector elements 10 ~ 19. For all other bit rates, only elements 0 to 9 are valid and used for the quantization.
pSrcDstPrevQLsfResidual	Pointer to the ten-element quantized LSF residual from the previous frame, represented using Q0.15. On output, points to the ten-element quantized LSF residual for the current frame, represented using Q0.15.
pDstQLsp	Pointer to the 20-element quantized LSP vector, represented using Q0.15. For 12.2 kbps frames, elements 0 to 9 contain the first quantized LSP set, and elements 10 to 19 contain the second quantized LSP set. For all other bit rates there is only one LSP set contained in elements 0 to 9.
pDstQLspIndex	Pointer to the five-element vector of quantized LSP indices. For 12.2Kbps frames, all five elements contain valid data; for all other bit rates, only the first three elements contain valid indices (see the following discussion of SMQ and SVQ for the various modes).
mode	Bit rate specifier. The enumerated values of `IPP_SPCHBR_4750` to `IPP_SPCHBR_12200` are valid.

Description

The function `ippsLSPQuant_GSMAMR` is declared in `ippsc.h` file. This function quantizes the LSP coefficient vector, then obtains quantized LSP codebook indices. The functionality can be summarized as follows:

1. LSP-to-LSF conversion – the LSPs are mapped to line spectrum frequencies (LSFs). For 12.2 kbps frames, two sets of LSPs are converted to LSFs. For all other bit rates, a single set of LSPs is converted using the following relation:

$$f_i = \frac{f_S}{2\pi} arc\cos(q_i)$$

2. LSF prediction – first-order moving average (MA) prediction is applied to the LSF vectors. Then, the prediction residual is quantized.
   For 12.2 kbps frames, two prediction residual vectors $r^{(1)}(n)$ and $r^{(2)}(n)$ are formed as follows:

$$r^{(1)}(n) = z^{(1)}(n) - 0.65\hat{r}^{(1)}(n-1), \qquad r^{(2)}(n) = z^{(2)}(n) - 0.65\hat{r}^{(2)}(n-1)$$

where $z^{(1)}(n)$ and $z^{(2)}(n)$ are the zero-mean LSF vectors at frame n and $\hat{r}^{(2)}(n-1)$ is the quantized residual vector from frame n-1.

For all other bit rates, a signal prediction residual vector, $r(n)$, is given by:

$$r_j(n) = z_j(n) - \alpha_j \hat{r}_j(n-1), \quad j = 0, \ldots, 9$$

where $z(n)$ is the mean-removed LSF vector at frame n, and $\hat{r}(n-1)$ is the quantized residual vector from frame n-1.

3. Quantization – Split matrix quantization (SMQ) is applied to the prediction residual vectors. For 12.2 kbps frames, the matrix ($r^{(1)}$, $r^{(2)}$) is split into 5 submatrices of dimension 2x2, and the 5 submatrices are quantized with 7, 8, 9, 8, and 6 bits, respectively. For all other bit rates, the vector r is split into 3 subvectors of dimension 3, 3, and 4. The three subvectors are quantized with 7-9 bits. The VQ search identifies the index k which minimizes the weighted error

$$E_{LSP} = \sum_{i=0}^{9} [r_i w_i - \hat{r}_i^k w_i]^2$$

where the weighting factor, w_i, is given by

$$w_i = \begin{cases} 3.347 - \frac{1.547}{450} d_i, & d_i < 450 \\ 1.8 - \frac{0.8}{1050}(d_i - 450), & \text{otherwise} \end{cases}$$

and $d_i = f_{i+1} - f_{i-1}$ with $f_0 = 0$, and $f_{11} = 4000$.

4. Resonance minimization – The quantized LSF vector elements are rearranged in order to avoid sharp resonances in the LP synthesis filter.
5. LSF-to-LSP conversion – convert the quantized LSFs to LSPs.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrcLsp*, *pSrcDstPrevQLsfResidual*, *pDstQLsp*, or *pDstQLspIndex* pointer is NULL.
ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.

QuantLSPDecode_GSMAMR

Decodes quantized LSPs.

Syntax

```
IppStatus ippsQuantLSPDecode_GSMAMR_16s(const Ipp16s*  pSrcQLspIndex, Ipp16s*
    pSrcDstPrevQLsfResidual, Ipp16s*  pSrcDstPrevQLsf, Ipp16s*  pSrcDstPrevQLsp,
    Ipp16s*  pDstQLsp, Ipp16s bfi, IppSpchBitRate mode);
```

Parameters

pSrcQLspIndex	Pointer to the five-element vector containing codebook indices of the quantized LSPs. For 12.2 kbps frames, all five elements contain valid indices; for all other bit rates, only the first three elements contain valid indices.
pSrcDstPrevQLsfResidual	Pointer to the ten-element quantized LSF residual from the previous frame, represented using Q0.15. On output, points to the ten-element updated LSF residual, represented using Q0.15.
pSrcDstPrevQLsf	Pointer to the ten-element quantized LSF vector from the previous frame, represented using Q0.15. On output, points to the ten-element updated quantized LSF vector, represented using Q0.15.
pSrcDstPrevQLsp	Pointer to the ten-element quantized LSP vector from the previous frame, represented using Q0.15. On output, points to the ten-element updated quantized LSP vector, represented using Q0.15.
pDstQLsp	Pointer to a 40-element vector containing four subframe LSP sets. Two sets are generated by interpolation for 12.2 kbps frames; for all other bit rates three sets are generated by interpolation. All elements are represented in using Q0.15.

bfi	Bad frame indicator; “0” signifies a good frame; all other values signify a bad frame.
mode	Bit rate specifier. The enumerated values of `IPP_SPCHBR_4750` to `IPP_SPCHBR_12200` are valid.

Description

The function `ippsQuantLSPDecode_GSMAMR` is declared in `ippsc.h` file. This function decodes quantized LSPs from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized LSPs from previous quantized LSPs using linear interpolation. The functionality can be summarized as follows:

1. If no errors are detected on the current frame, obtain the quantized LSPs from the codebook indices and the previous quantized residual using inverse LSP quantization.
2. If errors are detected on the current frame, quantized LSFs are obtained using the following rate-dependent interpolation scheme:

$$\begin{cases} lsf_q1(i) = lsf_q2(i) = \alpha \times past_lsf_q(i) + (1-\alpha) \times mean_lsf(i), & \text{12.2kbit/s mode} \\ lsf_q(i) = \alpha \times past_lsf_q(i) + (1-\alpha) \times mean_lsf(i), & \text{otherwise} \end{cases}$$

where $i = 0, 1, \ldots, 9$, $\alpha = 0.95$, *lsf_q*1 and *lsf_q*2 (for 12.2 kbps) are two sets of quantized LSF vectors for current frame, *past_lsf_q* is *lsf_q*2 of the previous frame, and *mean_lsf* is the average LSF vector. Note that there is only one set of quantized LSF coefficients for rates other than 12.2 kbps. The corresponding quantized LSPs are obtained by LSF-to-LSP conversion.

3. Linear interpolation is applied to generate four sets of quantized LSPs from the decoded set(s) of LSPs and the quantized LSPs from the previous frame.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcQLspIndex*, *pSrcDstPrevQLsfResidual*, *pSrcDstPrevQLsf*, *pSrcDstPrevQLsp* or *pDstQLsp* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

Adaptive Codebook Functions

This section describes primitives that are concerned with various aspects of the adaptive codebook, including primitives that perform the following functions:

- Open-loop pitch search, including Non-DTX, VAD1, and VAD2
- Impulse response and target signal computation
- Adaptive codebook search
- Decoding of the adaptive codebook vector.

Open-Loop Pitch Search (OLP)

Three functions are provided for OLP search. Use of these functions is mutually exclusive, that is, only one is appropriate for an application during any given frame type. The appropriate choice of an OLP search function depends upon the state of the DTX and VAD modules. The OLP search functions should be applied as follows:

Encoder Mode	Appropriate Function
DTX disabled	`ippsOpenLoopPitchSearchNonDTX_GSMAMR`
DTX, VAD 1 enabled	`ippsOpenLoopPitchSearchDTXVAD1_GSMAMR`
DTX, VAD 2 enabled	`ippsOpenLoopPitchSearchDTXVAD2_GSMAMR`

The OLP search functions extract a pitch estimate from a weighted version of the input speech. For 5.15 and 4.75 kbps frames, the search is performed once per frame. For all other modes, the search is performed twice per frame. A unique search is employed for 10.2 kbps frames. The OLP search details are as follows:

1. If the transmission bit rate is 10.2 kbps,
   a. Compute a windowed autocorrelation of weighted speech, that is,

$$R(k) = w(k) \times \sum_{n=0}^{79} sw(n) \times sw(n-k), \quad 20 \le k \le 143$$

where the sequence $sw(n)$ contains weighted speech, and w is the weighting function given by

$$w(k) = wl(k) \times wn(k)$$

In this weighting function, wl emphasizes low pitch, and is defined in terms of the table cw, while wn is a sequence of neighboring emphasis lag coefficients associated with the previous frame, i.e.

$$wn(k) = \begin{cases} cw(|T_{old} - k| + 20) & weighflag = 1 \\ 1 & \text{otherwise} \end{cases}$$

and the parameter T_{old} is the median filtered pitch lag of 5 previous voiced speech half frames.

The estimated open-loop pitch lag is the value k that maximizes $R(k)$. It is denoted by T_{op}.

b. Compute the optimal open-loop gain using the relation

$$g = \frac{\sum_{n=0}^{79} sw(n) \times sw(n - T_{op})}{\sum_{n=0}^{79} sw(n) \times sw(n)} - 0.4$$

In addition,

$$v = \begin{cases} 1 & g > 0 \\ 0.9v & \text{otherwise} \end{cases}$$

and

$$weighflag = \begin{cases} 1 & v > 0 \\ 0 & \text{otherwise} \end{cases}$$

If $g > 0$, the previous pitch lag buffer and v median pitch lag of the previous pitch lags are updated.

2. For rates other than 10.2 kbps, the following OLP search procedure is employed:

a. Three maxima are found in three different ranges for the correlations given by

$$R(k) = \sum_{n=0}^{length-1} sw(n) \times sw(n-k)$$

For 5.15 and 4.75 kbps frames, $length$ = 160. For all other bit rates (except 10.2 kbps), $length$ = 80.

b. Normalize the three maximum correlations according to

$$M_i = \frac{M_i}{\sqrt{\sum_{n=0}^{length-1} sw^2(n - T_i)}} \qquad i = 1,2,3$$

c. Determine the best open-loop lag using the following rule:

$$T_{op} = T_1 \,,\; M(T_{op}) = M_1$$
$$if(M_2 > 0.85M(T_{op}))$$
$$M(T_{op}) = M_2 \,,\; and\; T_{op} = T_2$$
$$if(M_3 > 0.85M(T_{op}))$$
$$T_{op} = T_3$$

Each of the OLP search functions implements the rate-dependent search algorithms described above. Next, details are given for non-DTX, VAD1, and VAD2 OLP search primitives.

OpenLoopPitchSearchNonDTX_GSMAMR

Computes the open-loop pitch lag.

Syntax

```
IppStatus ippsOpenLoopPitchSearchNonDTX_GSMAMR_16s(const Ipp16s*  pSrcWgtLpc1,
    const Ipp16s*  pSrcWgtLpc2, const Ipp16s*  pSrcSpch, Ipp16s*
    pValResultPrevMidPitchLag, Ipp16s*  pValResultVvalue, Ipp16s*
    pSrcDstPrevPitchLag, Ipp16s*  pSrcDstPrevWgtSpch, Ipp16s*  pDstOpenLoopLag,
    Ipp16s*  pDstOpenLoopGain, IppSpchBitRate mode);
```

Parameters

pSrcWgtLpc1	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. This set of LP coefficients comprises the numerator coefficients for the perceptual weighting filter.

pSrcWgtLpc2	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. This set of LP coefficients comprises the denominator coefficients for the perceptual weighting filter.
pSrcSpch	Pointer to the 170-sample input speech vector, represented using Q15.0.
pValResultPrevMidPitchLag	Pointer to the median filtered pitch lag of the 5 previous voiced speech half-frames, represented using Q15.0. On output, points to the updated median filtered pitch lag vector (5 previous voiced speech half-frames), represented using Q15.0. This argument is valid only for 10.2 kbps frames.
pValResultVvalue	Pointer to the adaptive parameter *v* described in Open-Loop Pitch Search (OLP), represented using Q0.15. Used also as an output argument. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevPitchLag	Pointer to the five-element vector that contains the pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five previous voiced half-frames. This argument is used only for 10.2 kbps frames.
pSrcDstPrevWgtSpch	Pointer to a 143-element vector containing perceptually weighted speech from the previous frame, represented using Q15.0. On output, points to the updated 143-element vector of perceptually weighted speech, represented using Q15.0.
pDstOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
pDstOpenLoopGain	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Description

The function `ippsOpenLoopPitchSearchNonDTX_GSMAMR` is declared in `ippsc.h` file. This function computes the open-loop pitch lag (as well as optimal pitch gain for 10.2 kbps frames only) when both DTX and VAD are disabled. The search algorithm is described in Open-Loop Pitch Search (OLP) .

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

OpenLoopPitchSearchDTXVAD1_GSMAMR

Extracts an open-loop pitch lag estimate (VAD 1 scheme is enabled).

Syntax

```
IppStatus ippsOpenLoopPitchSearchDTXVAD1_GSMAMR_16s(const Ipp16s*  pSrcWgtLpc1,
    const Ipp16s*  pSrcWgtLpc2, const Ipp16s*  pSrcSpch,
    Ipp16s*  pValResultToneFlag, Ipp16s*  pValResultPrevMidPitchLag,
    Ipp16s*  pValResultVvalue, Ipp16s*  pSrcDstPrevPitchLag,
    Ipp16s*  pSrcDstPrevWgtSpch, Ipp16s*  pResultMaxHpCorr, Ipp16s*
    pDstOpenLoopLag, Ipp16s*  pDstOpenLoopGain, IppSpchBitRate mode);
```

Parameters

pSrcWgtLpc1	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the numerator of the perceptual weighting filter.
pSrcWgtLpc2	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the denominator of the perceptual weighting filter.
pSrcSpch	Pointer to the 170-element input speech vector, represented using Q15.0.

pValResultToneFlag	Pointer to the tone flag for the VAD module. On output, points to the updated tone flag for the VAD module.
pValResultPrevMidPitchLag	Pointer to a vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
pValResultVvalue	Pointer to the adaptive parameter *v*, represented using Q0.15. On output, points to the updated adaptive parameter *v*, represented using Q0.15. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevPitchLag	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevWgtSpch	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output, points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.
pResultMaxHpCorr	Pointer to the correlation maximum.
pDstOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
pDstOpenLoopGain	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Description

The function `ippsOpenLoopPitchSearchDTXVAD1_GSMAMR` is declared in `ippsc.h` file. This function extracts an open-loop pitch lag estimate from the weighted input speech when the VAD 1 scheme is enabled using a version of the pitch search algorithm described in Open-Loop Pitch Search (OLP) , that is modified as follows:

1. For 10.2 kbps frames, the following modification is applied after the best open-loop pitch is found:
   a. Update the tone flag (when initialized or reset, it is set to 0) in the following way

$$toneflag >>= 1$$

$$DelayEnergy = \sum_{n=0}^{79} sw^2(n - T_{op})$$

$$MaxCorr = \sum_{n=0}^{79} sw(n - T_{op}) \times sw(n)$$

$$if\,(0.325 \times DelayEnergy < MaxCorr) \quad toneflag = toneflag \mid 0x4000$$

   b. On the second OLP search for the current frame, find the maximum of the high-pass filtered autocorrelations, i.e.,

$$maxhpcorr = \max(R(k) \times 2 - R(k-1) - R(k+1) \mid k = 142,\ldots,24)$$

Then, *maxhpcorr* is normalized by *NormFactor* = *frameEnergy* - *frameCorr*:

$$frameEnergy = \sum_{n=0}^{79} sw^2(n) \quad , \qquad frameCorr = \sum_{n=0}^{79} sw(n) \times sw(n-1)$$

2. For all other bit rates, the following modifications are applied:
   a. Before the open-loop pitch search, update the tone flag as follows:

   toneflag = *toneflag* >>1

   If the bit rate is either 5.15 or 4.75 kbps, update the tone flag as follows:

   toneflag = *toneflag* >> 1, *toneflag* = *toneflag* | 0x2000

   b. After finding three open-loop pitch candidates, update the tone flag as follows:
   if (*DelayEnergy* × 0.65 < *MaxCorr*) *toneflag* = *toneflag* | 0x4000

This update is repeated three times with *DelayEnergy* and *MaxCorr* corresponding to the three pitch candidates. Note that the computation length of *DelayEnergy* and *MaxCorr* for 4.75 and 5.15 kbps frames is 160 samples. For all other bit rates, the length is 80 samples.

c. On the second OLP search for each frame, find the maximum of the high passed autocorrelations. The implementation is identical the 10.2 kbps correlation search, but the search range is rate-dependent.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

OpenLoopPitchSearchDTXVAD2_GSMAMR

Extracts an open-loop pitch lag estimate (VAD 2 scheme is enabled).

Syntax

```
IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR(const Ipp16s*  pSrcWgtLpc1,
    const Ipp16s*  pSrcWgtLpc2, const Ipp16s*  pSrcSpch,
    Ipp16s*  pValResultPrevMidPitchLag, Ipp16s*  pValResultVvalue,
    Ipp16s*  pSrcDstPrevPitchLag, Ipp16s*  pSrcDstPrevWgtSpch,
    Ipp32s*  pResultMaxCorr, Ipp32s pResultWgtEnergy, Ipp16s*  pDstOpenLoopLag,
    Ipp16s*  pDstOpenLoopGain, IppSpchBitRate mode);
IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR_16s32s(const Ipp16s*
    pSrcWgtLpc1, const Ipp16s*  pSrcWgtLpc2, const Ipp16s*  pSrcSpch, Ipp16s*
    pValResultPrevMidPitchLag, Ipp16s*  pValResultVvalue, Ipp16s*
    pSrcDstPrevPitchLag, Ipp16s*  pSrcDstPrevWgtSpch, Ipp32s*  pResultMaxCorr,
    Ipp32s pResultWgtEnergy, Ipp16s*  pDstOpenLoopLag, Ipp16s*
    pDstOpenLoopGain, IppSpchBitRate mode);
```

Parameters

pSrcWgtLpc1	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the numerator of the perceptual weighting filter.

pSrcWgtLpc2	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the denominator of the perceptual weighting filter.
pSrcSpch	Pointer to the 170-element input speech vector, represented using Q15.0.
pValResultToneFlag	Pointer to the tone flag for the VAD module.
pValResultPrevMidPitchLag	Pointer to a vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
pValResultVvalue	Pointer to the adaptive parameter *v*, represented using Q0.15. On output, points to the updated adaptive parameter *v*, represented using Q0.15. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevPitchLag	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevWgtSpch	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output, points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.
pResultMaxCorr	Pointer to the correlation maximum.
pResultWgtEnergy	Pointer to the pitch delayed energy of the weighted speech signal, as described above. The output may be scaled, and the Q representation is not given.
pDstOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.

pDstOpenLoopGain	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Description

The function `ippsOpenLoopPitchSearchDTXVAD2_GSMAMR` is declared in `ippsc.h` file. This function extracts an open-loop pitch lag estimate from the weighted input speech when the VAD 2 scheme is enabled, using a version of the pitch search algorithm described in Open-Loop Pitch Search (OLP) that is modified in the following way:

After finding the best open-loop pitch, extract from the weighted speech the maximum correlation *MaxCorr* and the delayed energy *DelayEnergy*. For both 4.75 and 5.15 kbps frames, the computation is carried for 160 samples. For all other bit rates, the computation is carried for only 80 samples using the relations

$$MaxCorr = \sum_{n=0}^{length-1} sw(n) \times sw(n - T_{op})$$

$$DelayEnergy = \sum_{n=0}^{length-1} sw^2(n - T_{op})$$

The *MaxCorr* and *DelayEnergy* values corresponding to the two half-frame open-loop pitch lag estimates are combined to obtain whole-frame estimates of *MaxCorr* and *DelayEnergy* (except during 4.75 and 5.15 kbps frames, when the OLP search is performed only once per frame).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

ImpulseResponseTarget_GSMAMR

Computes the impulse response and target signal required for the adaptive codebook search.

Syntax

```
IppStatus ippsImpulseResponseTarget_GSMAMR_16s(const Ipp16s* pSrcSpch, const
    Ipp16s* pSrcWgtLpc1, const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcQLpc,
    const Ipp16s* pSrcSynFltState, const Ipp16s* pSrcWgtFltState, Ipp16s*
    pDstImpulseResponse, Ipp16s* pDstLpResidual, Ipp16s* pDstAdptTarget);
```

Parameters

pSrcSpch	Pointer to the 50-element input speech vector, where elements 0 - 9 are from the previous subframe, and elements 10 - 49 are from the current subframe.
pSrcWgtLpc1	Pointer to an eleven-element vector of weighted LP coefficients associated with $A(z/\gamma_1)$ on the current subframe, represented using Q3.12.
pSrcWgtLpc2	Pointer to an eleven-element vector of weighted LP coefficients associated with $A(z/\gamma_2)$ on the current subframe, represented using Q3.12.
pSrcQLpc	Pointer to an eleven-element vector of quantized LP coefficients for the current subframe, represented using Q3.12.
pSrcSynFltState	Pointer to the ten-element vector that contains the state of the synthesis filter, represented using Q15.0.
pSrcWgtFltState	Pointer to the ten-element vector that contains the state of the weighting filter, represented using Q15.0.
pDstImpulseResponse	Pointer to the 40-element vector that contains the impulse response, represented using Q3.12.
pDstLpResidual	Pointer to the 40-element vector that contains the LP residual, represented using Q15.0.
pDstAdptTarget	Pointer to the 40-element vector that contains the adaptive codebook search target signal, represented using Q15.0.

Description

The function `ippsImpulseResponseTarget_GSMAMR` is declared in `ippsc.h` file. This function computes the impulse response and target signal required for the adaptive codebook search. This function is performed on a subframe basis using the following approach:

1. The impulse response $h(n)$ of the weighted synthesis filter,

$$H(z)W(z) = A(z/\gamma_1)/[\hat{A}(z)A(z/\gamma_2)] \quad ,$$

   is computed by applying the filters $1/\hat{A}(z)$ and $1/A(z/\gamma_2)$ to the zero-padded impulse response of the filter $A(z/\gamma_1)$.

2. The target signal is then obtained by applying to the LP residual $res_{LP}(n)$ the cascaded synthesis and weighting filters, $1/A(z)$, and $A(z/\gamma_1)/A(z/\gamma_2)$, respectively. The adaptive codebook search also uses the residual signal $res_{LP}(n)$ to update the history of past excitations. The LP residual is obtained by inverse filtering the input speech, i.e.,

$$res_{LP}(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i s(n-i)$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcSpch*, *pSrcWgtLpc1*, *pSrcWgtLpc2*, *pSrcQLpc*, *pSrcSynFltState*, *pSrcWgtFltState*, *pDstImpulseResponse*, *pDstLpResidual*, or *pDstAdptTarget* pointer is `NULL`.

AdaptiveCodebookSearch_GSMAMR

Performs the adaptive codebook search.

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_GSMAMR_16s(const Ipp16s*  pSrcTarget,
    const Ipp16s* pSrcImpulseResponse, Ipp16s*  pSrcOpenLoopLag, Ipp16s*
    pValResultPrevIntPitchLag, Ipp16s*  pSrcDstExcitation, Ipp16s*
    pResultFracPitchLag, Ipp16s*  pResultAdptIndex, Ipp16s*  pDstAdptVector,
    Ipp16s subFrame, IppSpchBitRate mode);
```

Parameters

pSrcTarget	Pointer to the 40-element adaptive target signal vector, represented using Q15.0. This should be aligned on an 8-byte boundary.
pSrcImpulseResponse	Pointer to the 40-element impulse response of the weighted synthesis filter, represented using Q3.12. This should be aligned on an 8-byte boundary.
pSrcOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
pValResultPrevIntPitchLag	Pointer to the previous integral pitch lag.
pSrcDstExcitation	Pointer to the 194-element excitation vector. Elements 0 ~ 153 contain samples of the past excitation, represented using Q15.0. Elements 154 ~ 193 contain samples of the prediction residual, however, the prediction residuals are used only when the subframe length exceeds the integer closed-loop pitch estimate.
pValResultPrevIntPitchLag	Pointer to the current integer pitch lag.
pSrcDstExcitation	Pointer to the updated 194-element excitation vector. Elements 0 ~ 153 contain past excitations, and are represented using Q15.0. Elements 154 ~ 193 contain the 40-sample adaptive codebook vector, *v*.
pResultFracPitchLag	Pointer to the fractional pitch lag obtained during the adaptive codebook search.
pResultAdptIndex	Pointer to the coded closed-loop pitch index.
pDstAdptVector	Pointer to the 40-sample adaptive codebook vector, represented using Q15.0.
subFrame	Subframe index.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Description

The function `ippsAdaptiveCodebookSearch_GSMAMR` is declared in `ippsc.h` file. This function performs the adaptive codebook search. The adaptive codebook search consists of a closed-loop pitch search followed by computation of an adaptive excitation vector. The adaptive excitation vector is obtained by interpolating the past excitation at the fractional

pitch lag obtained during the closed-loop pitch search. The adaptive codebook is searched on every subframe. A detailed description of the adaptive codebook search procedure is given next:

1. A closed-loop pitch analysis is performed in the neighborhood of the open-loop pitch estimate, T_{op}, on a subframe basis. In the first and third subframes (only the first subframe for 5.15 and 4.75 kbps modes), the search neighborhood is rate-dependent. For 12.2 kbps frames, the search range is T_{op} ±3, and is bounded by 18…143. For 5.15 or 4.75 kbps frames, the search range is T_{op} ±5, and is bounded by 20…143. For all other bit rates, the search range is T_{op} ±3, and is bounded by 20…143. For the second and fourth subframes, the search neighborhood surrounds T_1, the nearest integer to the fractional pitch lag of the previous subframe, and the neighborhood boundaries are also rate-dependent. For 12.2 kbps frames, the search range is [T_1 – 5 3/6, T_1 + 4 3/6]. For 7.95 kbps frames, the search range is [T_1 – 10 2/3, T_1 + 9 2/3]. For 10.2 or 7.40 kbps frames, the search range is [T_1 – 5 2/3, T_1 + 4 2/3]. For all other bit rates, the search range is [T_1 – 5, T_1 + 4].

2. The optimum integer pitch search minimizes the mean-square weighted error between the original and synthesized speech. This is achieved by maximizing the normalized cross-correlation given by

$$R(k) = \frac{\sum_{n=0}^{39} x(n) y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n) y_k(n)}}$$

   where $x(n)$ is the target signal, and $y_k(n)$ is the past filtered excitation at delay k (past excitation convolved with the impulse response $h(n)$). The convolution $y_k(n)$ is computed for the first delay $t_{\min}$ in the searched range. For other delays in the range $k = t_{\min} + 1, \ldots, t_{\max}$, it is updated using the recursive relation

$$y_k(n) = y_{k-1}(n-1) + u(-k)h(n)$$

   where $u(n)$, n = -154,…,39 , is the excitation history buffer. To simplify the search, the prediction residual is copied to $u(n)$, n = 0,…39, making it valid for all delays.

3. The fractional pitch search is performed by interpolating $R(k)$ and searching for its maximum. Fractional delay interpolation is achieved using an FIR filter, b_{24}, based on a hamming windowed *sinc* function truncated at ±23 and padded with zeros at ±24, i.e.,

$$R_t(k) = \sum_{i=0}^{3} R(k-i) b_{24}(t+6i) + \sum_{i=0}^{3} R(k+1+i) b_{24}(6-t+6i)$$

   where t corresponds to the fractional delay. If the bit rate is 12.2 kbps, then the delay is –1/2 to 1/2 with a resolution of 1/6; otherwise the fraction is –2/3 to 2/3 with a resolution of 1/3.

4. The adaptive codebook vector $v(n)$ is computed by interpolating the past excitation signal $u(n)$ at the integer delay k and fractional delay t, i.e.,

$$v(n) = \sum_{i=0}^{9} u(n-k-i)b_{60}(t+6i) + \sum_{i=0}^{9} u(n-k+1+i)b_{60}(6-t+6i)$$

The interpolation filter is based on a Hamming windowed *sinc* function truncated at ±59 and padded with zeros at ±60.

5. For the first and the third subframes (only the first subframe during 4.75 and 5.15 kbps modes), the pitch lag bit allocation is rate-dependent. For 12.2 kbps frames, the pitch lag is encoded with 9 bits. For all other bit rates, the pitch lag is encoded with 8 bits. For the second and fourth subframes, pitch is encoded with 6 bits for the 12.2 and 7.95 kbps modes; 5 bits for 10.2 or 7.4 kbps modes, and 4 bits for all other modes.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcTarget*, *pSrcImpulseResponse*, *pSrcOpenLoopLag*, *pValResultPrevIntPitchLag*, *pSrcDstExcitation*, *pResultFracPitchLag*, *pResultAdptIndex* or *pDstAdptVector* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.
`ippStsSizeErr`	Indicates an error when *subFrame* is not in the range [0, 3].

AdaptiveCodebookDecode_GSMAMR

Decodes the adaptive codebook parameters.

Syntax

```
IppStatus ippsAdaptiveCodebookDecode_GSMAMR_16s(Ipp16s valAdptIndex, Ipp16s*
    pValResultPrevIntPitchLag, Ipp16s*  pValResultLtpLag, Ipp16s*
    pSrcDstExcitation, Ipp16s*  pResultIntPitchLag, Ipp16s*  pDstAdptVector,
    Ipp16s subFrame, Ipp16s bfi, Ipp16s inBackgroundNoise, Ipp16s
    voicedHangover, IppSpchBitRate mode);
```

Parameters

valAdptIndex	Adaptive codebook index.
pValResultPrevIntPitchLag	Pointer to the previous integer pitch lag. Used as an output argument also.

pValResultLtpLag	Pointer to the LTP-Lag value. Used as an output argument also.
pSrcDstExcitation	Pointer to the 194-element excitation vector. Elements 0 ~ 153 contain the past excitation, represented using Q15.0. Elements 154 ~ 193 are used as a buffer whenever the subframe length exceeds the pitch lag. On output, elements 154 - 193 are updated to contain the adaptive codebook vector.
pResultIntPitchLag	Pointer to the integer pitch.
pDstAdptVector	Pointer to the 40-sample adaptive codebook vector, represented using Q15.0.
subFrame	Subframe index.
bfi	Bad frame indicator. “0” signifies a good frame; any other value signifies a bad frame.
inBackgroundNoise	Flag set when the previous frame is considered to contain background noise and only shows minor energy level changes.
voicedHangover	Counter used to monitor the time since a frame was presumably voiced.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Description

The function `ippsAdaptiveCodebookDecode_GSMAMR` is declared in `ippsc.h` file. This function decodes the adaptive codebook parameters transmitted by the encoder, and then applies them to interpolate an adaptive codebook vector. If errors are detected on the received frame, previously received parameters are used to approximate the parameters of the current frame and the adaptive codebook vector interpolation procedure is carried with the approximated parameter set. Adaptive codebook vectors are decoded for every subframe. Details of this primitive are given next:

1. If no errors are detected on the current frame, integer and fractional pitch lags are extracted from the adaptive codebook indices.
2. If errors are detected, the integer pitch is recovered either from the previous integer pitch or the *LTP-Lag*, and the fractional pitch is set to zero. The *LTP-Lag* value is replaced by the integer pitch of the 4th subframe of the previous frame (12.2 Kbps mode) or slightly modified values based on the last correctly received value (all other modes).
3. The same adaptive codebook interpolation procedure described in section 13.4.3 is applied to obtain the adaptive codebook vector.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pValResultPrevIntPitchLag*, *pValResultLtpLag*, *pSrcDstExcitation*, *pResultIntPitchLag*, or *pDstAdptVector* pointer is NULL.
ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.
ippStsSizeErr	Indicates an error when *subFrame* is not in the range [0, 3].

AdaptiveCodebookGain_GSMAMR

Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.

Syntax

```
IppStatus ippsAdaptiveCodebookGain_GSMAMR_16s (const Ipp16s*  pSrcAdptTarget,
    const Ipp16s* pSrcFltAdptVector, Ipp16s* pResultAdptGain);
IppStatus ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s (const Ipp16s*
    pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, Ipp16s* pResultAdptGain,
    Ipp16s* pResultAdptGainCoeffs);
```

Parameters

pSrcAdptTarget	Pointer to the adaptive target signal vector [40].
pSrcFltAdptVector	Pointer to the filtered adaptive-codebook vector [40], in Q12.
pResultAdptGain	Pointer to the output adaptive-codebook gain g_p, in Q14.
pResultAdptGainCoeffs	Pointer to the output gain coefficients vector [4] , in Q15.

Description

The functions ippsAdaptiveCodebookGain_GSMAMR and ippsAdaptiveCodebookGainCoeffs_GSMAMR are declared in ippsc.h file.

ippsAdaptiveCodebookGain_GSMAMR_16s. This function calculates the adaptive-codebook gain g_p as given by:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)}$$

bounded by $0 \le g_p \le 1.2$, in Q14,

where x is the adaptive target signal vector, and y is the filtered adaptive-codebook vector.

See also the `ippsAdaptiveCodebookGain_G729` function.

ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s. This function calculates the adaptive-codebook gain g_p in the same way as the `ippsAdaptiveCodebookGain_GSMAMR_16s` function does, and additionally returns the gain in a different representation given by the following formula:

$$g_p = \frac{c_{xy} 2^{\exp_{xy}}}{c_{yy} 2^{\exp_{yy}}}, 1/2 \le |c_{xy}|, |c_{yy}| < 1, Q15$$

The mantissas c_{xy} , c_{yy} and exponents exp_{xy} , exp_{yy} for both the normalized denominator and normalized numerator are returned in the *pResultAdptGainCoeffs* vector:

pResultAdptGainCoeffs[0] = c_{yy},

pResultAdptGainCoeffs[1] = exp_{yy},

pResultAdptGainCoeffs[2] = c_{xy},

pResultAdptGainCoeffs[3] = exp_{xy}

If $c_{xy} < 4$, then zero gain is returned.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcAdptTarget*, *pSrcFltAdptVector*, *pResultAdptGain* or *pResultAdptGainCoeffs* pointer is `NULL`.

Fixed Codebook Search

This section describes primitives that are concerned with the fixed codebook, including primitives that perform the following functions:

- Fixed (algebraic) codebook search
- Fixed codebook vector decode

AlgebraicCodebookSearch_GSMAMR

Searches the algebraic codebook.

Syntax

```
IppStatus ippsAlgebraicCodebookSearch_GSMAMR_16s(Ipp16s valIntPitchLag, Ipp16s
    valBoundQAdptGain, const Ipp16s* pSrcFixedTarget, const Ipp16s*
    pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse, Ipp16s* pDstFixedVector,
    Ipp16s* pDstFltFixedVector, Ipp16s* pDstEncPosSign, Ipp16s subFrame,
    IppSpchBitRate mode);
IppStatus ippsAlgebraicCodebookSearchEX_GSMAMR_16s(Ipp16s valIntPitchLag,
    Ipp16s valBoundQAdptGain, const Ipp16s* pSrcFixedTarget, const Ipp16s*
    pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse, Ipp16s* pDstFixedVector,
    Ipp16s* pDstFltFixedVector, Ipp16s* pDstEncPosSign, Ipp16s subFrame,
    IppSpchBitRate mode, Ipp32s* pBuffer);
```

Parameters

valIntPitchLag	The nearest integer pitch lag T to the closed-loop fractional pitch lag of this subframe, which is computed by closed-loop pitch search routine.
valBoundQAdptGain	Bounded quantized adaptive codebook gain, which is denoted by the parameter β of the filter $F_E(z)$ in the ETSI GSM 06.90 standards document. For MR122 mode, this value is the bounded quantized pitch gain of current subframe. While for other modes, it is the bounded quantized pitch gain of previous subframe. This value is represented using Q1.14 format.
pSrcFixedTarget	Pointer to the 40-element fixed target signal vector $x_2(n)$, which is used to search the fixed codebook vector, represented using Q15.0. This should be aligned on an 8-byte boundary.

pSrcLtpResidual	Pointer to the 40-element long-term prediction residual signal vector $res_{LTP}(n)$, represented using Q15.0.
pSrcDstImpulseResponse	Pointer to the 40-element weighted synthesis filter impulse response vector, represented using Q3.12. This should be aligned on an 8-byte boundary. On output, points to the updated 40-element impulse response vector, which is obtained by filtering original impulse response $h(n)$ through the pre-filter $F_E(z)$. It is represented using Q3.12.
pDstFixedVector	Pointer to the 40-element fixed codebook vector $c(n)$, represented using Q2.13.
pDstFltFixedVector	Pointer to the 40-element filtered fixed codebook vector $z(n)$, which is obtained by convolving the impulse response with the fixed codebook vector, represented using Q2.13.
pDstEncPosSign	Pointer to the ten-element buffer that contains the encoded positions and signs of optimal pulses. For 12.2 kbps mode, 10 short words are used to store the result of this encoding. For the 10.2 kbps mode, 7 short words are used. For all other modes, only 2 short words are used.
subFrame	Subframe index, which ranges from 0 to 3.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.
pBuffer	Pointer to internal working buffer, of length 1K.

Description

The functions `ippsAlgebraicCodebookSearch_GSMAMR` and `ippsAlgebraicCodebookSearchEX_GSMAMR` are declared in `ippsc.h` file. These functions search the algebraic codebook by minimizing the mean square error between the weighted input speech and the weighted synthesized speech. After the fixed codebook vector has been obtained, it is filtered through the weighted synthesis filter to obtain a fixed codebook vector. The positions and signs of the optimal pulses are encoded respectively according to the GSM06.90 specification. Algebraic codebook search is applied on each subframe.

These two functions work identically with the following exception: `ippsAlgebraicCodebookSearchEX_GSMAMR` uses an internal working buffer pointed by *pBuffer* allocated by user, but `ippsAlgebraicCodebookSearch_GSMAMR` allocates this internal working buffer in stack.

Details of the algebraic search are given next.

1. Prior to fixed codebook search, if the integer part of the closed-loop pitch is less than the subframe length, filter the impulse response $h(n)$ through the pre-filter $F_E(z)$, i.e.,
$$h(n) = h(n) - \beta h(n-T) \quad , \quad n = T,\ldots, 39$$
where β is the bounded and quantized adaptive gain, and T is the integer part of the closed pitch.

2. Compute backward filtered target vector **d** by
$$d(n) = \sum_{i=n}^{39} x_2(i) \times h(i-n) \quad , \quad 0 \le n \le 39$$
where $x_2(n)$ is the fixed target signal used for fixed codebook search.

3. For 12.2 Kbps and 10.2 Kbps modes, the signal $b(n)$ which is used for presetting the pulse amplitudes is computed:
$$b(n) = \frac{res_{LTP}(n)}{\sqrt{\sum_{i=0}^{39} res_{LTP}(i) res_{LTP}(i)}} + \frac{d(n)}{\sqrt{\sum_{i=0}^{39} d(i) d(i)}} \quad , \quad n = 0,\ldots, 39$$
For other modes, $b(n)$ is equal to the signal $d(n)$.

4. Calculate the symmetric Toepliz matrix Φ for each mode, the element of it is computed by
$$\Phi(i, j) = \sum_{n=j}^{39} h(n-i) \times h(n-j) \ , \ i \le j \quad , \quad 0 \le i \le 39$$

5. Search the optimal pulse positions for each mode. An efficient non-exhaustive analysis-by-synthesis search technique is used for 12.2Kbps, 10.2Kbps, 7.95Kbps, 7.4Kbps, 6.70 Kbps modes. An exhaustive analysis-by-synthesis search technique is used for other modes. GSM 06.90 (clause 5.7) contains complete details.

6. Construct the fixed codebook vector $c(n)$ according to optimal pulse positions, then convolve it with $h(n)$ to get filtered fixed codebook vector $z(n)$
$$z(n) = \sum_{i=0}^{39} c(i) \times h(n-i) \quad , \quad 0 \le n \le 39$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.

ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.

FixedCodebookDecode_GSMAMR

Decodes the fixed codebook vector.

Syntax

```
IppStatus ippsFixedCodebookDecode_GSMAMR_16s(const Ipp16s*  pSrcFixedIndex,
    Ipp16s*  pDstFixedVector, Ipp16s subFrame, IppSpchBitRate mode);
```

Parameters

pSrcFixedIndex	Pointer to the fixed codebook index vector. If the mode is 12.2 kbps, the vector length is 10; if the mode is 10.2 kbps, the vector length is 7; otherwise the vector length is 2.
pDstFixedVector	Pointer to the 40-element fixed codebook vector.
subFrame	Subframe index.
mode	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Description

The function ippsFixedCodebookDecode_GSMAMR is declared in ippsc.h file. This function decodes the fixed codebook vector from the received fixed codebook index.

Return Values

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error when the *pSrcFixedIndex* or *pDstFixedVector* pointer is NULL.
ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.
ippStsSizeErr	Indicates an error when *subFrame* is not in the range [0, 3].

Discontinuous Transmission (DTX)

This section describes primitives that are concerned with discontinuous transmission (DTX), including primitives that perform the following functions:

- Signal pre-emphasis prior to VAD option 2
- VAD Decision Function for Option 1
- VAD Decision Function for Option 2
- Parameter Extraction for the SID frame
- DTX Handler
- DTX Buffering

Each of these primitives is described next.

Preemphasize_GSMAMR

Computes pre-emphasis of an input signal in VAD option 2.

Syntax

```
IppStatus ippsPreemphasize_GSMAMR_16s (Ipp16s gamma, const Ipp16s* pSrc,
    Ipp16s* pDst, int len, Ipp16s* pMem);
```

Parameters

gamma	The filter coefficient, in Q15.
pSrc	Pointer to the source vector, in Q0.
pDst	Pointer to the destination vector, in Q0.
len	Number of elements in the source and destination vectors.
pMem	Pointer to the filter memory [1].

Description

The function `ippsPreemphasize_GSMAMR` is declared in the `ippsc.h` file. This function computes pre-emphasis of the input signal prior to frequency domain conversion in Voice Activity Detector option 2. The function `ippsPreemphasize_GSMAMR` performs the same operation as the `ippsPreemphasize_G729A` function does, but has a slightly different order to deliver accuracy required by the GSM-AMR transcoding standard.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pMem* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

VAD1_GSMAMR

Implements the VAD functionality corresponding to VAD option 1.

Syntax

```
IppStatus ippsVAD1_GSMAMR_16s(const Ipp16s pSrcSpch, IppGSMAMRVad1State*
    pValResultVad1State, Ipp16s* pResultVadFlag, Ipp16s maxHpCorr, Ipp16s toneFlag);
```

Parameters

pSrcSpch	Pointer to the input speech signal, of length 160, in Q0.
pValResultVad1State	On input, pointer to the VAD Option 1 history variables. On output, points to the updated VAD Option 1 history variables. The structure IppGSMAMRVad1State is defined below.
pResultVadFlag	Pointer to the VAD flag of this frame. If it is set to “1”, it indicates the presence of signals that should be transmitted. If set to “0”, there is no signals in this frame needed to be transmitted.
maxHpCorr	*best_corr_hp* value of previous frame, which is the maximum normalized value of the high pass filtered correlation. This value is the output of the open-loop pitch search function.
toneFlag	Tone flag, which indicates the presence of information tones or signals containing very strong periodic component. This value is the output of the open-loop pitch search function.

Description

The function ippsVAD1_GSMAMR is declared in ippsc.h file. This function implements the VAD functionality corresponding to VAD option 1 of *ETSI GSM 06.94*. It is used to indicate whether each 20ms frame contains signals that should be transmitted - for example,

speech, music or information tones. The structure `IppGSMAMRVad1State` contains the history variables of VAD Option 1. These variables are initialized before the beginning of the encoder, and can be only updated by this f unction. Refer to *ETSI GSM 06.94* VAD Option 1 specification for details of the implementation.

Structure definition of `IppGSMAMRVad1State`:

`typedef struct{`	**Description**
`Ipp16s pPrevSignalLevel[9];`	Signal level vector of *level*[*n*] previous frame.
`Ipp16s pPrevSignalSublevel[9];`	Intermediate signal sublevel vector of previous frame.
`Ipp16s pPrevAverageLevel[9];`	Average signal level vector *ave_level*[*n*] of previous frame.
`Ipp16s pBkgNoiseEstimate[9];`	Background noise estimate vector *back_est*[*n*] of previous frame.
`Ipp16s pFifthFltState[6];`	The history state of the three 5th order filters of filter bank.
`Ipp16s pThirdFltState[5];`	The history state of the five 3rd order filters of filter bank.
`Ipp16s burstCount;`	Burst counter *burst_count* which counts length of a speech burst, used by VAD hangover addition.
`Ipp16s hangCount;`	Hang counter *hang_counter* which is used by VAD hangover addition.
`Ipp16s statCount;`	Stationary counter variable *stat_count* which is used in background noise estimation.
`Ipp16s vadReg;`	Value that indicates intermediate VAD decision.
`Ipp16s complexHigh;`	*complex_high* value which is used as intermediate complex signal decision.
`Ipp16s complexLow;`	*complex_low* value which is used as intermediate complex signal decision.
`Ipp16s complexHangTimer;`	*complex_hang_timer* which is used as hangover initiator by Complex Activity Estimation.
`Ipp16s complexHangCount;`	*complex_hang_count* which is used as hangover counter by VAD hangover addition.
`Ipp16s complexWarning;`	*complex_warning* flag.
`Ipp16s corrHp;`	The high-pass filtered value of *best_corr_hp.*
`Ipp16s pitchFlag;`	Pitch flag which indicates the presence of vowel sounds and other periodic signals.

`}IppGSMAMRVad1State.`

Note. **VAD option 1 history variables initialization**:
Whenever the Encoder is reset, all elements in `pPrevSignalLevel, pPrevSignalSublevel, pPrevAverageLevel` vector should be set as 150, `corrHp` should be set as 13106. While all other variables should be initialized as 0.
For the detail usage of these history variables, please refer to *ETSI GSM 06.94*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the input or output pointers is NULL.

VAD2_GSMAMR

Implements the VAD functionality corresponding to VAD option 2.

Syntax

```
IppStatus ippsVAD2_GSMAMR_16s(const Ipp16s*  pSrcSpch, IppGSMAMRVad2State*
    pValResultVad2State, Ipp16s*  pResultVadFlag, Ipp16s ltpFlag);
```

Parameters

pSrcSpch	Pointer to the input speech frame, of length 160, in Q0.
pValResultVad2State	On input, pointer to the VAD Option 2 history variables. On output, points to the updated VAD Option 2 history variables. The structure IppGSMAMRVad2State is defined below.
pResultVadFlag	Pointer to the Boolean flag *VAD_flag*. If it is set to “1”, it indicates the presence of signals that should be transmitted. If set to “0”, there is no signals in this frame needed to be transmitted.
ltpFlag	*LTP_flag* value in *GSM 06.94* equation (4.24), which is generated by the comparison of the long-term prediction to a constant threshold *LTP_THLD*.

Description

The function ippsVAD2_GSMAMR is declared in ippsc.h file. This function implements the VAD functionality corresponding to VAD option 2 of *ETSI GSM 06.94*. It is used to indicate whether each 20ms frame contains signals that should be transmitted. For example, speech, music, or information tones. The structure IppGSMAMRVad2State contains the history variables of VAD Option 2.

Please refer to *ETSI GSM 06.94* VAD Option 2 specification for details.

Structure definition of `IppGSMAMRVad2State`:

`typedef struct{`	**Description**
`Ipp32s pEngyEstimate[16];`	Channel energy estimates vector $\boldsymbol{E}_{ch}$ of current half-frame, which is calculated during the previous half-frame according to the equation *ETSI GSM 06.94 (4.4)*.
`Ipp32s pNoiseEstimate[16];`	Channel noise estimate vector $\boldsymbol{E}_{n}$ of current half-frame, which is calculated during the previous half-frame according to the equation *ETSI GSM 06.94 (4.26)* .
`Ipp16s pLongTermEngyDb[16];`	Channel average long-term spectral estimate vector $\overline{E}_{dB}$, which is calculated during the previous half-frame according to the equation *ETSI GSM 06.94 (4.20).*
`Ipp16s preEmphasisFactor;`	Pre-emphasis factor $\zeta_{p,}$ which is used to pre-emphasize the input speech signal according to the equation *ETSI GSM 06.94 (4.1).*
`Ipp16s updateCount;`	*update_cnt* value used in background noise update decision logic.
`Ipp16s lastUpdateCount;`	*last_update_cnt* value used in background noise update decision logic.
`Ipp16s hysterCount;`	*hyster_cnt* value used in background noise update decision logic.
`Ipp16s prevNormShift;`	Shifted bits of previous half-frame input speech when normalized to obtain high precision and avoid overflow when doing FFT transformation.
`Ipp16s shiftState;`	Shift state flag which indicates whether previous half-frame has been shifted or not.
`Ipp16s forcedUpdateFlag;`	*fupdate_flag* value which is the result of forced update logic of background noise update decision.
`Ipp16s ltpSnr;`	Long-term peak signal-noise ratio SNR_p of previous half-frame, which is used to calibrate the responsiveness of VAD decision.
`Ipp16s variabFactor;`	Variability factor ψ of previous half-frame, which indicates the variability of the background noise estimate and is updated according to the equation *ETSI GSM 06.94 (4.13).*
`Ipp16s negSnrBias;`	Negative SNR sensitivity Bias factor μ of previous half-frame.
`Ipp16s burstCount;`	Burst counter *b(m)* used in the 10 ms half-frames's VAD Decision.

Structure definition of `IppGSMAMRVad2State`:

`typedef struct{`	**Description**
`Ipp16s hangOverCount;`	Hangover counter *h*(*m*) used in the 10ms half-frame's VAD Decision.
`Ipp32s frameCount;`	Half-frame counter.
`}IppGmrVad2State;`	

Note. All elements in the structure should be initialized to zero whenever encoder is reset.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.

EncDTXSID_GSMAMR

Extracts parameters for the SID frame.

Syntax

```
IppStatus ippsEncDTXSID_GSMAMR_16s(const Ipp16s*  pSrcLspBuffer, const Ipp16s*
    pSrcLogEnergyBuffer, Ipp16s*  pValResultLogEnergyIndex,
    Ipp16s*  pValResultDtxLsfRefIndex, Ipp16s*  pSrcDstQLsfIndex,
    Ipp16s*  pSrcDstPredQErr, Ipp16s*  pSrcDstPredQErrMR122, Ipp16s sidFlag);
```

Parameters

pSrcLspBuffer	Pointer to the LSP coefficients of eight consecutive frames marked with VAD = 0, in the length of 80, in Q0.15.
pSrcLogEnergyBuffer	Pointer to the log energy coefficients of eight consecutive frames marked with unvoiced, in the length of 8, in Q5.10.
pValResultLogEnergyIndex	Pointer to log energy index of last frame, in Q2.13. On output, points to the log energy index of current frame, in Q2.13.
pValResultDtxLsfRefIndex	Pointer to the LSF quantization reference index of last frame. On output, points to the LSF quantization reference index of current DTX frame.

pSrcDstQLsfIndex	Pointer to the LSF residual quantization indices of last frame, in the length of 3. On output, points to the LSF residual quantization indices of current frame, in the length of 3.
pSrcDstPredQErr	Pointer to the fixed gain prediction error of four previous subframes for non-12.2 Kbps modes, in the length of 4, in Q5.10. On output, points to the updated fixed gain prediction error for non 12.2 Kbps modes, in the length of 4, in Q5.10.
pSrcDstPredQErrMR122	Pointer to the fixed gain prediction error of four previous subframes for 12.2 Kbps, in the length of 4, in Q5.10. On output, points to the updated fixed gain prediction error for 12.2 Kbps mode, in the length of 4, in Q5.10.
sidFLag	The SID flag of the current frame. If it is set to 1, the current frame is a SID frame, and the function will extract the LSF and energy parameters. If it is set to 0, the LSF and energy parameters will copy from previous frame.

Description

The function `ippsEncDTXSID_GSMAMR` is declared in `ippsc.h` file. This function is called only when the current frame is a DTX frame. It extracts the needed parameters for the SID frame (that is, the LSF quantization parameter and the energy index parameter). If the SID flag is off, no operation is needed, and all the parameters are copied from last frame. If the SID flag is on, the functionality is as below:

1. Compute the log energy index

   $$EnLogIndex = \frac{1}{8}\sum_{n=0}^{7} EnLog(t-n) \quad ,$$

   where *EnLog*(*t*) is the log energy of the current frame in log2 scale.

2. Update the fixed gain prediction error for 12.2 Kbps mode and other modes respectively with the log energy index:

   $$PredErr(i) = EnLogIndex$$
   $$PredErrMR122(i) = (EnLogIndex)/\ (20 \times \log 10(2))$$
   i = 0,..., 3

3. Compute the average LSP coefficients of the current frame and past seven frames:

   $$Lsp_{mean}(i) = \frac{1}{8}\sum_{n=0}^{7} Lsp(i-n)$$

4. Quantize the average LSP coefficients:
   a. Convert the average LSP to LSF, then reorder LSF and multiply LSF with weighting coefficients.
   b. Find the LSF reference vector:

$$Lsfref_Index = min\ (\sum (Lsf\ (n) - Lsf_ref_{index}\ (n))^2\ \mid index = 0,...8)$$

   c. Get the LSF residual:

$$Lsf_residual(i) = Lsf(i) - Lsf_ref_{index}(i), \quad i = 0,...,9$$

   d. Quantize the LSF residual, using split band vector quantization method.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.

EncDTXHandler_GSMAMR

Determines the SID flag of current frame.

Syntax

```
IppStatus ippsEncDTXHandler_GSMAMR_16s(Ipp16s*  pValResultHangOverCount,
    Ipp16s* pValResultDtxElapsedCount, Ipp16s*  pValResultUsedMode, Ipp16s*
    pResultSidFlag, Ipp16s vadFlag);
```

Parameters

pValResultHangOverCount	Pointer to the DTX hangover count. When initialized or reset, it is set to 0. On output, points to the updated DTX hangover count.
pValResultDtxElapsedCount	Pointer to elapsed frame count since last non-DTX frame. When initialized or reset, it is set 0. On output, points to the updated elapsed frame count since last non-DTX frame.
pValResultUsedMode	Pointer to the transmission mode. At the input stage, the mode is one of the bit rate modes ranging from 4.75 Kbps to 12.2 Kbps. At the output stage, this value is either unchanged or set to the DTX frame mode.

pResultSidFlag	Pointer to the output SID flag, “1” indicates a SID frame, and “0” indicates a non-SID frame.
vadFlag	This is the VAD flag of the current frame, if it is set 1, the current frame is marked with voiced, and if it is set to 0, it is marked with unvoiced.

Description

The function `ippsEncDTXHandler_GSMAMR` is declared in `ippsc.h` file. This function determines the SID flag of current frame, and it determines whether the current frame should use DTX encoding.

1. Update the elapsed frame count since last SID frame: *DTX_ElapsedCount* = *DTX_ElapsedCount* + 1(Bounded to 0~0x7fff).
2. If the VAD flag of current frame is 1 (voiced frame), the DTX hangover count is set to 7, and this function ends.
3. If the VAD flag of current frame is 0 (unvoiced frame), and the DTX hangover count is 0, the transmission mode of this frame is set to DTX frame mode, and the elapsed frame count since last DTX frame is set to 0, the SID flag is set to 1.
4. If the VAD flag of current frame is 0 (unvoiced frame), but the DTX hangover count is not 0, then decrease DTX hangover count by 1, and if *DTX_HangOver_Count* + *DTX_ElapsedCount* < 30
   The SID flag is set to 0, and the transmission mode is set to “MRDTX”.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.

EncDTXBuffer_GSMAMR , DecDTXBuffer_GSMAMR

Buffer the LSP (or LSF) coefficients and previous log energy coefficients.

Syntax

```
IppStatus ippsEncDTXBuffer_GSMAMR_16s(const Ipp16s*  pSrcSpch, const Ipp16s*
    pSrcLsp, Ipp16s* pValResultUpdateIndex, Ipp16s*  pSrcDstLspBuffer, Ipp16s*
    pSrcDstLogEnergyBuffer);
```

```
IppStatus  ippsDecDTXBuffer_GSMAMR_16s(const Ipp16s*  pSrcSpch, const Ipp16s*
    pSrcLsf, Ipp16s* pValResultUpdateIndex, Ipp16s* pSrcDstLsfBuffer, Ipp16s*
    pSrcDstLogEnergyBuffer);
```

Parameters

pSrcSpch	Pointer to the input speech signal, in the length of 160, in Q15.0
pSrcLsp	Pointer to the LSP for this frame, in the length of 10, in Q0.15.
pSrcLsf	Pointer to the LSF coefficients of the current frame, in the length of 10, in Q0.15.
pValResultUpdateIndex	Pointer to the previous memory update index. On output, points to the current memory update index. It is circularly increased between 0 and 7.
pSrcDstLspBuffer	Pointer to the LSP coefficients of eight previous frames, in the length of 80, in Q0.15. On output, points to the LSP coefficients of eight most recent frames (including current frame), in the length of 80, in Q0.15.
pSrcDstLsfBuffer	Pointer to the LSF coefficients of eight previous frames, in the length of 80, in Q0.15.
pSrcDstLogEnergyBuffer	Pointer to the logarithm energy coefficients of eight previous frames, in the length of 8, in Q5.10. On output, points to the log energy coefficients of eight most recent frames (including current frame), in the length of 8, in Q5.10.

Description

The functions `ippsEncDTXBuffer_GSMAMR` and `ippsDecDTXBuffer_GSMAMR` are declared in `ippsc.h` file. These functions buffer the LSP (or LSF) coefficients and previous log energy coefficients. These LSPs (or LSFs) and energy coefficients will be used for SID frame to extract necessary parameters. The memory update index indicates which part of the buffer will be updated, and it saves the cost for some memory copy. The log energy is computed as follows:

$$EnLog = \log 2\left(\frac{1}{N}\sum_{n=0}^{N-1} s^2(n)\right),$$

where N is the frame length, and s is the input speech signal.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the input or output pointers is NULL.

Post Processing

PostFilter_GSMAMR

Filters the synthesized speech.

Syntax

```
IppStatus ippsPostFilter_GSMAMR_16s(const Ipp16s*  pSrcQLpc, const Ipp16s*
    pSrcSpch, Ipp16s*  pValResultPrevResidual,
    Ipp16s* pValResultPrevScalingGain, Ipp16s* pSrcDstFormantFIRState, Ipp16s*
    pSrcDstFormantIIRState, Ipp16s*  pDstFltSpch, IppSpchBitRate mode);
```

Parameters

pSrcQLpc	Pointer to the reconstructed LP coefficients, in the length of 44, in Q3.12.
pSrcSpch	Pointer to the start position of the input speech signal for current frame, in the length of 160, in Q15.0.
pValResultPrevResidual	On entry, pointer to the last output of the FIR filter of the formant filter for previous subframe, in Q15.0. It is the input of the tilt compensation filter. On exit, points to the last output of the FIR filter of the formant filter for this subframe (in Q15.0) and is the output of the tilt compensation filter. This value is initialized to 0 and can only be updated by this function.
pValResultPrevScalingGain	Pointer to the scaling factor *b* of the last signal for the previous subframe, in Q3.12. On output, points to the scaling factor *b* of the last signal for this subframe, in Q3.12

pSrcDstFormantFIRState	Pointer to the state of the FIR part of the formant filter, in the length of 10, in Q15.0. On output, points to the updated state of the FIR part of the formant filter, in the length of 10, in Q15.0.
pSrcDstFormantIIRState	Pointer to the state of the IIR part of the formant filter, in the length of 10, in Q15.0. On output, points to the updated state of the IIR part of the formant filter, in the length of 10, in Q15.0.
pDstFltSpch	Pointer to the filtered speech, in the length of 160, in Q15.0.
mode	The source transmission mode, for this function, the value between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` is valid.

Description

The function `ippsPostFilter_GSMAMR` is declared in `ippsc.h` file. This function filters the synthesized speech to enhance reconstruction quality. This primitive implements the following procedure:

1. Get the weighted LP coefficients of the formant postfilter:

$$H_f(z) = \frac{\hat{A}(z/\gamma_n)}{A(z/\gamma_d)}$$

For 12.2 and 10.2 Kbps modes,

$$\gamma_n = 0.7, \quad \gamma_d = 0.75$$

For other modes,

$$\gamma_n = 0.55, \quad \gamma_d = 0.7$$

Then filter the input speech with $HF(z) = \hat{A}(z/\gamma_n)$.

2. Compute the impulse response $h(n)$ of the formant postfilter, then get the first reflection coefficients:

$$k_1' = \frac{r_h(1)}{r_h(0)} \quad , \quad r_h(i) = \sum_{j=0}^{21-i} h(j) \times h(j+i)$$

Then compute the tilt factor: $\mu = \gamma_t k'_1$

For 12.2 and 10.2 Kbps mode,

$$\gamma_t = \begin{cases} 0.8 & k'_1 > 0, \\ 0 & \text{otherwise} \end{cases}$$

For other modes, $\gamma_t = 0.8$

Then filter the signal through the tilt compensation filter: $H_t(z) = 1 - \mu z^{-1}$

The output of the tilt compensation filter is filtered by:

$$HI(z) = \hat{A}(z/\gamma_d)$$

3. Compute the adaptive gain scaling factor:
4.

$$\gamma_{SC} = \sqrt{\frac{\sum_{n=0}^{39} \hat{s}(n)}{\sum_{n=0}^{39} \hat{s}_f(n)}}$$

Here, $\hat{s}(n)$ is the synthesized speech, and $\hat{s}_f(n)$ is the post-filtered speech. The output speech is given by:

$$\hat{s}'(n) = \beta_{SC}(n)\hat{s}_f(n)$$

The scaling factor is updated sample-by-sample:

$$\beta_{SC}(n) = 0.9\beta_{SC}(n-1) + 0.1\gamma_{SC}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.r the input variable *mode* is out of range.
`ippStsRangeErr`	Indicates an error when the input variable *mode* is out of range.

AMR Wideband Related Functions

The Intel IPP functions described in this section if properly combined can be used to construct the Adaptive Multi-Rate Wideband (AMR WB) Speech Codec compliant to 3rd Generation Partnership Project (3GPP) specification TS 26.173: "*AMR Wideband Speech Codec; ANSI-C code*". The description of AMR WB codec may be found in 3GPP TS 26.190: "*AMR Wideband Speech Codec; Transcoding functions*". Also it is known as ITU-T G.722.2 AMR WB codec.

The primitives are primarily designed to implement the well-defined, computationally expensive core operations that comprise the codec portion of the AMR WB system. The AMR WB codec comprises an adaptive multi-rate algorithm intended for encoding 7 kHz bandwidth speech signals at the sampling rate of 16 000 samples per second, which results in bit rates for the encoded bit stream of 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s.

The list of Intel IPP AMR WB functions is given in the following table:

Table 9-5 Intel IPP AMR WB Related Functions

Function Base Name	Operation
ResidualFilter_AMRWB	Computes the LPC residual
LPCToISP_AMRWB	Performs LP to ISP coefficients conversion
ISPToLPC_AMRWB	Performs ISP to LP coefficients conversion.
ISPToISF_Norm_AMRWB	Performs ISP to ISF coefficients conversion
ISFToISP_AMRWB	Performs ISF conversion to ISP
OpenLoopPitchSearch_AMRWB	Extracts an open-loop pitch lag estimate from the weighted input speech
HighPassFilterGetSize_AMRWB	Returns the size of the high-pass filter state memory
HighPassFilterInit_AMRWB	Initializes the state memory of high-pass filter
HighPassFilter_AMRWB	Performs high-pass filtering
HighPassFilterGetDlyLine_AMRWB	Receives the parameters of delay line of high-pass filter
HighPassFilterSetDlyLine_AMRWB	Sets the parameters of delay line of high-pass filter
Preemphasize_AMRWB	Computes pre-emphasis of a speech signal
Deemphasize_AMRWB	Performs de-emphasis filtering
SynthesisFilter_AMRWB	Reconstructs the speech signal from LP coefficients and residuals
VADGetSize_AMRWB	Returns the size of the VAD module state memory

Table 9-5 Intel IPP AMR WB Related Functions (continued)

Function Base Name	Operation
VADInit_AMRWB	Initializes the VAD module state memory
VAD_AMRWB	Performs VAD in AMR WB encoder
VADGetEnergyLevel_AMRWB	Gets the vector of energy levels from VAD memory.
AlgebraicCodebookSearch_AMRWB	Performs the fixed (algebraic) codebook search
AlgebraicCodebookDecode_AMRWB	Decodes the fixed (algebraic) codebook indexes
AdaptiveCodebookGainCoeff_AMRWB	Computes the adaptive codebook gain
AdaptiveCodebookSearch_AMRWB	Performs the adaptive codebook search
AdaptiveCodebookDecodeGetSize_AMRWB	Queries the memory length of the adaptive codebook decode module
AdaptiveCodebookDecodeInit_AMRWB	Initializes the adaptive codebook decode module memory
AdaptiveCodebookDecodeUpdate_AMRWB	Updates the adaptive codebook decode module memory
AdaptiveCodebookDecode_AMRWB	Performs the adaptive codebook decoding
ISFQuant_AMRWB	Quantizes the ISF
ISFQuantDecode_AMRWB	Decodes quantized ISFs from the received codebook index
ISFQuantDTX_AMRWB	Quantizes the ISF coefficient vector in case of DTX mode
ISFQuantDecodeDTX_AMRWB	Decodes quantized ISFs in case of DTX mode
GainQuant_AMRWB	Quantizes the adaptive codebook gains
DecodeGain_AMRWB	Decodes adaptive and fixed-codebook gains
EncDTXBuffer_AMRWB	Buffers the ISP coefficients and previous logarithm energy coefficients
DecDTXBuffer_AMRWB	Buffers the ISF coefficients and previous logarithm energy coefficients.

The use of these functions is demonstrated in the Intel IPP *GSM/AMR Wideband/G.722.2 Speech Encoder-Decoder* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

ResidualFilter_AMRWB

Computes the LPC residual

Syntax

```
IppStatus ippsResidualFilter_AMRWB_16s_Sfs (const Ipp16s* pSrcLpc, int order,
    const Ipp16s* pSrcSpeech, Ipp16s* pDstResidualSignal, int len, int
    scaleFactor);
```

Parameters

pSrcLpc	Pointer to the input LPC.
order	Length of the LPC vector.
pSrcSpeech	Pointer to the input vector [-*order*,..,-1,0,...,*len*-1].
pDstResidualSignal	Pointer to the output vector of length [*len*].
len	Length of the vectors.
scaleFactor	Scale factor value.

Description

This function is declared in `ippsc.h` file. The functionality is the same as for the function ippsResidualFilter_G729E_16s. The only difference is that the function `ippsResidualFilter_AMRWB` scales the result according to *scaleFactor* value.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLpc, pSrcSpeech or pDstResidualSignal* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *order* or *len* is less than or equal to zero, or when *order* is greater than *len*.
`ippStsScaleRangeErr`	Indicates an error when *scaleFactor* is negative or *scaleFactor* is greater than 15.

LPC Analysis Functions

LPCToISP_AMRWB

Performs LP to ISP coefficients conversion.

Syntax

```
IppStatus ippsLPCToISP_AMRWB_16s( const Ipp16s* pSrcLpc, Ipp16s* pDstIsp,
    const Ipp16s* pSrcPrevIsp);
```

Parameters

pSrcLpc	Pointer to the input predictor coefficients.
pDstIsp	Pointer to the output immittance spectral pairs.
pSrcPrevIsp	Pointer to the input previous immittance spectral pairs.

Description

This function is declared in `ippsc.h` file. The function performs the following steps:

1. Calculates the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the following recursive relations:

$i = 0,1,\ldots,7$,

$f_1(i) = a_i + a_{m-i}$

$f_2(i) = a_i - a_{m-i}$

$f_1(8) = 2a_8$

where $f_2(-2) = f_2(-1) = 1.0$.

2. Uses Chebyshev polynomials to evaluate $F_1(z)$ and $F_2(z)$. The Chebyshev polynomials are given by:

$$C_1(\omega) = \sum_{i=0}^{7} f_1(i)\cos((8-i)\omega) + f_1(8)/2$$

$$C_2(\omega) = \sum_{i=0}^{6} f_2(i)\cos((8-i)\omega) + f_2(7)/2$$

$F_1(z)$ and $F_2(z)$ polynomials are evaluated at 100 points in equally spaced intervals between 0 and π and are checked for sign changes. A sign change indicates the existence of a root in corresponding interval, which is divided then four times to track the root.

3. If all 16 roots needed to determine ISP coefficients are not found, the function returns the previous set of ISP coefficients instead.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLpc*, *pDstIsp* or *pSrcPrevIsp* pointer is `NULL`.

ISPToLPC_AMRWB

Performs ISP to LP coefficients conversion.

Syntax

```
IppStatus ippsISPToLPC_AMRWB_16s(const Ipp16s* pSrcIsp, Ipp16s* pDstLpc, int
    len);
```

Parameters

pSrcIsp	Pointer to the input immittance spectral pairs.
pDstLpc	Pointer to the output predictor coefficients.
len	Number of ISP and LPC.

Description

This function is declared in `ippsc.h` file. The function performs the following steps:

1. Calculates the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursive relations from
i equal to 2 to *len*/2 by formulae

$$f_1(i) = -2q_{2i-2}f_1(i-1) + 2f_1(i-2)$$

for `j=i-1` down to 2

$$f_1(j) = f_1(j) - 2q_{2i-2}f_1(j-1) + 2f_1(j-2)$$

end

$$f_1(1) = f_1(1) - 2q_{2i-2}$$

with initial values $f_1(0) = 1$ and $f_1(1) = -2q_0$.

The coefficients $f_2(i)$ are computed similarly by replacing q_{2i-2} by q_{2i-1} and $len/2$ by $len/2-1$, and with initial conditions $f_2(0) = 1$ and $f_2(1) = -2q_1$.

2. Once the coefficients $f_1(z)$ and $f_2(z)$ are found, $F_2(z)$ is multiplied by $1-z-2$ to obtain $F_2'(z)$, that is

$$f_2'(i) = f_2(i) - f_2(i-2), \quad i = 2,..,len/2-1$$

$$f_1'(i) = f_1(i), \quad i = 0,..,len/2$$

Then $F_1'(z)$ and $F_2'(z)$ are multiplied by $1+q_{len-1}$ and $1-q_{len-1}$, respectively. That is

$$f_2'(i) = (1 - q_{m-1}) f_2'(i), \quad i = 0,..,len/2-1$$

$$f_1'(i) = (1 + q_{m-1}) f_1'(i), \quad i = 0,..,len/2$$

3. Finally, the function computes the LP coefficients from $f_1'(i)$ and $f_2'(i)$ as:

$$a_i = \begin{cases} 0.5 f_1'(i) + 0.5 f_2'(i), & i = 1,..,len/2-1 \\ 0.5 f_1'(i) - 0.5 f_2'(i), & i = len/2+1,..,len-1 \\ 0.5 f_1'(len/2), & i = len/2 \\ q_{len-1}, & i = len \end{cases}$$

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when the *pSrcIsp or pDstLpc* pointer is NULL.
IppStsSizeErr	Indicates an error when *len* is less than or equal to zero, or when *len* is greater than 20.

ISPToISF_Norm_AMRWB

Performs ISP to ISF coefficients conversion.

Syntax

```
IppStatus ippsISPToISF_Norm_AMRWB_16s(const Ipp16s* pSrcIsp, Ipp16s* pDstIsf,
    int len);
```

Parameters

pSrcIsp	Pointer to the ISP input vector of values scaled in range [-1:1].
pDstIsf	Pointer to the ISF output vector of values scaled in range [0:0.5].
len	Number of ISP and ISF.

Description

This function is declared in `ippsc.h` file. The function `ippsISPToISF_Norm_AMRWB_16s` converts the ISP coefficients to ISF coefficients as follows:

$$pDstIsf[i] = \begin{cases} \frac{f_s}{2\pi}\arccos(pSrcIsp[i]), & i = 0,\ldots len-1 \\ \frac{f_s}{4\pi}\arccos(pSrcIsp[i]) & i = len \end{cases}$$

Here f_s = 12800 is the sampling frequency. The scale factor is chosen such that the first ISF coefficient is normalized to the interval [0:0.5] by multiplying to 1/π.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcIsp or pDstIsf* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

ISFToISP_AMRWB

Performs ISF conversion to ISP.

Syntax

```
IppStatus ippsISFToISP_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s* pDstIsp, int
    len);
```

Parameters

pSrcIsf	Pointer to the ISF input vector of values scaled in range [0:0.5].
pDstIsp	Pointer to the ISP output vector of values scaled in range [-1:1].
len	Number of ISP and ISF.

Description

This function is declared in `ippsc.h` file. It converts ISF to the ISP coefficients as given by:

$$pDstIsp[i] = \cos(pSrcIsf[i]), \quad i = 0, \ldots, len$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcIsf or pDstIsp* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

Open-loop Pitch Analysis Functions

OpenLoopPitchSearch_AMRWB

Extracts an open-loop pitch lag estimate from the weighted input speech.

Syntax

```
IppStatus ippsOpenLoopPitchSearch_AMRWB_16s(const Ipp16s* pSrcWgtSpch,
    const Ipp16s* pSrcFltWgtSpch, Ipp16s* pPrevMidPitchLag, Ipp16s*
    pAdaptiveParam, Ipp16s* pDstOpenLoopLag, Ipp16s* pToneFlag, Ipp16s*
    pDstOpenLoopGain, Ipp16s* pSrcDstPrevPitchLag, Ipp16s* pSrcDstLagSwitcher,
    int len );
```

Parameters

pSrcWgtSpch	Pointer to a 320-element vector containing perceptually weighted speech.
pSrcFltWgtSpch	Pointer to a 320-element vector containing filtered through high-pass filter perceptually weighted speech.
pPrevMidPitchLag	Pointer to the median filtered pitch lag of the 5 previous voiced speech half-frames.
pAdaptiveParam	Pointer to the adaptive parameter.
pDstOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags.
pToneFlag	Pointer to the tone flag for the VAD module.
pDstOpenLoopGain	Pointer to a 2-element vector containing optimal open-loop pitch gains.
pSrcDstPrevPitchLag	Pointer to the five-element vector that contains the pitch lags associated with the five most recent voiced speech half-frames.
pSrcDstLagSwitcher	Switches lag weighting on and off.
len	Length of the frame.

Description

This function is declared in `ippsc.h` file. The open-loop pitch (OLP) search function `ippsOpenLoopPitchSearch_AMRWB_16s` extracts a pitch estimate from a weighted version of the input speech. The OLP search details are as follows:

1.Compute a windowed autocorrelation of weighted speech, that is,

$$R(k) = w(k) \times \sum_{n=0}^{len} sw(n) \times sw(n-k), \quad 17 \le k \le 115$$

where the sequence *sw*(*n*) contains weighted speech, and *w* is the weighting function given by

$$w(k) = wl(k) \times wn(k)$$

In this weighting function, *wl* emphasizes low pitch and is defined in terms of the table *cw*, while *wn* is a sequence of neighboring emphasis lag coefficients associated with the previous frame, that is,

$$wn(k) = \begin{cases} cw(|T_{old} - k| + 98), & pSrcDstLagSwitcher = 1 \\ 1.0, & otherwise \end{cases}$$

and the parameter T_{old} is the median filtered pitch lag of 5 previous voiced speech half frames.

The estimated open-loop pitch lag is the value *k* that maximizes *R*(*k*). It is denoted by T_{op}.

2.Compute the optimal open-loop gain using the relation

$$g = \frac{\sum_{k=0}^{len-1} sw(k) sw(k - T_{op})}{\sqrt{\sum_{k=0}^{len-1} s^2 w(k) \sum_{k=0}^{len-1} s^2 w(k - T_{op})}}$$

In addition, adaptive parameter is re-calculated

$$v = \begin{cases} 1.0, & g > 0.6 \\ 0.9v, & otherwise \end{cases}$$

and

$$\text{pSrcDstLagSwitcher} = \begin{cases} 1, & v > 0.8 \\ 0, & otherwise \end{cases}$$

If $g > 0.6$, the previous pitch lag buffer and median pitch lag of the previous pitch lags are updated.

Update the tone flag as follows:

toneflag = *toneflag*>>*1*

and if g>0.65

toneflag = *toneflag* | 0x4000.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcWgtSpch`*, *`pSrcFltWgtSpch`* , *`pPrevMidPitchLag`*, *`pAdaptiveParam`*, *`pDstOpenLoopLag`*, *`pToneFlag`*, *`pDstOpenLoopGain`*, *`pSrcDstPrevPitchLag`* or *`pSrcDstLagSwitcher`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to zero.

Filtering Functions

HighPassFilterGetSize_AMRWB

Returns the size of the high-pass filter state memory.

Syntax

```
IppStatus ippsHighPassFilterGetSize_AMRWB_16s (int order, int* pDstSize);
```

Parameters

order	The order of high-pass filter (orders of 2 or 3 are currently supported).
pDstSize	Pointer to the output value of the memory size.

Description

This function is declared in `ippsc.h` file. It returns the size of memory needed for proper operation of the high-pass filter of the given *order*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsRangeErr`	Indicates an error when *order* is not equal to either 2 or 3.
`ippStsNullPtrErr`	Indicates an error when the *pDstSize* pointer is `NULL`.

HighPassFilterInit_AMRWB

Initializes the state memory of high-pass filter.

Syntax

```
IppStatus ippsHighPassFilterInit_AMRWB_16s(Ipp16s* pFilterCoeffA, Ipp16s*
    pFilterCoeffB, int order, IppsHighPassFilterState_AMRWB_16s* pState);
```

Parameters

order	The order of high-pass filter; orders of 2 and 3 are currently supported.
pFilterCoeffA	Pointer to the vector of *order* size containing the IIR part of the filter coefficients .
pFilterCoeffB	Pointer to the vector of *order* size containing the FIR part of the filter coefficients.
pState	Pointer to the memory supplied for filtering.

Description

This function is declared in `ippsc.h` file. It sets the coefficients and initializes the state memory of the high-pass filter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pFilterCoeffA, pFilterCoeffB or pState* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *order* is not equal to either 2 or 3.

HighPassFilter_AMRWB

Performs high-pass filtering.

Syntax

```
IppStatus ippsHighPassFilter_AMRWB_16s_Sfs (const Ipp16s* pSrc, Ipp16s* pDst,
    int len, IppsHighPassFilterState_AMRWB_16s* pState, int scaleFactor);
IppStatus ippsHighPassFilter_AMRWB_16s_ISfs (Ipp16s* pSrcDst, int len,
    IppsHighPassFilterState_AMRWB_16s* pState, int scaleFactor);
IppStatus ippsHighPassFilter_Direct_AMRWB_16s (const Ipp16s* pSrcCoeff, const
    Ipp16s* pSrc, Ipp16s* pDst, int len, int borderMode);
```

Parameters

pSrc	Pointer to the vector containing input signal.
pDst	Pointer to the vector containing filtered signal.
pSrcDst	Pointer to the input/output vector.

pState	Pointer to the memory supplied for filtering.
scaleFactor	Scale factor of the input signal.
pSrcCoeff	Pointer to a 2-element vector containing the coefficients of the filter.
len	Length of the input and output vectors.
borderMode	if *borderMode* = 0 then *pSrc*[-1] and *pSrc*[*len*] are equal to zero; otherwise these values must be given.

Description

These functions are declared in `ippsc.h` file.

ippsHighPassFilter_AMRWB_16s. This function performs the high-pass filtering of the input signal by the following transfer function:

$$H_h(z) = \frac{b_0 + b_1 z^{-1} \ldots + b_{order-1} z^{-order}}{a_0 + a_1 z^{-1} \ldots + a_{order-1} z^{-order}}$$

Currently, only a_0 = 1.0 is supported. The scale factor is used to update the filter memory according to the input signal.

ippsHighPassFilter_Direct_AMRWB_16s. This function performs the high-pass filtering of the input signal by the following transfer function:

$$H_h(z) = \frac{a_0}{1 + a_1 z^{-1}}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pSrcCoeff*, *pSrcDst*, *pState* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

HighPassFilterGetDlyLine_AMRWB

Receives the parameters of delay line of high-pass filter.

Syntax

```
IppStatus ippsHighPassFilterGetDlyLine_AMRWB_16s(const
    IppsHighPassFilterState_AMRWB_16s* pState, Ipp16s* pDlyLine, Ipp32s order);
```

Parameters

pState	Pointer to the memory supplied for filtering.
pDlyLine	Pointer to the vector (6-elemnt size for order 2, or 9-element size for order 3) containing the filter memory.
order	The order of high-pass filter; possible value is 2 or 3.

Description

The function `ippsHighPassFilterGetDlyLine_AMRWB` is declared in `ippsc.h` file. This function gets the parameters of the delay line of the high-pass filter structure. The layout of the returned filter memory is follows:

Yh_2, Yl_2, Yh_1, Yl_1, X_0, X_1 - for the high-pass filter of order 2.

Yh_3, Yl_3, Yh_2, Yl_2, Yh_1, Yl_1, X_0, X_1 X_2 - for order 3 high pass filter,

where X_i - the memory of the FIR filter part, and Yh_i, Yl_i - the memories of the IIR filter part.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *order* is not equal 2 or 3.

HighPassFilterSetDlyLine_AMRWB

Sets the parameters of delay line of high-pass filter.

Syntax

```
IppStatus ippsHighPassFilterSetDlyLine_AMRWB_16s(const Ipp16s* pDlyLine,
    IppsHighPassFilterState_AMRWB_16s* pState, Ipp32s order);
```

Parameters

pDlyLine	Pointer to the vector (6-elemnt size for order 2, or 9-element size for order 3) containing the filter memory.
pState	Pointer to the memory supplied for filtering.
order	The order of high-pass filter; possible value is 2 or 3.

Description

The function `ippsHighPassFilterSetDlyLine_AMRWB` is declared in `ippsc.h` file. This function updates *pState* structure by *pDlyLine* vector with the parameters of the delay line of the high-pass filtering. The layout of the *pDlyLine* vector is the same as in the description of the function `ippsSNR_AMRWBE`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *order* is not equal 2 or 3.

Preemphasize_AMRWB

Computes pre-emphasis of a speech signal.

Syntax

```
IppStatus ippsPreemphasize_AMRWB_16s_ISfs (Ipp16s gamma, Ipp16s* pSrcDst, int
    len, int scaleFactor,  Ipp16s* pMem);
```

Parameters

gamma	Filter coefficient.
pSrcDst	Pointer to the input/output vector.
scaleFactor	Scale factor for the result.
len	Length of the input/output vector.
pMem	Pointer to the filter memory of length 1.

Description

This function is declared in `ippsc.h` file. It computes pre-emphasis of the input speech according to the difference signal pre-emphasis equation:

$$H(z) = 1 - (\gamma \cdot z^{-1})$$

The memory value *pMem*[0] is updated by *pSrcDst*[n-1]. For proper use of this function in AMR WB codec, the memory value must be initialized to zero in the user program.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDst or pMem* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.
`ippStsScaleRangeErr`	Indicates an error when *scaleFactor* < 0 or *scaleFactor* > 15.

Deemphasize_AMRWB

Performs de-emphasis filtering.

Syntax

```
IppStatus ippsDeemphasize_AMRWB_NR_16s_I(Ipp16s gamma, Ipp16s* pSrcDst, Ipp16s
    len, Ipp16s* pMem);
IppStatus ippsDeemphasize_AMRWB_32s16s(Ipp16s gamma, const Ipp32s* pSrc,
    Ipp16s* pDst, int len, Ipp16s* pMem);
```

Parameters

gamma	De-emphasis factor.

pSrc	Pointer to the input vector.
pDst	Pointer to the output vector.
pSrcDst	Pointer to the input/output vector.
len	Length of the input/output vector.
pMem	Pointer to the filter memory element.

Description

This function is declared in `ippsc.h` file. It performs de-emphasis of the input synthesized signal by filtering it through the filter with the following transfer function:

$$H(z) = \frac{1}{1 - \gamma z^{-1}}$$

The initial memory of the filter must be set to zero. The memory value *pMem*[0] is updated by *pDst*[*len* -1] or *pSrcDst*[*len* -1].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *pSrcDst* or *pMem* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

SynthesisFilter_AMRWB

Reconstructs the speech signal from LP coefficients and residuals.

Syntax

```
IppStatus ippsSynthesisFilter_AMRWB_16s32s_I(const Ipp16s* pSrcLpc, int  order,
    const Ipp16s* pSrcExc, Ipp32s* pSrcDstSignal, int len);
```

Parameters

pSrcLpc	Pointer to the LP coefficients vector $a_0, a_1, \ldots, a_{order}$
order	Order of LP filter.

pSrcExc	Pointer to the excitation vector.
pSrcDstSignal	Pointer to the synthesized and updated speech.
len	Length of the filter.

Description

This function is declared in `ippsc.h` file. It computes the filter given by:

$$H(z) = \frac{1}{a_0 + \sum_{i=1}^{order} a_i z^{-i}}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLpc, pSrcExc or pSrcDstSignal* pointer is `NULL`.
`IppStsSizeErr`	Indicates an error when *len or order* is less than or equal to zero.

Discontinuous Transmission (DTX) Functions

VADGetSize_AMRWB

Returns the size of the VAD module state memory.

Syntax

```
IppStatus ippsVADGetSize_AMRWB_16s (int* pDstSize);
```

Parameters

pDstSize	Pointer to the output value of the memory size.

Description

This function is declared in `ippsc.h` file. It returns the size of memory needed for proper operation of the VAD module in the AMR WB codec .

Return Values

IppStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when the *pDstSize* pointer is NULL.

VADInit_AMRWB

Initializes the VAD module state memory.

Syntax

```
IppStatus ippsVADInit_AMRWB_16s(IppsVADState_AMRWB_16s* pState);
```

Parameters

pState	Pointer to the VAD memory.

Description

This function is declared in ippsc.h file. It initializes the state of VAD (Voice Activity Decision) module of the AMR WB codec. The structure IppsVADState_AMRWB_16s contains the VAD history that must be initialized prior to the use of the function ippsVAD_AMRWB_16s.

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when the *pState* pointer is NULL.

VAD_AMRWB

Performs VAD in AMR WB encoder.

Syntax

```
IppStatus ippsVAD_AMRWB_16s(const Ipp16s* pSrcSpch, IppsVADState_AMRWB_16s*
    pSrcDstVadState, Ipp16s* pToneFlag, Ipp16s* pVadFlag);
```

Parameters

pSrcSpch	Pointer to the input speech signal, of length 320.
pSrcDstVadState	Pointer to the VAD memory.
pToneFlag	Pointer to the output tone flag.
pVadFlag	Pointer to the VAD flag of this frame.

Description

This function is declared in `ippsc.h` file. The function `ippsVAD_AMRWB_16s` implements the Voice Activity Decision (VAD) functionality of the AMR WB encoder. This function is used to indicate whether the input speech frame contains active speech or some other audio signal such as silence or music.
The structure `IppsVADState_AMRWB_16s` contains the VAD history and is updated in the function. This structure should be initialized in advance by using the function `ippsVADInit_AMRWB_16s`.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pSrcSpch, pSrcDstVadState, pToneFlag or pVadFlag* pointer is `NULL`.

VADGetEnergyLevel_AMRWB

Gets the vector of energy levels from VAD memory.

Syntax

```
IppStatus ippsVADGetEnergyLevel_AMRWB_16s(const IppsVADState_AMRWB_16s* pState,
    Ipp16s* pEnergyLevel);
```

Parameters

pState	Pointer to the VAD memory.
pEnergyLevel	Pointer to the energy level vector [12].

Description

The function ippsVADGetEnergyLevel_AMRWB is declared in ippsc.h file. The function ippsVAD_AMRWB produces signal energy in the 12 non-uniform bands over the frequency range from 0 to 6.4 kHz for 256-sample frame. And the function ippsVADGetEnergyLevel_AMRWB returns normalized energy levels which are produced by dividing the energy level form each band by the width of this band in Hz. The lower index refers to the lower sub band.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.

Codebook Search Functions

AlgebraicCodebookSearch_AMRWB

Performs the fixed (algebraic) codebook search.

Syntax

```
IppStatus ippsAlgebraicCodebookSearch_AMRWB_16s(const Ipp16s* pSrcFixedTarget,
    const Ipp16s* pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse, Ipp16s*
    pDstFixedVector, Ipp16s* pDstFltFixedVector, IppSpchBitRate mode, Ipp16s*
    pDstIndex);
```

Parameters

pSrcFixedTarget	Target vector.
pSrcLtpResidual	Long term prediction residual.
pSrcDstImpulseResponse	Impulse response of the weighted synthesis filter; *pSrcDstImpulseResponse*[-L_subfr..-1] must be set to zero.
pDstFixedVector	Innovative codebook.
pDstFltFixedVector	Filtered fixed codebook excitation.
mode	Coder mode.

pDstIndex	Indexes of the pulses.

Description

This function is declared in `ippsc.h` file. It performs the following steps:

1.Computes backward filtered target vector **d** by

$$d(n) = \sum_{i=n}^{63} x_2(i) \times h(i-n), \quad 0 \le n \le 63$$

where $x_2(n)$ is the fixed target signal used for fixed codebook search.

2.Computes the signal $b(n)$, which is used for presetting the pulse amplitudes, as:

$$b(n) = \sqrt{\frac{\sum_{i=0}^{63} d(i)d(i)}{\sum_{i=0}^{63} res_{LTP}(i)res_{LTP}(i)}}\, res_{LTP}(n) + \alpha d(n), \quad 0 \le n \le 63$$

The scaling factor α controls the amount of dependence of the reference signal on $d(n)$, it is lowered as the bit rate is increased. Here α=2 for 6.6 and 8.85 modes; α=1 for 12.65, 14.25 and 15.85 modes; α=0.8 for 18.25 mode; α=75 for 19.25 mode; and α=0.5 for 23.05 and 23.85 modes.

3.Calculates the symmetric Toepliz matrix Φ for each mode. The element of this matrix is computed by

$$\Phi(i,j) = \sum_{n=j}^{63} h(n-i)h(n-j), \quad i = 0, ..., 63, \; j = 0, ..., 63$$

4.Searches signed pulses in appropriate positions and encodes them. The pulse positions for different modes are in the tables below:

Modes 23.05 and 23.85:

Track	Pulse	Positions
1	$i_0, i_4, i_8, i_{12}, i_{16}, i_{20}$	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9, i_{13}, i_{17}, i_{21}$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	$i_2, i_6, i_{10}, i_{14}, i_{18}, i_{22}$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	$i_3, i_7, i_{11}, i_{15}, i_{19}, i_{23}$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 19.85:

Track	Pulse	Positions
1	$i_0, i_4, i_8, i_{12}, i_{16}$	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9, i_{13}, i_{17}$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6, i_{10}, i_{14}	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7, i_{11}, i_{15}	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 18.25:

Track	Pulse	Positions
1	i_0, i_4, i_8, i_{12}	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60

2	i_1, i_5, i_9, i_{13}	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6, i_{10}, i_{14}	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7, i_{11}, i_{15}	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 15.85:

Track	Pulse	Positions
1	i_0, i_4, i_8	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	i_1, i_5, i_9	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6, i_{10}	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7, i_{11}	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 14.25:

Track	Pulse	Positions
1	i_0, i_4, i_8	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	i_1, i_5, i_9	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 12.65:

Track	Pulse	Positions
1	i_0, i_4	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	i_1, i_5	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2, i_6	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3, i_7	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 8.85:

Track	Pulse	Positions
1	i_0	0, 4, 8, 12, 16, 20, 24, 28, 32 36, 40, 44, 48, 52, 56, 60
2	i_1	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	i_2	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	i_3	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Mode 6.60:

Track	Pulse	Positions

1	i_0	0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62
2	i_1	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcFixedTarget*, *pSrcLtpResidual*, *pSrcDstImpulseResponse*, *pDstFixedVector*, *pDstFltFixedVector* or *pDstIndex* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

AlgebraicCodebookDecode_AMRWB

Decodes the fixed (algebraic) codebook indexes.

Syntax

```
IppStatus ippsAlgebraicCodebookDecode_AMRWB_16s (const Ipp16s* pSrcIdxs,
    Ipp16s* pDstFixedCode, IppSpchBitRate mode);
```

Parameters

pSrcIdxs	Algebraic codebook indexes.
pDstFixedCode	Pointer to the algebraic code vector of length 64.
mode	Coder mode.

Description

This function is declared in `ippsc.h` file. The function decodes the fixed codebook vector from the received fixed codebook index.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcIdxs or pDstFixedCode* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not an element of the enumerated type `IppSpchBitRate` appropriate to AMR WB.

AdaptiveCodebookGainCoeff_AMRWB

Computes the adaptive codebook gain.

Syntax

```
IppStatus ippsAdaptiveCodebookGainCoeff_AMRWB_16s(const Ipp16s*
    pSrcAdptTarget, const Ipp16s*  pSrcFltAdptVector, Ipp16s*  pAdptGainCoeffs,
    Ipp16s*  pResultAdptGain, int len);
```

Parameters

pSrcAdptTarget	Pointer to the adaptive-codebook vector.
pSrcFltAdptVector	Pointer to the input filtered adaptive-codebook vector.
pResultAdptGain	Pointer to the adaptive-codebook gain in the length of 1.
pAdptGainCoeffs	Pointer to the output vector in the length 4; represents the adaptive-codebook gain as a fraction.
len	Length of vectors.

Description

This function is declared in `ippsc.h` file. The adaptive codebook gain is found by:

$$g_p = \frac{\sum_{n=0}^{len-1} x(n)y(n)}{\sqrt{\sum_{n=0}^{len-1} y(n)y(n)}}$$

bounded by

$$0 \le g_p \le 1.2$$

Additionally, the function returns the gain in a different representation given by the following formula:

$$g_p = \frac{c_{xy} 2^{exp_{xy}}}{c_{yy} 2^{exp_{yy}}}, 1/2 \le |c_{xy}|, |c_{yy}| \le 1$$

The mantissas C_{xy} , C_{yy} and exponents exp_{xy} , exp_{yy} for both the normalized denominator and normalized numerator are returned in the *pAdptGainCoeffs* vector:

pAdptGainCoeffs[0] = C_{yy}

pAdptGainCoeffs[1] = exp_{yy}

pAdptGainCoeffs[2] = C_{xy}

pAdptGainCoeffs[3] = exp_{xy}

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcAdptTarget*, *pSrcFltAdptVector*, *pResultAdptGain* or *pAdptGainCoeffs* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

AdaptiveCodebookSearch_AMRWB

Performs the adaptive codebook search.

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_AMRWB_16s(const Ipp16s*  pSrcAdptTarget,
    const Ipp16s*  pSrcImpulseResponse, const Ipp16s*  pSrcOpenLoopLag, Ipp16s*
    pPitchLag, Ipp16s*  pPitchLagBounds, Ipp16s*  pSrcDstExcitation, Ipp16s*
    pFracPitchLag, Ipp16s*  pAdptIndex, int   subFrame, IppSpchBitRate mode);
```

Parameters

pSrcAdptTarget	Pointer to the 64-element adaptive target signal vector.
pSrcImpulseResponse	Pointer to the 64-element impulse response of the weighted synthesis filter.
pSrcOpenLoopLag	Pointer to a two-element vector of the open-loop pitch lags.
pPitchLag	Pointer to the output previous integral pitch lag.
pPitchLagBounds	Pointer to the bounds of the output previous integral pitch lag.
pSrcDstExcitation	Pointer to the input/output 321-element excitation vector.
pFracPitchLag	Pointer to the output fractional pitch lag obtained during the adaptive codebook search.
pAdptIndex	Pointer to the coded closed-loop pitch index.
subFrame	Subframe number.
mode	Coder mode.

Description

This function is declared in `ippsc.h` file. The function performs the adaptive codebook search. The adaptive codebook search consists of a closed-loop pitch search followed by computation of an adaptive excitation vector. The adaptive excitation vector is computed by interpolating the past excitation at the fractional pitch lag obtained during the closed-loop pitch search.

The adaptive codebook is searched on every subframe. A detailed description of the adaptive codebook search procedure is given next:

1.In the 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, in the first and third

subframes, a fractional pitch delay is used with resolutions **1/4** in the range $\left[34, 127\frac{3}{4}\right]$, resolutions **1/2** in the range $\left[128, 159\frac{1}{2}\right]$, and integers only in the range [**160, 231**]. For the

second and fourth subframes, a pitch resolution of **1/4** is always used in the range $\left[T_1 - 8, T_1 + 7\frac{3}{4}\right]$, where T_1 is the nearest integer to the fractional pitch lag of the previous (1st or

3rd) subframe.

In 6.60 kbit/s mode, in the first subframe, a fractional pitch delay is used with resolutions **1/2** in

the range $\left[34, 91\frac{1}{2}\right]$, and integers only in the range [**92,231**]. For the second, third and fourth

subframes, a pitch resolution of **1/2** is always used in the range $\left[T_1 - 8, T_1 + 7\frac{3}{4}\right]$, where T_1 is the nearest integer to the fractional pitch lag of the first subframe.

Closed-loop pitch analysis is performed around the open-loop pitch estimates on a subframe basis. In 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, in the first (and third) subframe the range $T_{op} \pm 7$, bounded by 34...231, is searched. In 6.60 kbit/s mode, in the first subframe the range $T_{op} \pm 7$, bounded by 34...231, is searched. For all the modes, for the other subframes, closed-loop pitch analysis is performed around the integer pitch selected in the previous subframe, as described above. In 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, the pitch delay is encoded with 9 bits in the first and third subframes and the relative delay of the other subframes is encoded with 6 bits. In 8.85 kbit/s mode, the pitch delay is encoded with 8 bits in the first and third subframes and the relative delay of the other subframes is encoded with 5 bits. In 6.60 kbit/s mode, the pitch delay is encoded with 8 bits in the first subframe and the relative delay of the other subframes is encoded with 5 bits.

2.The closed loop pitch search is performed by minimizing the mean-square weighted error between the original and synthesized speech. This is achieved by maximizing the term

$$R(k) = \frac{\sum_{n=0}^{63} x(n) y_k(n)}{\sqrt{\sum_{n=0}^{63} y_k(n) y_k(n)}}$$

where *x(n)* is the target signal and $y_k(n)$ the past filtered excitation at delay *k* (past excitation convolved with *h(n)*). Note that the search range is limited around the open-loop pitch as explained earlier.

The convolution $y_k(n)$ is computed for the first delay in the searched range, and for the other delays it is updated using the recursive relation

$$y_k(n) = y_{k-1}(n-1) + u(-k)h(n)$$

where

$$u(n), n = -(231+17),\ldots,63$$

is the excitation buffer. Note that in search stage, the samples *u(n)*, *n* = 0,..., 63 are not known, and they are needed for pitch delays less than 64.

3.Let

$$B_{LP} = 0.18z + 0.64 + 0.18z^{-1}$$

be the low-pass filter. Thus the adaptive codebook vector *v(n)* is computed as

$$v(n) = \sum_{i=1}^{l} b_{LP}(i+1) v'(n+i)$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcAdptTarget`*, *`pSrcImpulseResponse`*, *`pSrcOpenLoopLag`*, *`pPitchLag`*, *`pPitchLagBounds`*, *`pSrcDstExcitation`*, *`pFracPitchLag`* or *`pAdptIndex`* pointer is `NULL`.

ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.
ippStsSizeErr	Indicates an error when *subFrame* < 0 or *subFrame* > 3.

AdaptiveCodebookDecodeGetSize_AMRWB

Queries the memory length of the adaptive codebook decode module.

Syntax.

```
IppStatus ippsAdaptiveCodebookDecodeGetSize_AMRWB_16s(int* pDstSize);
```

Parameters

pDstSize	Pointer to the output parameter to store the adaptive codebook decode module length in bytes.

Description

This function is declared in ippsc.h file. The function reports the size of memory needed for proper operation of the adaptive codebook decode module.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDstSize* pointer is NULL.

AdaptiveCodebookDecodeInit_AMRWB

Initializes the adaptive codebook decode module memory.

Syntax

```
IppStatus ippsAdaptiveCodebookDecodeInit_AMRWB_16s(
    IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

Parameters

pState	Pointer to the adaptive codebook decode module memory.

Description

This function is declared in `ippsc.h` file. The function initializes the adaptive codebook decode module memory.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.

AdaptiveCodebookDecodeUpdate_AMRWB

Updates the adaptive codebook decode module memory.

Syntax

```
IppStatus ippsAdaptiveCodebookDecodeUpdate_AMRWB_16s(Ipp16s valIntPitchLag,
    Ipp16s valPitchGain, IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

Parameters

valIntPitchLag	Pointer to the integral pitch lag.
valPitchGain	Pitch gain.
pState	Pointer to the adaptive codebook decode module memory.

Description

This function is declared in `ippsc.h` file. The function updates the adaptive codebook decode module memory with the given pitch lag and pitch gain.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.

AdaptiveCodebookDecode_AMRWB

Performs the adaptive codebook decoding.

Syntax

```
IppStatus ippsAdaptiveCodebookDecode_AMRWB_16s(int valAdptIndex, Ipp16s*
    pResultFracPitchLag, Ipp16s* pSrcDstExcitation, Ipp16s* pPitchLag, Ipp16s*
    pPitchLagBounds, int subFrame, int bfi, int unusableFrame, IppSpchBitRate
    mode, IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

Parameters

valAdptIndex	Adaptive codebook index.
pResultFracPitchLag	Pointer to the fractional pitch lag obtained during the adaptive codebook search.
pSrcDstExcitation	Pointer to the 321-element excitation vector.
pPitchLag	Pointer to the integral pitch lag.
pPitchLagBounds	Pointer to the previous integral pitch lag bounds.
subFrame	Subframe number.
bfi	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
unusableFrame	Value "0" signifies a lost frame with content that can not be used at all.
mode	Encoding mode.
pState	Pointer to the memory of the adaptive codebook decode module.

Description

This function is declared in `ippsc.h` file. It performs the following steps:

1. If no errors are detected on the current frame, the function extracts integer and fractional pitch lags from the adaptive codebook indices.

2. If errors are detected, the function recovers the integer pitch either from the previous integer pitch or the LTP-lag. See clause *6.2.3.4* in 3GPP TS 26.191 V5.1.0.

3. To obtain the adaptive codebook vector, the function `ippsAdaptiveCodebookDecode_AMRWB_16s` applies the same adaptive codebook computation procedure as described in the function `ippsAdaptiveCodebookSearch_AMRWB_16s`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState*, *pResultFracPitchLag*, *pSrcDstExcitation*, *pPitchLag* or *pPitchLagBounds* pointer is NULL.
ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.
ippStsSizeErr	Indicates an error when *subFrame* < 0 or *subFrame* > 3.

Quantization Functions

ISFQuant_AMRWB

Quantizes the ISF.

Syntax

```
IppStatus ippsISFQuant_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s*
    pSrcDstResidual, Ipp16s* pDstQIsf, Ipp16s* pDstQIsfIndex, IppSpchBitRate
    mode);
```

Parameters

pSrcIsf	Pointer to the unquantized 16-element ISF vector.
pSrcDstResidual	Pointer to the 16-element quantized ISF residual from the previous frame.
pDstQIsf	Pointer to the quantized 16-element ISF vector.
pDstQIsfIndex	Pointer to the 7-element vector of quantized ISF indices. For 6.60 Kbps frames, only the first five elements contain valid indices; for all other bit rates, all seven elements contain valid data.
mode	Coder mode.

Description

This function is declared in ippsc.h file. It applies a first order moving-average (MA) prediction and quantifies the residual ISF vector using a combination of split vector quantization (SVQ) and multistage vector quantization (MSVQ).

The prediction and quantization are performed as follows. Let *z(n)* denote the mean-removed ISF vector at frame *n*.

The prediction residual vector *r(n)* is given by:

r(n) = z(n) - p(n)

where *p(n)* is the predicted LSF vector at frame *n*. First order moving-average (MA) prediction is used where:

$$p(n) = \frac{1}{3}\hat{r}(n-1)$$

where $\hat{r}(n-1)$ is the quantized residual vector at the past frame.

The ISF residual vector *r* is quantized using split-multistage vector quantization S-MSVQ. The vector is split into two subvectors $r_1(n)$ and $r_2(n)$ of dimensions 9 and 7, respectively. The two subvectors are quantized in two stages. In the first stage $r_1(n)$ is quantized with 8 bits and $r_2(n)$ with 8 bits.

For 8.85 ,12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, the quantization error vectors are split in the next stage into 3 and 2 subvectors, respectively.

For 6.60 kbit/s mode, the quantization error vectors

$$r_i^{(2)} = r - \hat{r}_i, i = 1,2$$

are split in the next stage into 2 and 1 subvectors, respectively. A squared error ISF distortion measure is used in the quantization process. In general, for an input ISF or error residual subvector r_i , $i = 1,2$ a quantized vector at index

$$k, \hat{r}_i^k$$

the quantization is performed by finding the index *k* which minimizes

$$E = \sum_{i=m}^{n}\left[r_i - \hat{r}_i^k\right]^2$$

where *m* and *n* are the first and last elements of the subvector.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcIsf*, *pSrcDstResidual*, *pDstQIsf* or *pDstQIsfIndex* pointer is NULL.
ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.

ISFQuantDecode_AMRWB

Decodes quantized ISFs from the received codebook index.

Syntax

```
IppStatus ippsISFQuantDecode_AMRWB_16s(const Ipp16s* pSrcIdxs, Ipp16s*
    pDstQntIsf, Ipp16s* pSrcDstResidual, const Ipp16s* pSrcPrevQntIsf, Ipp16s*
    pSrcDstIsfMemory, int bfi, IppSpchBitRate mode);
```

Parameters

pSrcIdxs	Pointer to the seven-element vector containing codebook indices of the quantized LSPs.
pSrcDstResidual	Pointer to the 16-element quantized ISF residual from the previous frame.
pSrcPrevQntIsf	Pointer to the 16-element quantized ISF vector from the previous frame.
pSrcDstIsfMemory	Pointer to the 64-element vector containing four subframe ISF sets.
pDstQntIsf	Pointer to a 16-element destination vector containing quantized ISF in frequency domain (0..0.5).
bfi	Bad frame indicator: value "0" signifies a good frame, all other values signify a bad frame.
mode	Coder mode.

Description

This function is declared in `ippsc.h` file. The function decodes quantized ISFs from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized ISFs from previous quantized ISFs using linear interpolation.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcIdxs, pSrcDstResidual , pSrcPrevQntIsf, pSrcDstIsfMemory or pDstQntIsf* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

ISFQuantDTX_AMRWB

Quantizes the ISF coefficient vector in case of DTX mode.

Syntax

```
IppStatus ippsISFQuantDTX_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s* pDstQntIsf,
    Ipp16s* pDstIdxs);
```

Parameters

pSrcIsf	Pointer to the unquantized 16-element ISF vector in the frequency domain (0..0.5) .
pDstQntIsf	Pointer to the 16-element quantized ISF vector.
pDstIdxs	Pointer to the 5-element vector quantization indices.

Description

This function is declared in `ippsc.h` file. The function quantizes averaged ISF coefficient vector using the comfort noise ISF quantization tables.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrcIsf, pDstQntIsf or pDstIdxs* pointer is NULL.

ISFQuantDecodeDTX_AMRWB

Decodes quantized ISFs in case of DTX mode.

Syntax

```
IppStatus ippsISFQuantDecodeDTX_AMRWB_16s(const Ipp16s* pSrcIdxs, Ipp16s*
    pDstQntIsf);
```

Parameters

pSrcIdxs	Pointer to the 5-element vector quantization indices.
pDstQntIsf	Pointer to the 16-element ISF vector in the frequency domain (0..0.5) .

Description

This function is declared in ippsc.h file. The function decodes quantized ISFs from the received codebook index in case of DTX mode.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcIdxs or pDstQntIsf* pointer is NULL.

GainQuant_AMRWB

Quantizes the adaptive codebook gains.

Syntax

```
IppStatus ippsGainQuant_AMRWB_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
    pSrcFltAdptVector, int valFormat, const Ipp16s*  pSrcFixedVector, const
    Ipp16s* pSrcFltFixedVector, const Ipp16s* pSrcCorr, Ipp16s*
    pSrcDstEnergyErr, Ipp16s* pSrcDstPitchGain, int* pDstCodeGain, int
    valClipFlag, Ipp16s* pDstQGainIndex, int lenSrc, IppSpchBitRate mode);
```

Parameters

pSrcAdptTarget	Pointer to the input target vector x(n).
pSrcFltAdptVector	Pointer to the input filtered adaptive codebook vector y(n).
valFormat	Format of the *pSrcAdptTarget* and *pSrcFltAdptVector* vectors.
pSrcFixedVector	Pointer to the input pre-filtered codebook contribution c(n).
pSrcFltFixedVector	Pointer to the input filtered codebook vector z(n).
pSrcCorr	Pointer to the vector of correlations between the *pSrcAdptTarget*, *pSrcFltAdptVector*, *pSrcFltFixedVector* vectors.
pSrcDstEnergyErr	Pointer to the input/output energy error vector for 4 previous subframes.
pSrcDstPitchGain	Pointer to the input/output pitch gain.
pDstCodeGain	Pointer to the output code gain.
valClipFlag	If *valClipFlag* = 1 , then limit gain pitch to 1.0.
pDstQGainIndex	Pointer to the output codebook indexes found.
lenSrc	Length of the input vectors.
mode	Encoding mode.

Description

This function is declared in `ippsc.h` file. The adaptive codebook gain (pitch gain) and the fixed (algebraic) codebook gain are vector quantized using a 6-bit codebook for modes 8.85 and 6.60 kbit/s and using a 7-bit codebook for all the other modes.

The functionality is as follows:

1.Calculate the average energy minus the mean energy of the fixed-codebook contribution:

$$E = 10 log\left(\frac{1}{lenSrc}\sum_{i=0}^{lenSrc-1} c^2(i)\right)$$

Then calculate

$$E_b - E = 30(dB) - E$$

2.Calculate the predicted energy from prediction error of the previous subframes:

$$\tilde{E}^{m} = \sum_{i=1}^{4} b_i \hat{U}^{(m-1)}$$

where $\hat{U}^{m}$ is calculated by the equation $\hat{U}^{m} = E - \tilde{E}^{m}$ and [*b*1 *b*2 *b*3 *b*4] = [0.5,0.4,0.3,0.2] are the MA prediction coefficients.

3.Calculate g_c (the fixed-codebook gain) by the equation

$$g_c = 10^{0.05(\tilde{E}^{m} + E_b - E)}$$

4.The gain codebook search is performed by minimizing the mean-square of the weighted error between original and reconstructed speech which is given

$$E = x^t x + g_p^2 y^t y + g_c^2 z^t z - 2 g_p x^t y - 2 g_c x^t z + 2 g_p g_c y^t z$$

where *x* is the target vector, *y* is the filtered adaptive codebook vector, and *z* is the filtered fixed codebook vector. (Each gain vector in the codebook also has an element representing the quantized energy prediction error.) The quantized energy prediction error associated with the

chosen gains is used to update $\hat{U}^{m}$. In the search, only the 64 codevectors that are closest to the unquantized pitch gain g_p are taken into account.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcAdptTarget`*, *`pSrcFltAdptVector`*, *`pSrcFixedVector`*, *`pSrcFltFixedVector`*, *`pSrcCorr`*, *`pSrcDstEnergyErr`*, *`pSrcDstPitchGain`*, *`pDstCodeGain`* or *`pDstQGainIndex`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`lenSrc`* is less than or equal to zero.
`ippStsRangeErr`	Indicates an error when *`mode`* is not a valid element of the enumerated type `IppSpchBitRate`.

DecodeGain_AMRWB

Decodes adaptive and fixed-codebook gains.

Syntax

```
IppStatus ippsDecodeGain_AMRWB_16s(int valQIndex, Ipp32s valEnergy, Ipp16s*
    pDstPitchGain, int* pDstCodeGain, int bfi, int prevBfi, Ipp16s*
    pSrcDstPastEnergy, Ipp16s*  pPrevCodeGain, Ipp16s* pSrcDstPastCodeGain,
    IppSpchBitRate mode);
```

Parameters

valQIndex	Index of quantization.
valEnergy	Energy of the code vector.
pDstPitchGain	Pointer to the decoded pitch gain.
pDstCodeGain	Pointer to the decoded code gain.
bfi	Bad frame indicator.
prevBfi	Bad frame indicator of the previous frame.
pSrcDstPastEnergy	Past quantized energies.
pPrevCodeGain	Past code gain.
pSrcDstPastCodeGain	Past code gain for frame erasures.
mode	Decoding mode.

Description

This function is declared in `ippsc.h` file. It performs the following steps.

If there are no errors detected in the frame, the function computes gains as described below:

1.The received index gives the fixed codebook gain correction factor $\hat{\gamma}$.

2.The predicted energy for every subframe m is computed as

$$\tilde{E}^{m} = \sum_{i=1}^{4} b_i \hat{U}^{(m-1)}$$

3.The mean innovation energy is found by

$$E = 10 log\left(\frac{1}{64}\sum_{i=0}^{63} c^2(i)\right)$$

4.The predicted gain g_c' is computed as

$$g_c = 10^{0.05(\tilde{E}^m + E_b - E)}$$

5.The quantized fixed codebook gain is given by

$$\hat{g}_c = \gamma \hat{g}_c'$$

In case of frame erasure, the gains are the attenuated versions of the previous gains.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDstPitchGain, pDstCodeGain, pSrcDstPastEnergy, pPrevCodeGain or pSrcDstPastCodeGain* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

EncDTXBuffer_AMRWB

Buffers the ISP coefficients and previous logarithm energy coefficients.

Syntax

```
IppStatus ippsEncDTXBuffer_AMRWB_16s(const Ipp16s* pSrcSpch, const Ipp16s*
    pSrcIsp, Ipp16s* pUpdateIndex, Ipp16s* pSrcDstIspBuffer, Ipp16s*
    pSrcDstLogEnergyBuffer, IppSpchBitRate mode);
```

Parameters

pSrcSpch	Pointer to the input speech signal of length 256.
pSrcIsp	Pointer to the ISP for this frame, of length 16.
pUpdateIndex	Holds the previous memory update index. On output, it contains the current memory update index. Circularly increased between 0 and 7.
pSrcDstIspBuffer	Pointer to the ISP coefficients of eight previous frames,of length 128. On output, points to the ISP coefficients of eight most recent frames (including the current frame).
pSrcDstLogEnergyBuffer	Pointer to the logarithm energy coefficients of eight previous frames, of length 8. On output, points to the logarithm energy coefficients of eight most recent frames (including the current frame).
mode	Bit rate specifier.Values between IPP_SPCHBR_6600 and IPP_SPCHBR_23850 are valid.

Description

The function `ippsEncDTXBuffer_AMRWB` is declared in `ippsc.h` file. This function buffers the ISP coefficients and previous logarithm energy coefficients. The buffered ISP coefficients and energy coefficients will be used for SID frame to extract necessary parameters.
The memory update index indicates which part of the buffer will be updated and thus saves the cost of some memory copy operation. The logarithm energy is computed as follows:

$$en_{\log}(i) = \frac{1}{2}\log_2\left(\frac{1}{256}\sum_{n=0}^{255}\mathrm{pSrcSpch}^2(n)\right)$$

After computation, the energy is adjusted according to the current encoder mode.

The function `ippsEncDTXBuffer_AMRWB` behaves similarly to the `ippsEncDTXBuffer_GSMAMR` function of the GSM-AMR codec.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrcSpch*, *pSrcIsp*, *pUpdateIndex*, *pSrcDstIspBuffer* or *pSrcDstLogEnergyBuffer* pointer is NULL.
ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.

DecDTXBuffer_AMRWB

Buffers the ISF coefficients and previous logarithm energy coefficients.

Syntax

```
IppStatus ippsDecDTXBuffer_AMRWB_16s(const Ipp16s* pSrcSpch, const Ipp16s*
    pSrcIsf, Ipp16s* pUpdateIndex, Ipp16s* pSrcDstIsfBuffer, Ipp16s*
    pSrcDstLogEnergyBuffer);
```

Parameters

pSrcSpch	Pointer to the input speech signal of length 256.
pSrcIsf	Pointer to the ISF for this frame, of length 16.
pUpdateIndex	Holds the previous memory update index. On output, it contains the current memory update index. Circularly increased between 0 and 7.
pSrcDstIsfBuffer	Pointer to the ISF coefficients of eight previous frames, of length 128. On output, points to the ISF coefficients of eight most recent frames (including the current frame).
pSrcDstLogEnergyBuffer	Pointer to the logarithm energy coefficients of eight previous frames, of length 8. On output, points to the log energy coefficients of eight most recent frames (including the current frame).

Description

The function ippsDecDTXBuffer_AMRWB is declared in ippsc.h file.
This function buffers the ISF coefficients and previous logarithm energy coefficients. The buffered ISF coefficients and energy coefficients will be used for SID frame to extract

necessary parameters.
The memory update index indicates which part of the buffer will be updated and thus saves the cost of some memory copy operation. The logarithm energy is computed as follows:

$$en_{\log}(i) = \frac{1}{2}\log_2\left(\frac{1}{256}\sum_{n=0}^{255}\mathrm{pSrcSpch}^2(n)\right)$$

The function `ippsDecDTXBuffer_AMRWB` behaves similarly to the `ippsDecDTXBuffer_GSMAMR` function of the GSM-AMR codec.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcSpch`*, *`pSrcIsf`*, *`pUpdateIndex`*, *`pSrcDstIsfBuffer`* or *`pSrcDstLogEnergyBuffer`* pointer is `NULL`.

AMR Wideband Plus Related Functions

The Intel IPP functions described in this section if properly combined can be used to construct the Extended Adaptive Multi-Rate Wideband (AMR WB+) Speech Codec compliant to 3rd Generation Partnership Project (3GPP) specification TS 26.273: *"ANSI-C code for fixed-point Extended AMR Wideband (AMRWB+) Speech Codec"*. The description of AMRWB+ codec may be found in 3GPP Ttchnical Specification 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"* [AMRWB+].

The primitives are primarily designed to implement the well-defined, computationally expensive core operations that comprise the codec portion of the AMRWB+ system. The AMRWB+ codec comprises an extended adaptive multi-rate algorithm, intended for encoding 16-bit uniform PCM at the sampling rate of 16, 24, 32 and 48 KHz. The AMRWB+ codec supports AMRWB compatible modes: 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbps for 16 KHz PCM, four special extension stereo modes: 13.6, 18.0, 24.0 kbps and 24.0 kbps for fixed internal sampling frequency only (25600 Hz) and input PCM 16 or 24 kHz with 20ms frame length. Besides, for thirteen internal sampling frequencies: 12800, 14400, 16000, 17067, 19200, 21333, 24000, 25600, 28800, 32000, 34133, 36000, 38400 and 16, 24, 32 and 48 KHz PCM audio AMRWB+ supports 24 extended bitrates, resulted for fixed internal frequency 25600 Hz in eight mono: 10.4, 12.0, 13.6, 15.2, 16.8, 19.2, 20.8 and 24 kbps, and sixteen srereo bitrates: from 2.0 to 8.0 kbps with step of 0.4 kbps. For each internal sampling frequency ISF supported by AMRWB+ codec the correspondent 24 bitrates can be calculated by multiplication to ISF/25600, that is 1/2, 9/16, 5/8, 2/3, 3/4, 5/6, 15/16, 1, 9/8, 5/4, 4/3, 45/32 and 3/2 respectively. For example, for ISF=12800 with multiplier=1/2, the following bitrates are supported: eight mono 5.2, 6.0, 6.8, 7.6, 9.6, 10.4 and 12 kbps, and sixteen stereo bitrates: from 1.0 to 4.0 with step 0.2 kbps.

The list of Intel IPP AMR WB+ functions is given in the following Table 9-6:

Table 9-6 Intel IPP AMR WB Plus Related Functions

Function Base Name	Operation
SNR_AMRWBE	Computes the signal-to-noise ratio
OpenLoopPitchSearch_AMRWBE	Extracts an open-loop pitch lag estimate from the weighted input speech
LPCToISP_AMRWBE	Performs LP to ISP coefficients conversion
SynthesisFilter_AMRWBE	Reconstructs the speech signal from LP coefficients and residuals
Deemphasize_AMRWBE	Performs de-emphasis filtering

Table 9-6 Intel IPP AMR WB Plus Related Functions (continued)

Function Base Name	Operation
FIRGenMidBand_AMRWBE	Computes a shape-constrained FIR filter using the covariance method
PostFilterLowBand_AMRWBE	Post-processes the low-band decoded signal
FFTFwd_RToPerm_AMRWBE	Computes the forward fast Fourier transform (FFT) of a real signal
FFTInv_PermToR_AMRWBE	Computes the inverse fast Fourier transform (FFT) of a real signal
AdaptiveCodebookSearch_AMRWBE	Performs the adaptive codebook search
AdaptiveCodebookDecode_AMRWBE	Decodes the adaptive codebook vector
Downsample_AMRWBE	Performs signal downsampling
Upsample_AMRWBE	Performs signal upsampling
BandSplit_AMRWBE	Gets the vector of energy levels form VAD memory
BandJoin_AMRWBE	Joins the low and high frequency signals
BandSplitDownsample_AMRWBE	Decimates input signal and splits it into high and low frequency components
BandJoinUpsample_AMRWBE	Joins high and low frequency signals and upsamples the result
ResamplePolyphase_AMRWBE	Oversamples or downsamples the input signal to or from the upper frequency (44.1/48 khz)
ISFQuantDecode_AMRWBE	Decodes quantized ISF of HF-band signal.
ISFQuantDecodeHighBand_AMRWBE	Performs ISF quantization of HF-band encoded signal
ISFQuantHighBand_AMRWBE	Performs ISF quantization of HF-band encoded signal
GainQuant_AMRWBE	Quantizes the adaptive codebook gains
QuantTCX_AMRWBE	Quantizes the pre-shaped spectrum in TCX mode
GainQuantTCX_AMRWBE	Performs the gain optimization and quantization
GainDecodeTCX_AMRWBE	Decodes the global TCX gain
EncodeMux_AMRWBE	Encodes and multiplexes subvectors into several packets
DecodeDemux_AMRWBE	Demultiplexes and decodes subvectors from several packets

SNR_AMRWBE

Computes the signal-to-noise ratio.

Syntax

```
IppStatus ippsSNR_AMRWBE_16s(const Ipp16s* pSrcSignal, const Ipp16s*
pSrcEstimatedSignal, int lenSrc, int lenSeg, Ipp16s* pDstSNR);
```

Parameters

pSrcSignal	Pointer to the input signal.
pSrcEstimatedSignal	Pointer to the estimated signal.
lenSrc	Length of the signals.
lenSeg	Length of the segments (subframes).
pDstSNR	Pointer to the signal-to-noise ratio in dB.

Description

The function `ippsSNR_AMRWBE` is declared in `ippsc.h` file. This function computes in decibels (dB) the average segmental signal-to-noise ratio between the *lenSrc* samples of the signal *pSrcSignal* (x) and its estimation *pSrcEstimatedSignal* (e). The segmental signal-to-noise ratio (snr_i) in the i-th segment is defined as follows:

$$snr_i = \log\left(\frac{\sum_{j=1}^{lenSeg} x_j^2}{\sum_{j=1}^{lenSeg} (x_j - e_j)^2}\right)$$

The average segmental signal-to-noise ratio SNR is computed as:

$$SNR = \frac{1}{N_{seg}} \cdot \sum_{i=1}^{N_{seg}} snr_i$$

where N_{seg} is the number of segments.

The segment length is *lenSeg* samples. The results are stored in the *pDstSNR*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.
ippStsSizeErr	Indicates an error when *lenSrc* or *lenSeg* is less than or equal to zero.

OpenLoopPitchSearch_AMRWBE

Extracts an open-loop pitch lag estimate from the weighted input speech.

Syntax

```
IppStatus ippsOpenLoopPitchSearch_AMRWBE_16s (const Ipp16s* pSrcWgtSpch, const
    Ipp16s* pSrcFltWgtSpch, Ipp16s* pPrevMedPitchLag, Ipp16s* pAdaptiveParam,
    Ipp16s* pDstOpenLoopLag, Ipp16s* pToneFlag, Ipp16s* pDstOpenLoopGain,
    Ipp16s* pSrcDstPrevPitchLag, Ipp16s* pSrcDstLagSwitcher, int len, Ipp16s
    minPitchLag, Ipp16s maxPitchLag);
```

Parameters

pSrcWgtSpch	Pointer to a 320-element vector containing perceptually weighted speech.
pSrcFltWgtSpch	Pointer to a 320-element vector containing filtered through high-pass filter perceptually weighted speech.
pPrevMedPitchLag	Pointer to the median filtered pitch lag of the 5 previous voiced speech half-frames.
pAdaptiveParam	Pointer to the adaptive parameter.
pDstOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags.
pToneFlag	Pointer to the tone flag for the VAD module.
pDstOpenLoopGain	Pointer to a 2-element vector containing optimal open-loop pitch gains.
pSrcDstPrevPitchLag	Pointer to the five-element vector that contains the pitch lags associated with the five most recent voiced speech half-frames.

pSrcDstLagSwitcher	Switches lag weighting on and off.
len	Length of the frame.
minPitchLag	Minimum pitch lag.
maxPitchLag	Maximum pitch lag.

Description

This function is declared in `ippsc.h` file. The functionality is the same as for the function ippsOpenLoopPitchSearch_AMRWB. The only difference is that the function `ippsOpenLoopPitchSearch_AMRWBE` has the variable range (*minPitchLag* <= *k* <= *maxPitchLag*) for computation of a windowed auto correlation. With *minPitchLag*=17 and *maxPitchLag*=115 these functions are identical.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the pointers *pSrcWgtSpch*, *pSrcFltWgtSpch*, *pDstOpenLoopLag*, or *pDstOpenLoopGain* is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

LPCToISP_AMRWBE

Performs LP to ISP coefficients conversion.

Syntax

```
IppStatus ippsLPCToISP_AMRWBE_16s(const Ipp16s* pSrcLpc, const Ipp16s*
    pSrcPrevIsp, Ipp16s* pDstIsp, int lpOrder);
```

Parameters

pSrcLpc	Pointer to the input predictor coefficients.
pDstIsp	Pointer to the output immittance spectral pairs.
pSrcPrevIsp	Pointer to the input previous immittance spectral pairs.
lpOrder	Order of conversion.

Description

The `ippsLPCToISP_AMRWBE` function is declared in `ippsc.h` file. This function is the extension of the `ippsLPCToISP_AMRWB` function. If value of the parameter *lpOrder* = 16, these functions are identical.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the pointers *pSrcLpc*, *pDstIsp* or *pSrcPrevIsp* is `NULL`.

Filtering Functions

This section describes functions that perform filtering operations specific to the AMRWB+ Speech Codec.

SynthesisFilter_AMRWBE

Reconstructs the speech signal from LP coefficients and residuals.

Syntax

```
IppStatus ippsSynthesisFilter_AMRWBE_16s32s_I (Ipp16s* pSrcLpc, int order,
    const Ipp16s* pSrcExc, Ipp32s* pSrcDstSignal, int len);
```

Parameters

pSrcLpc	Pointer to the LP coefficients vector a_0, a_1,…,a_{order}
order	Order of LP filter.
pSrcExc	Pointer to the excitation vector.
pSrcDstSignal	Pointer to the synthesized and updated speech.
len	Length of the filter.

Description

The function ippsSynthesisFilter_AMRWBE is declared in ippsc.h file. This function is identical to ippsSynthesisFilter_AMRWB but performs more accurate calculations and allows wider range of input data that is required for the extended codec.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcLpc, pSrcExc or pSrcDstSignal* pointer is NULL.
IppStsSizeErr	Indicates an error when *len or order* is less than or equal to zero.

Deemphasize_AMRWBE

Performs de-emphasis filtering.

Syntax

```
IppStatus ippsDeemphasize_AMRWBE_NR_16s_I(Ipp16s gamma, Ipp32s gammaScale,
    Ipp16s *pSrcDstSignal, int len, Ipp16s* pMem);
```

Parameters

gamma	De-emphasis factor.
gammaScale	Scale factor for the parameter *gamma*.
pSrcDstSignal	Pointer to the source and destination vector.
len	Length of the source and destination vector.
pMem	Pointer to element of the memory for filtering.

Description

The ippsDeemphasize_AMRWBE function is declared in ippsc.h file. This function is the extension of the ippsDeemphasize_AMRWB function. If the value of the parameter *gammaScale* = 15, these functions are identical.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrcDst* or *pMem* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.

FIRGenMidBand_AMRWBE

Computes a shape-constrained FIR filter using the covariance method.

Syntax

```
IppStatus ippsFIRGenMidBand_AMRWBE_16s(const Ipp16s* pSrcSignal, const Ipp16s*
    pSrcSideSignal, Ipp16s* pTaps);
```

Parameters

pSrcSignal	Pointer to the source signal, [-8...319].
pSrcSideSignal	Pointer to the real side signal [320].
pTaps	Pointer to the FIR coefficients of length [9].

Description

The function ippsFIRGenMidBand_AMRWBE is declared in ippsc.h file. This function computes a shape-constrained FIR filter minimizing the expression:

$$\sum_{i=0}^{319}\left(pSrcSideSignal_i - \sum_{j=0}^{8} pTaps_j \cdot pSrcSignal_{i-j}\right)^2$$

The filter coefficients are computed with the covariance method using a modified Cholesky algorithm.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clauses 5.5.2.3 [AMRWB+]

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.

PostFilterLowBand_AMRWBE

Post-processes the low-band decoded signal.

Syntax

```
IppStatus ippsPostFilterLowBand_AMRWBE_16s(const Ipp16s* pSrcOldPitchLag, const
    Ipp16s* pSrcOldPitchGain, Ipp16s* pSrcDstSignal, Ipp16s* pOldSynth, Ipp16s*
    pOldNoise, Ipp16s* pFilterScale, Ipp32s pitchAdjust);
```

Parameters

pSrcOldPitchLag	Pointer to the previous pitch periods for all subframes [16]
pSrcOldPitchGain	Pointer to the previous pitch gains for all subframes [16]
pSrcDstSignal	Pointer to the proceeding signal [1024].
pOldSynth	Pointer to the synthesis memory of post-filter [503].
pOldNoise	Pointer to the noise memory of post-filter [24].
pFilterScale	Pointer to the noise memory of post-filter scale factor.
pitchAdjust	Pitch adjustment flag, if it is `NULL`, then pitch adjustment is not implemented.

Description

The function `ippsPostFilterLowBand_AMRWBE` is declared in `ippsc.h` file. The decoded signal is first processed by an adaptive pitch enhancer, and then filtered through a long-term prediction filter. Then the obtained long term error signal is low-pass filtered and subtracted from the input signal.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clauses 6.1.3 [AMRWB+]

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.

Fast Fourier Transform Functions

This section describes functions that perform fast Fourier trancforms specific to the AMRWB+ Speech Codec.

FFTFwd_RToPerm_AMRWBE

Computes the forward fast Fourier transform (FFT) of a real signal.

Syntax

```
IppStatus ippsFFTFwd_RToPerm_AMRWBE_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len);
```

Parameters

pSrc	Pointer to the real-valued sequence.
pDst	Pointer to the transform result.
len	Length of the sequence, possible values 48, 96, 192, 288, 576 or 1152.

Description

The `ippsFFTFwd_RToPerm_AMRWBE` function is declared in `ippsc.h` file. This function computes N_1 DFTs of size N_2 for real source sequence *pSrc*. Values of N_1 and N_2 are functions of the parameter *len* as follows:

len	48	96	192	288	576	1152
N_1	3	3	3	9	9	9
N_2	16	32	64	32	64	128

The resulting separate transforms are then combined through radix-N_1 transform. The final resulting sequence is stored in the Perm format.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is not equal to 48, 96, 192, 288, 576 or 1152.

FFTInv_PermToR_AMRWBE

Computes the inverse fast Fourier transform (FFT) of a real signal.

Syntax

```
IppStatus ippsFFTInv_PermToR_AMRWBE_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len);
```

Parameters

pSrc	Pointer to the transform coefficients.
pDst	Pointer to the real-valued sequence.
len	Length of the sequence, possible values 48, 96, 192, 288, 576 or 1152.

Description

The `ippsFFTInv_RToPerm_AMRWBE` function is declared in `ippsc.h` file. This function performs inverse FFT transform. Source data are stored in the Perm format. The results are stored as the real-valued sequence.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is not equal to 48, 96, 192, 288, 576 or 1152.

Codebook Search Functions

This section describes functions that perform codebook searching specific to the AMRWB+ Speech Codec.

AdaptiveCodebookSearch_AMRWBE

Performs the adaptive codebook search.

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_AMRWBE_16s(const Ipp16s*  pSrcAdptTarget,
    const Ipp16s*  pSrcImpulseResponse, const Ipp16s*  pSrcOpenLoopLag, Ipp16s*
    pPitchLag, Ipp16s*  pPitchLagBounds, Ipp16s*  pSrcDstExcitation, Ipp16s*
    pFracPitchLag, Ipp16s*  pAdptIndex, int   subFrame, IppSpchBitRate mode,
    Ipp16s pitchOffset);
```

Parameters

pSrcAdptTarget	Pointer to the 64-element adaptive target signal vector.
pSrcImpulseResponse	Pointer to the 64-element impulse response of the weighted synthesis filter.
pSrcOpenLoopLag	Pointer to a two-element vector of the open-loop pitch lags.
pPitchLag	Pointer to the output previous integral pitch lag.
pPitchLagBounds	Pointer to the bounds of the output previous integral pitch lag.
pSrcDstExcitation	Pointer to the input/output 321-element excitation vector.
pFracPitchLag	Pointer to the output fractional pitch lag obtained during the adaptive codebook search.
pAdptIndex	Pointer to the coded closed-loop pitch index.
subFrame	Subframe number.
mode	Coder mode.
pitchOffset	Offset for the pitch adjustment, value range (-17, 17).

Description

The `ippsAdaptiveCodebookSearch_AMRWBE` function is declared in `ippsc.h` file. This function is the extension of the function `ippsAdaptiveCodebookSearch_AMRWB`. The *pitchOffset* parameter controls the adjustment of the search ranges. If value of *pitchOffset* = 0, both functions are identical.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *pitchOffset* is out of the range (-17, 17), or when *mode* has an illegal value.

AdaptiveCodebookDecode_AMRWBE

Decodes the adaptive codebook vector.

Syntax

```
IppStatus ippsAdaptiveCodebookDecode_AMRWBE_16s(Ipp32s valAdptIndex, Ipp16s*
    pSrcDstExcitation, Ipp16s* pSrcDstPitchLag, Ipp16s* pSrcDstFracPitchLag,
    Ipp16s* pSrcDstPitchLagBounds, int subFrame, Ipp32s bfi, ipp32s
    pitchOffset);
```

Parameters

valAdptIndex	Adaptive codebook index.
pSrcDstExcitation	Pointer to the 321-element excitation vector.
pSrcDstPitchLag	Pointer to the integral pitch lag.
pSrcDstFracPitchLag	Pointer to the fractional pitch lag obtained during the adaptive search.
pPitchLagBounds	Pointer to the integral pitch lag bounds (two-element vector).
subFrame	Subframe number.
bfi	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
pitchOffset	Offset for the pitch adjustment, value range (-17, 17).

Description

The ippsAdaptiveCodebookDecode_AMRWBE function is declared in ippsc.h file. This function is similar to the function ippsAdaptiveCodebookDecode_AMRWB but has additional parameter *pitchOffset* that controls the adjustment of the search ranges. If value of *pitchOffset* = 0, both functions are identical.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.
ippStsSizeErr	Indicates an error when *pitchOffset* is out of the range (-17, 17), or when *mode* has an illegal value.

Resample Functions

This section describes functions that perform different signal resampling operations specific to the AMRWB+ Speech Codec.

Downsample_AMRWBE

Performs the signal decimating.

Syntax

```
IppStatus ippsDownsample_AMRWBE_16s(const Ipp16s* pSrcSignal, int lenSrc,
    Ipp16s* pDstSignal, Ipp16s* pMem, Ipp32s bandIdx);
```

Parameters

pSrcSignal	Pointer to the source signal.
lenSrc	Length of the source vector, [640] for 8 kHz source signal, [1280] for 16 kHz and [1920] for 24 kHz.
pDstSignal	Pointer to destination signal [1024].
pMem	Pointer to the memory for descimating [46].
bandIdx	Index of interpolating band; possible values: 0 for 0..6.4 kHz band, any other for 6.4..10.8 kHz band.

Description

The function `ippsDownsample_AMRWBE` is declared in `ippsc.h` file. This function performs downsampling of the source signal *pSrcSignal* with the following sample rate: 8, 16, 24 kHz. The function filters input signal to the band 0...6.4 kHz, or 6.4...10.8 kHz in accordacne with the parameter *bandIdx*, and downsamples (upsamples for 8 kHz input signal) to 12.6 kHz.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *lenDst* has an illegal value.

Upsample_AMRWBE

Performs signal oversampling.

Syntax

```
IppStatus ippsUpsample_AMRWBE_16s(const Ipp16s* pSrcSignal, Ipp16s*
    pSrcDstSignal, int lenDst, Ipp16s* pMem, Ipp32s bandIdx, Ipp32s addFlag);
```

Parameters

pSrcSignal	Pointer to source signal signal [1024].
pSrcDstSignal	Pointer to the destination oversampled signal.
lenDst	Length of destination vector, [640] for 8kHz output signal, [1280] for 16kHz and [1920] for 24kHz.
pMem	Pointer to the memory for oversampling [24].
bandIdx	Index of interpolating band; possible values: 0 for 0..6.4k band, any other value for 6.4..10.8k band.
addFlag	Flag for adding operation; if it equals to 1 then the result is added to the output vector, if it equals to 0 - the result is not added.

Description

The function `ippsUpsample_AMRWBE` is declared in `ippsc.h` file. This function performs the signal oversampling of the source vector *pSrcSignal*. This operation is inverse of the operation described in the decsription of the function `ippsDownsample_AMRWBE`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.
ippStsSizeErr	Indicates an error when *lenDst* has an illegal value.

BandSplit_AMRWBE

Splits the signal into low and high frequency components.

Syntax

```
IppStatus ippsBandSplit_AMRWBE_16s(const Ipp16s* pSrcSignal, Ipp16s*
    pSrcDstSig2k, Ipp16s* pDstSigHi, int len);
```

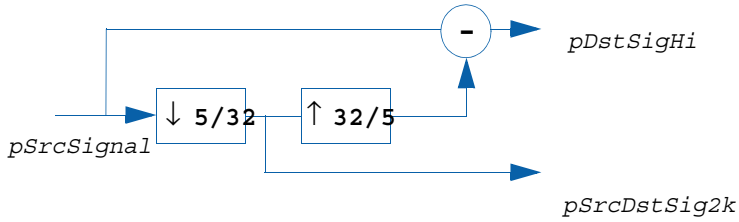
Parameters

pSrcSignal	Pointer to the source vector [*len* + 2*64].
pSrcDstSig2k	Pointer to the source and destination low frequency vector [*len*5/32 + 2*10].
pDstSigHi	Pointer to the destination high frequency vector [*len*].
len	Length of the sample.

Description

The function `ippsBandSplit_AMRWBE` is declared in `ippsc.h` file. This function splits the input signal (*f* kHz) into two bands: 0...5/32**f* kHz band, and 5/32**f* ... *f* kHz band. The low frequency band is critically down-sampled and the side signal is computed according to the diagram on the Figure 9-1.

Figure 9-1 Band Split Computing



See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clauses 5.1.2 [AMRWB+]

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

BandJoin_AMRWBE

Joins the low and high frequency signals.

Syntax

```
IppStatus ippsBandJoin_AMRWBE_16s(const Ipp16s* pSrcSig2k, const Ipp16s*
    pSrcSigHi, Ipp16s* pDstSignal, int len);
```

Parameters

pSrcSig2k	Pointer to the source low frequency vector [*len**5/32 + 2*10].

pSrcSigHi	Pointer to the source high frequency vector [*len*].
pDstSignal	Pointer to the destination vector [*len*].
len	Length of the sample.

Description

The function ippsBandJoin_AMRWBE is declared in ippsc.h file. This function performs operation that is the inverse of the encoder band-splitting operation described in the decsription of the function ippsBandSplit_AMRWBE, that is the source low frequency signal *pSrcSig2k* is oversampled with factor 32/5 and then combined with the source high frequency signal *pSrcSigHi*. The result is store in the vector *pDstSignal*.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clauses 6.3.4 [AMRWB+]

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.

BandSplitDownsample_AMRWBE

Downsamples input signal and splits it into high and low frequency components.

Syntax

```
IppStatus ippsBandSplitDownsample_AMRWBE_16s(const Ipp16s* pSrcSig, int lenSrc,
    Ipp16s* pDstSigLF, Ipp16s* pDstSigHF, int lenDst, Ipp16s* pMem, Ipp16s*
    pInterFracMem, Ipp32s* pCountSamp, Ipp16s resampleFactor);
```

Parameters

pSrcSig	Pointer to the source signal.
lenSrc	Length of the source signal, [7680] for 48 kHz signal, or [7056] for 44.1 kHz signal.
pDstSigLF	Pointer to the destination low frequency component.
pDstSigHF	Pointer to the destination high frequency component.

lenDst	Length of destination vectors.
pMem	Pointer to the memory, [1608] elements.
pInterFracMem	Pointer to the memory for the length of interpolating fraction.
pCountSamp	Number of decimated samples.
resampleFactor	Resampling frequency scale factor.

Description

The function `ippsBandSplitDownsample_AMRWBE` is declared in `ippsc.h` file. This function downsamples input signal to the desired internal sampling frequency *f* of the encoder and splits it into high and low frequency components. Firstly the input signal *pSrcSig* is upsampled by the factor *resampleFactor*, filtered by a low pass filter, and then downsampled by the factor 180. The value of the *resampleFactor* factor depends on the desired internal sampling frequency *f* (see Table 9-7). The low frequency component is obtained by low-pass filtering downsampled signal to *f*/4 kHz, and critically downsampling result to *f*/2 kHz. The high frequency component is obtained by band-pass filtering to frequencies above *f*/4 kHz, and critically downsampling result to *f*/2 kHz.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clauses 5.1 [AMRWB+].

Table 9-7 Dependence between Values of the parameter *resampleFactor* and the Internal Sampling Frequency

Internal sampling frequency *f*, Hz	Factor for the internal frequency	Value of the parameter *resampleFactor*
12800	1/2	48
14400	9/16	54
16000	5/8	60
17067	2/3	64
19200	3/4	72
21333	5/6	80
24000	15/16	90
25600	1	96
28800	9/8	108
32000	5/4	120
34133	4/3	128
26000	45/32	135
38400	3/2	144

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.
ippStsSizeErr	Indicates an error when *lenSrc* or *lenDst* is less than or equal to zero.

BandJoinUpsample_AMRWBE

Joins high and low frequency signals and upsamples the result.

Syntax

```
IppStatus ippsBandJoinUpsample_AMRWBE_16s(const Ipp16s* pSrcSigLF, const
    Ipp16s* pSrcSigHF, int lenSrc, Ipp16s* pDstSig, int lenDst, Ipp16s* pMem,
    Ipp16s* pInterFracMem, Ipp32s* pCountSamp, Ipp16s resampleFactor);
```

Parameters

pSrcSigLF	Pointer to the low frequency signal.
pSrcSigHF	Pointer to the high frequency signal.
lenSrc	Length of low frequency and high frequency vectors.
pDstSig	Pointer to the upsampled signal.
lenDst	Length of the destination vector.
pMem	Pointer to the memory for resampling [72].
pInterFracMem	Pointer to the memory for interpolating fraction.
pCountSamp	Pointer to the number of oversampled samples.
resampleFactor	Downsampling frequency scale factor.

Description

The function ippsBandJoinUpsample_AMRWBE is declared in ippsc.h file. This function is used by the decoder: it combines the high and low frequency signals and upsamples the result to produce the full band output signal. This operation is inverse of the encoder band-splitting operation performed by the function ippsBandSplitDownsample_AMRWBE.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clauses 6.6 [AMRWB+]

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.
ippStsSizeErr	Indicates an error when *lenSrc* or *lenDst* is less than or equal to zero.

ResamplePolyphase_AMRWBE

Upsamples or downsamples the input signal to or from the upper frequency (44.1/48 khz).

Syntax

```
IppStatus ippsResamplePolyphase_AMRWBE_16s(const Ipp16s* pSrcSignal, int
    lenSrc,  Ipp16s upFactor, Ipp16s downFactor, Ipp16s* pInterpFracMem, Ipp16s*
    pMem, Ipp16s* pDstSignal, int lenDst);
```

Parameters

pSrcSignal	Pointer to the source signal.
lenSrc	Length of the input vector.
upFactor	Upsampling factor.
downFactor	Downsampling factor.
pInterpFracMem	Pointer to the memory for interpolating fraction.
pMem	Pointer to the memory for resampling, [144] for downsampling, or [44] for upsampling.
pDstSignal	Pointer to the destination resampled signal.
lenDst	Length of the destination vector.

Description

The function ippsResamplePolyphase_AMRWBE is declared in ippsc.h file. This function performs resampling of the input signal in accordance with the values of the resampling factors *upFactor* (upsampling factor) and *downFactor* (downsampling factor). If *upFactor* < *downFactor*, the function downsamples the input 44.1/48 kHz signal, if *upFactor* > *downFactor*, the function upsamples the inpuit signal to the 44.1/48 kHz signal. The possible combinations of the input/output frequencies and corresponding values of resampling factors are listed in the Table 9-8 and Table 9-9.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clauses 5.1, 6.6 [AMRWB+].

Table 9-8 Upsampling Mode

Input frequency, kHz	Upsampling factor	Downsampling factor	Output frequency, kHz
8	12	2	48
16	12	4	48
24	12	6	48
32	12	8	48
11.025	12	3	44.1
22.05	12	6	44.1

Table 9-9 Downsampling Mode

Input frequency, kHz	Upsampling factor	Downsampling factor	Output frequency, kHz
48	2	12	8
48	4	12	16
48	6	12	24
48	8	12	32
44.1	3	12	11.025
44.1	6	12	22.05

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`lenDst`* has an illegal value.

Quantization Functions

This section describes functions that perform quantization specific to the AMRWB+ Speech Codec.

ISFQuantDecode_AMRWBE

Decodes quantized ISFs from the received codebook index.

Syntax

```
IppStatus ippsISFQuantDecode_AMRWBE_16s(const Ipp16s* pSrcIdxs, Ipp16s*
    pDstQntIsf, Ipp16s* pSrcDstResidual, const Ipp16s* pSrcPrevQntIsf, Ipp16s*
    pSrcDstIsfMemory, Ipp32s bfi, Ipp32s splitMask);
```

Parameters

pSrcIdxs	Pointer to the seven-element vector containing codebook indices of the quantized LSPs.
pDstQntIsf	Pointer to a 16-element destination vector containing quantized ISF in cosine domain.
pSrcDstResidual	Pointer to the 16-element quantized ISF residual from the previous frame.
pSrcPrevQntIsf	Pointer to the 16-element quantized ISF vector from the previous frame.
pSrcDstIsfMemory	Pointer to the 64-element vector containing four subframe ISF sets.
bfi	Bad frame indicator: value "0" indicates a good frame, all other values indicate a bad frame.
splitMask	Binary mask for the second-stage splitting. It contains 5 significant bits, low significant bit corresponds to the low number of splitting.

Description

This function is declared in `ippsc.h` file. The function decodes quantized ISFs from the received codebook index. This function is extension of the function `ippsISFQuantDecode_AMRWB`. If the parameter *splitMask* is set to 0, both functions are identical. Otherwise the function performs the second-level splitting in accordance with the *splitMask*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.

ISFQuantDecodeHighBand_AMRWBE

Decodes quantized ISF of HF-band signal.

Syntax

```
IppStatus ippsISFQuantDecodeHighBand_AMRWBE_16s(const Ipp16s* pSrcQuantIdxs,
    Ipp16s* pSrcDstPastQISF, Ipp16s* pDstQISF, Ipp32s bfi, Ipp32s pitchAdjust);
```

Parameters

pSrcQuantIdxs	Pointer to the quantization indices [2].
pSrcDstPastQISF	Pointer to the past quantized ISF [8].
pDstQISF	Pointer to the output vector of decoded quantized ISF [8].
bfi	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
pitchAdjust	Controls pitch adjustment: if it equals to 0, then the function does not perform pitch adjustment.

Description

The function ippsISFQuantDecodeHighBand_AMRWBE is declared in ippsc.h file. This function decodes quantized ISF of HF-band signal from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized ISFs from previous quantized ISFs using linear interpolation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.

ISFQuantHighBand_AMRWBE

Performs ISF quantization of HF-band encoded signal.

Syntax

```
IppStatus ippsISFQuantHighBand_AMRWBE_16s(const Ipp16s* pSrcISF, Ipp16s*
    pSrcDstPastQISF, Ipp16s* pDstQISF, Ipp16s* pQuandIdxs, Ipp32s pitchAdjust);
```

Parameters

pSrcISF	Pointer to the source vector of proceeding ISF [8].
pSrcDstPastQISF	Pointer to the past quantized ISF [8].
pDstQISF	Pointer to the output vector of quantized ISF [8].
pQuantIdxs	Pointer to the output quantization indices [2].
pitchAdjust	Controls pitch adjustment: if it equals to 0, then the function does not perform pitch adjustment.

Description

The function `ippsISFQuantHighBand_AMRWBE` is declared in `ippsc.h` file. This function performs ISF quantization of HF-band encoded signal using one-stage VQ with 8 elements.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clauses 5.4 [AMRWB+].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.

GainQuant_AMRWBE

Quantizes the adaptive codebook gains.

Syntax

```
IppStatus ippsGainQuant_AMRWBE_16s(Ipp16s* pSrcAdptTarget, const Ipp16s *
    pSrcFltAdptVector, int valFormat, const Ipp16s* pSrcFixedVector, const
    Ipp16s* pSrcFltFixedVector, int lenSrc, const Ipp16s* pSrcCorr, Ipp16s
    meanEnergy, Ipp16s* pSrcDstPitchGain, Ipp16s* pDstCorrFactor, Ipp32s*
    pDstCodeGain, Ipp16s* pDstQGainIndex);
```

Parameters

pSrcAdptTarget	Pointer to the input target vector x(n).
pSrcFltAdptVector	Pointer to the input filtered adaptive codebook vector y(n).
valFormat	Format of the *pSrcAdptTarget* and *pSrcFltAdptVector* vectors.
pSrcFixedVector	Pointer to the input pre-filtered codebook contribution c(n).
pSrcFltFixedVector	Pointer to the input filtered codebook vector z(n).
lenSrc	Length of the input vectors (n).
pSrcCorr	Pointer to the vector of correlations between the *pSrcAdptTarget*, *pSrcFltAdptVector*, *pSrcFltFixedVector* vectors.
meanEnergy	Average energy in the whole frame.
pSrcDstPitchGain	Pointer to the input/output pitch gain.
pDstCorrFactor	Pointer to the correction factor.
pDstCodeGain	Pointer to the output code gain.
pDstQGainIndex	Pointer to the output codebook indexes found.

Description

The function `ippsGainQuant_AMRWBE` is declared in `ippsc.h` file. This function quantizes the adaptive codebook gain (pitch gain) and the fixed (algebraic) codebook gain using the same 7-bit codebook that is used in the function `ippsGainQuant_AMRWB` for modes from 12.6 to 23.8 kbit/s. However, instead of using "moving average" prediction to obtain the predicted gain g_C, it is found directly by quantizing the average energy *meanEnergy* in the whole frame.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.
ippStsSizeErr	Indicates an error when *lenSrc* is less than or equal to zero.

QuantTCX_AMRWBE

Quantizes the pre-shaped spectrum in TCX mode.

Syntax

```
IppStatus ippsQuantTCX_AMRWBE_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    nSubvectors, int nBits, Ipp16s* pDstNoiseFactor);
```

Parameters

pSrc	Pointer to the source signal [*nSubvectors**8].
pDst	Pointer to the quantized normalized vector[*nSubvectors**8 + *nSubvectors*].
nSubvectors	Number of subvectors.
nBits	Number of bits to use.
pDstNoiseFactor	Pointer to the comfort noise gain factor.

Description

The function ippsQuantTCX_AMRWBE is declared in ippsc.h file. This function quantizes the pre-shaped spectrum in TCX mode.This function quantizes the pre-shaped spectrum in TCX mode using lattice quantizers method. The spectrum is quantized in 8-dimensional blocks using vector codebooks composed of subsets of the Gosset lattice. In lattice quantization, the procedure of finding the nearest neighbour of an input vector among all codebook points is reduced to a several simple operations - rounding the components of the vector and verifying a few constraints with no exhaustive search. The binary indexes are computed only if a given TCX mode is retained as the best mode for a frame. It allows to reduce computation complexity.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clause 5.3.5.7 [AMRWB+].

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.

GainQuantTCX_AMRWBE

Performs the gain optimization and quantization.

Syntax

```
IppStatus ippsGainQuantTCX_AMRWBE_16s(const Ipp16s* pSrcSignal, Ipp16s
    srcScale, const Ipp16s* pSrcQuantSignal, int len, Ipp32s quantFlag, Ipp32s*
    pGain, Ipp16s* pQuantIdx);
```

Parameters

pSrcSignal	Pointer to the original weighted signal.
srcScale	Format of input signal.
pSrcQuantSignal	Pointer to the quantized weighted signal.
len	Length of the source vector.
quantFlag	If equals to 0, then the function computes and returns the optimal gain only, else computes and returns the quantized gain and quantization index.
pGain	Pointer to the output quantized or optimal gain.
pQuantIdx	Pointer to the output index of quantization.

Description

The function `ippsGainQuantTCX_AMRWBE` is declared in `ippsc.h` file. This function performs the gain optimization and quantization to maximize the correlation between the original weighted signal *pSrcSignal* (x) and the quantized weighted signal *pSrcQuantSignal* ($\hat{x}$). The optimal gain g^* between x and $\hat{x}$ is computed in accordance with the following formula:

$$g^* = \frac{\sum_{i=0}^{len-1} x_i \cdot \hat{x}_i}{\sum_{i=0}^{len-1} \hat{x}_i \cdot \hat{x}_i}$$

The gain g^* is quantized on the logarithmic scale to a 7-bit index in the following way:

- compute the energy of the quantized weighted signal $E = \sum_{i=0}^{len-1} \hat{x}_i^2$;
- compute the RMS value $rms = 4 \cdot \sqrt{\frac{E}{len}}$;
- compute the index $index = 28 \cdot \log(g^* \cdot rms) + 0.5$;
- if $index < 0$, then set $index = 0$; if $index > 127$, then set $index = 127$;
- and the quantized gain $\hat{g}^* = 10^{\frac{index}{28 \cdot rms}}$

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clause 5.3.5.10 [AMRWB+].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

GainDecodeTCX_AMRWBE

Decodes the global TCX gain.

Syntax

```
IppStatus ippsGainDecodeTCX_AMRWBE_16s(const Ipp16s* pSrcQuantSignal, int len,
    Ipp16s quantIdx, Ipp32s bfi, Ipp16s* pSrcDstRMSval, Ipp32s* pGain);
```

Parameters

pSrcQuantSignal	Pointer to the quantized signal.
len	Length of the source signal.
quantIdx	Index of quantization.
bfi	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
pSrcDstRMSval	Pointer to the root mean square value.
pGain	Pointer to the output global TCX gain.

Description

The function `ippsGainDecodeTCX_AMRWBE` is declared in `ippsc.h` file. This function decodes the global TCX gain g_{TCX} by inverting the 7-bit logarithmic quantization calculated in the TCX encoder as in the function ippsGainQuantTCX_AMRWBE.

First the RMS value of the TCX target signal *pSrcQuantSignal* ($\hat{x}$) is computed as

$$rms = \sqrt{\frac{\sum_{i=0}^{len-1} \hat{x}_i^2}{len}}$$

From the received 7-bit index $0 \le quantIdx \le 127$, the TCX gain *pGain* can be computed

$$pGain = 10^{\frac{quantIdx}{28 \cdot 4 \cdot rms}}$$

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clause 6.1.2 [AMRWB+].

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.

EncodeMux_AMRWBE

Encodes and multiplexes subvectors into several packets.

Syntax

```
IppStatus ippsEncodeMux_AMRWBE_16s(const Ipp16s* pSrc, int nSubvectors, const
    int* pPacketSizes, Ipp16s* pDstParams, int nPackets);
```

Parameters

pSrc	Pointer to the rounded subvectors [*nSubvectors**8 + *nSubvectors*].
nSubvectors	Number of subvectors.
pPacketSizes	Pointer to the vector of size of each packet [*nPackets*].
pDstParams	Multiplexed parameters [(*pPacketSize*[0]+3)/4 +...+ (*pPacketSize*[*nPackets*]+3)/4].
nPackets	Number of packets.

Description

The function ippsEncodeMux_AMRWBE is declared in ippsc.h file. This function encodes the TCX parameters and puts them in one or several binary packets for transmission.

See also 3GPP TS 26.290: *"Extended AMR Wideband Speech Codec; Transcoding functions"*, clause 5.6 [AMRWB+].

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.

DecodeDemux_AMRWBE

Demultiplexes and decodes subvectors from several packets.

Syntax

```
IppStatus ippsDecodeDemux_AMRWBE_16s(const Ipp16s* pSrcParams, const Ipp32s*
    pPacketSizes, const Ipp32s* pBFI, int nPackets, Ipp16s* pDst, int
    nSubvectors);
```

Parameters

pSrcParams	Demultiplexed parameters [(*pPacketSizes*[0]+3)/4 +...+ (*pPacketSize*[*nPackets*]+3)/4].
pPacketSizes	Pointer to the vector of size of each packet [*nPackets*].
pBFI	Pointer to the vector of bad frame indicator for each packet [*nPackets*].
nPackets	Number of packets.
pDst	Pointer to the rounded subvectors [*nSubvectors**8].
nSubvectors	Number of subvectors.

Description

The function ippsDecodeDemux_AMRWBE is declared in ippsc.h file. This function demultiplexes and decodes subvectors from several packets, that is performs inverse operation to the encoding procedure performed by the function ippsEncodeMux_AMRWBE.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointer is NULL.

GSM Full Rate Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following the GSM 06.10, 06.11, 06.12, 06.31, and 06.32 recommendations.
The list of these functions is given in the following table:

Table 9-10 Intel IPP GSM Full Rate Related Functions

Function Base Name	Operation
RPEQuantDecode_GSMFR	Performs APCM inverse quantization.
Deemphasize_GSMFR	Performs de-emphasis filtering.
ShortTermAnalysisFilter_GSMFR	Performs short-term analysis filtering.
ShortTermSynthesisFilter_GSMFR	Performs short-term synthesis filtering.
HighPassFilter_GSMFR	Filters input speech signal through a high-pass filter.
Schur_GSMFR	Estimates the reflection coefficients by Schur recursion.
WeightingFilter_GSMFR	Calculates the weighting filter.
Preemphasize_GSMFR	Computes pre-emphasis of a speech signal.

RPEQuantDecode_GSMFR

Performs APCM inverse quantization.

Syntax

```
IppStatus ippsRPEQuantDecode_GSMFR_16s (const Ipp16s* pSrc, Ipp16s ampl, Ipp16s
    amplSfs, Ipp16s* pDst);
```

Parameters

pSrc	Pointer to the input vector [13] of the RPE samples.
ampl	The block amplitude.
amplSfs	Scale factor of the block amplitude.
pDst	Pointer to the output reconstructed long-term residual vector [13].

Description

The function `ippsRPEQuantDecode_GSMFR` is declared in `ippsc.h` file. This function performs APCM inverse quantization of the input RPE samples. The output reconstructed long-term residual vector is formed as

$$pDst[i] = (2 \cdot pSrc[i] - 7) \cdot \frac{amp}{2^{amplSfs}}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsScaleRangeErr`	Indicates an error when *amplSfs* is less then `0`.

Deemphasize_GSMFR

Performs de-emphasis filtering.

Syntax

```
IppStatus ippsDeemphasize_GSMFR_16s_I (Ipp16s* pSrcDst, int len, Ipp16s* pMem);
```

Parameters

pSrcDst	Pointer to the input short-term synthesized signal and output post-processed speech vector [*len*].
len	Length of the input residual and output speech vectors.
pMem	Pointer to the filter memory element.

Description

The function `ippsDeemphasize_GSMFR` is declared in `ippsc.h` file. This function performs de-emphasis of the input synthesized signal by filtering it through the filter with the following transfer function:

$$H(z) = \frac{1}{1 - \alpha z^{-1}}$$

where α = 0.86 (28180 in Q15).

The initial memory of the filter will be set to zero.

The filtered speech signal is scaled up by multiple of 2 and then stored in *pSrcDst* with truncation of the three least significant bits.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* or *pMem* pointer is NULL.
ippStsRangeErr	Indicates an error when *len* is less or equal to 0.

ShortTermAnalysisFilter_GSMFR

Performs short-term analysis filtering.

Syntax

```
IppStatus ippsShortTermAnalysisFilter_GSMFR_16s_I (const Ipp16s* pRC, Ipp16s*
    pSrcDstSpch, int len, Ipp16s* pMem);
```

Parameters

pRC	Pointer to the input reflection coefficients vector [8]: $a_1, a_2, \ldots, a_8$.
pSrcDstSpch	Pointer to the input pre-processed speech and output short term residual vector [*len*].
len	Length of the input speech and output residual vectors.
pMem	Pointer to the filter memory vector [8]: $m_0, m_1, \ldots, m_7$.

Description

The function ippsShortTermAnalysisFilter_GSMFR is declared in ippsc.h file. This function performs filtering of the input pre-processed speech vector $s(n)$ and stores the result in the output short-term residual vector $r(n)$ as given below:

$$r_0 = s(n)$$

$$r_i = r_{i-1} + a_i * m_{i-1}, \quad i = 1,\ldots,8$$

$$m_0 = s(n)$$

$$m_i = m_{i-1} + a_i * r_{i-1}, \quad i = 1,\ldots,7$$

$$r(n) = r_8$$

where m_i , i = 0,...,7 is the filter memory and r_i , i = 0,...,8 is the reusable local memory.

The initial filter memory vector will be zeroed.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRC*, *pSrcDstSpch* or *pMem* pointer is NULL.
ippStsRangeErr	Indicates an error when *len* is less or equal to 0.

ShortTermSynthesisFilter_GSMFR

Performs short-term synthesis filtering.

Syntax

```
IppStatus ippsShortTermSynthesisFilter_GSMFR_16s (const Ipp16s* pRC, const
    Ipp16s* pSrcResidual, Ipp16s* pDstSpch, int len, Ipp16s* pMem);
```

Parameters

pRC	Pointer to the input reflection coefficients vector [8]: a_1, a_2 ,..., a_8 .
pSrcResidual	Pointer to the input reconstructed short term residual vector [*len*].
pDstSpch	Pointer to the output speech vector [*len*].
len	Length of the input residual and output speech vectors.
pMem	Pointer to the filter memory vector [8]: m_0, m_1,..., m_7 .

Description

The function ippsShortTermSynthesisFilter_GSMFR is declared in ippsc.h file. This function performs filtering of the input reconstructed short-term residual $r(n)$ and stores the result in the output speech vector $s(n)$ as given below:

$$s_0 = r(n)$$

$$s_i = s_{i-1} - a_{9-i} * m_{8-i} , \qquad i = 1,...,8$$

$$m_{8-i} = m_{7-i} + - a_{9-i} * s_i , \quad i = 1,...,7$$

$s(n) = s_8$

$m_0 = s(n)$

where m_i , i = 0,...,7 is the filter memory and s_i , i = 0,...,8 is the reusable local memory.

The initial filter memory vector will be zeroed.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRC*, *pSrcResidual*, *pMem*, or *pDstSpch* pointer is NULL.
ippStsRangeErr	Indicates an error when *len* is less or equal to 0.

HighPassFilter_GSMFR

Performs high-pass filtering of the input speech signal.

Syntax

```
IppStatus ippsHighPassFilter_GSMFR_16s (const Ipp16s* pSrc, Ipp16s* pDst,
    int len, int* pMem);
```

Parameters

pSrc	Pointer to the source speech vector [*len*].
pDst	Pointer to the destination filtered vector [*len*].
len	Length of the source and destination vectors.
pMem	Pointer to the filter memory vector [2].

Description

The function ippsHighPassFilter_GSMFR is declared in ippsc.h file. This function filters the input speech signal according to the transfer function:

$$H(z) = 0.5 \cdot \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

where α = 0.99899.

The initial filter memory will be set to zero.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pMem* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.

Schur_GSMFR

Estimates the reflection coefficients by Schur recursion.

Syntax

```
IppStatus ippsSchur_GSMFR_32s16s (const Ipp32s* pSrc, Ipp16s* pDst,
    int dstLen);
```

Parameters

pSrc	Pointer to the input autocorrelation vector [*dstLen*+1].
pDst	Pointer to the output reflection coefficients vector [*dstLen*].
dstLen	The number of reflection coefficients to estimate.

Description

The function ippsSchur_GSMFR is declared in ippsc.h file. This function implements the Schur algorithm according to GSM 06.10 clause 4.2.5. See also ippsSchur function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *dstLen* is less or equal to 0, or *dstLen* is less than 9.

WeightingFilter_GSMFR

Calculates the weighting filter.

Syntax

```
IppStatus ippsWeightingFilter_GSMFR_16s (const Ipp16s* pSrc, Ipp16s* pDst,
    int dstLen);
```

Parameters

pSrc	Pointer to the input long-term residual vector [-5,..,*dstLen*+4].
pDst	Pointer to the filtered output vector [*dstLen*].
dstLen	The number of filtered elements to calculate.

Description

The function `ippsWeightingFilter_GSMFR` is declared in `ippsc.h` file. This function performs filtering of the input signal by symmetric FIR filter with predefined taps given below:

taps[i]=[-134, -374, 0, 2054, 5741, 8192, 5741, 2054, 0, -374, -134], i=0,..,10

The filtering is performing according to formula:

$$dst[n] = \sum_{i=0}^{10} taps[i] \cdot src[n+i-5], \quad n = 0, \ldots, dstLen-1$$

The result of filtering is stored in *pDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *dstLen* is less or equal to `0`.

Preemphasize_GSMFR

Computes pre-emphasis of a speech signal.

Syntax

```
IppStatus ippsPreemphasize_GSMFR_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int* pMem, int len);
```

Parameters

pSrc	Pointer to the offset-free input speech signal.
pDst	Pointer to the pre-emphasized output signal.
pMem	Pointer to the filter memory value.
len	Length of the input and output signals.

Description

The function `ippsPreemphasize_GSMFR` is declared in `ippsc.h` file. This function computes pre-emphasis of the input speech according to the difference signal pre-emphasis equation:

$$H(z) = 1 - \gamma z^{-1}$$

for γ = -0.86 and *pSrc*[-1] = *pMem*[0].

The result of filtering is stored in *pDst*. The memory value *pMem*[0] is updated by *pSrc*[n-1]. For proper use of this function in GSM Full Rate codec, the memory value will be initialized to zero.

The function `ippsPreemphasize_GSMFR` performs NR rounding (see Rounding mode).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pMem* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less or equal to `0`.

G.722.1 Related Functions

This chapter describes Intel IPP functions that can be used in implementing speech codecs following ITU-T recommendations G.722.1.
The list of these functions is given in the following table:

Table 9-11 Intel IPP G.722.1 Related Functions

Function Base Name	Operation
DCTFwd_G722, DCTInv_G722	Computes the forward or inverse discrete cosine transform (DCT) of a signal.
DecomposeMLTToDCT	Decomposes the MLT transform input signal to the form of the DCT input signal.
DecomposeDCTToMLT	Decomposes IDCT output signal to the form of the MLT transform output signal.
HuffmanEncode_G722	Performs Huffman encoding of the quantized amplitude envelope indexes.

These functions are used in the Intel IPP *G.722.1 Speech Encoder-Decoder* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

DCTFwd_G722, DCTInv_G722

Computes the forward or inverse discrete cosine transform (DCT) of a signal.

Syntax

```
IppStatus ippsDCTFwd_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst);
IppStatus ippsDCTInv_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst);
```

Parameters

pSrc	Pointer to the source vector[320].
pDst	Pointer to the destination vector[320]

Description

The functions `ippsDCTFwd_G722` and `ippsDCTInv_G722` are declared in `ippsc.h` file. These functions compute the forward and inverse discrete cosine transform (DCT) of length 320 as follows:

$$y(m) = \alpha \sum_{n=0}^{319} \cos\left(\frac{\pi}{320} \cdot (n+0.5) \cdot (m+0.5)\right) \cdot x(n) \quad , \text{ for forward DCT}$$

where α = 2/320;

$$x(n) = \beta \sum_{m=0}^{319} \cos\left(\frac{\pi}{320} \cdot (m+0.5) \cdot (n+0.5)\right) \cdot y(m) \quad , \text{ for inverse DCT (IDCT) .}$$

where β = 0.81.

The formulae are the same except for the different scaling that is applied to meet the bit exact requirement.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.

DecomposeMLTToDCT

Decomposes the MLT transform input signal to the form of the DCT input signal.

Syntax

```
IppStatus ippsDecomposeMLTToDCT_G722_16s(const Ipp16s* pSrcSpch, Ipp16s*
    pSrcDstSpchOld, Ipp16s* pDstSpchDecomposed);
```

Parameters

pSrcSpch	Pointer to the source vector [320].
pSrcSpchOld	Pointer to the source/destination vector [320] of speech samples of the previous frame.

pDstSpchDecomposed Pointer to the destination vector [320].

Description

The function `ippsDecomposeMLTToDCT_G722` is declared in `ippsc.h` file.
This function decomposes the input signal of the modulated lapped transform (MLT) to the form that fits DCT, so that the MLT transform may be performed in two steps: first, decomposition of the input signal, and second, DCT of the decomposed signal.

The decomposed speech signal is computed as follows:

$$\mathrm{v}(n) = w(159-n)x(159-n) + w(160+n)x(160+n), 0 \le n \le 159$$

$$v(n+160) = w(319-n)(320+n) - w(n)x(639-n), 0 \le n \le 159$$

where

$$w(n) = \sin\left(\frac{\pi}{640}(n+0.5)\right), 0 \le n \le 319$$

and

$$x(n) = \begin{cases} pSrcDstSpchOld[n], 0 \le n < 319 \\ pSrcSpch(n-320), 320 \le n < 639 \end{cases}$$

The input signal *pSrcSpch* is stored in *pSrcDstSpchOld* for use in the next frame.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pSrcSpch* or *pDstSpch* or *pSrcSpchOld* pointer is `NULL`.

DecomposeDCTToMLT

Decomposes IDCT output signal to the form of the MLT transform output signal.

Syntax

```
IppStatus ippsDecomposeDCTToMLT_G722_16s(const Ipp16s* pSrcSpchDecomposed,
    Ipp16s* pSrcDstSpchDecomposedOld, Ipp16s* pDstSpch);
```

Parameters

pSrcSpchDecomposed	Pointer to the source vector [320].
PsrcSpchDecomposedOld	Pointer to the source/destination vector [160] of decomposed speech samples of the previous frame.
pDstSpch	Pointer to the destination vector [320].

Description

The function `ippsDecomposeDCTToMLT_G722` is declared in `ippsc.h` file. This function decomposes the output signal of the Inverse DCT to the form that fits the output of the Inverse Modulated Lapped Transform (IMLT), so that the IMLT transform may be performed in two steps: first, Inverse Discrete Cosine Transform (IDCT), and second, decomposition of the IDCT output.

The decomposed signal is computed as follows:

$pDstSpch(n) = w(n)u(159-n) + w(319-n)u_old(n), \quad 0 \le n \le 159$

$pDstSpch(n+160) = w(160+n)u(n) - w(159-n)u_old(159-n), \quad 0 \le n \le 159$

where

$$w(n) = \sin\left(\frac{\pi}{640} \cdot (n+0.5)\right), \qquad 0 \le n \le 319$$

$u(n) = pSrcSpchDecomposed[n]$, and
$u_old(n) = pSrcDstDecomposedOld[n]$.

The unused high half of the input signal is stored as *u_old*() for use in the next frame:

$u_old(n) = u(160+n), \quad 0 \le n \le 159$

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when the *pSrcSpch* or *pDstSpch* or *pSrcSpchOld* pointer is NULL.

HuffmanEncode_G722

Performs Huffman encoding of the quantized amplitude envelope indexes.

Syntax

```
IppStatus ippsHuffmanEncode_G722_16s32u(int category, int qntAmpEnvIndex, const
    Ipp16s* pSrcMLTCoeffs, Ipp32u* pDstCode, int* pCodeLength);
```

Parameters

category	The category of the Modulated Lapped Transform (MLT) region in the range of [0-7].
qntAmpEnvIndex	The quantized amplitude envelope index in the range of [0-63].
pSrcMLTCoeffs	Pointer to the source vector [20] of raw MLT coefficients.
pDstCode	Pointer to the output Huffman code.
pCodeLength	Pointer to the output Huffman code length in bits.

Description

The function ippsHuffmanEncode_G722 is declared in ippsc.h file. This function performs Huffman encoding of the quantized index of the amplitude envelope of one of the twenty regions of the MLT coefficients.

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when the *pSrcMLTCoeffs* or *pDstCode* or *pCodeLength* pointer is NULL.
IppStsScaleRangeErr	Indicates an error when *category* or *qntAmpEnvIndex* is out of proper range.

G.726 Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following ITU- T recommendations G.726 with Annex A.

These functions are used in the Intel IPP *G.726 Speech Encoder-Decoder* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

The list of these functions is given in the following table:

Table 9-12 Intel IPP G.726.1 Related Functions

Function Base Name	Operation
EncodeGetStateSize_G726	Informative function, returns the number of bytes needed for encoder memory.
EncodeInit_G726	Initializes the memory for the ADPCM encoder.
Encode_G726	Performs ADPCM compression of the uniform PCM input.
DecodeGetStateSize_G726	Informative function, returns the number of bytes needed for decoder memory.
DecodeInit_G726	Initializes the memory for the G726 decoder.
Decode_G726	Performs decompression of the ADPCM bit-stream.

EncodeGetStateSize_G726

Informative function, returns the number of bytes needed for encoder memory.

Syntax

```
IppStatus ippsEncodeGetStateSize_G726_16s8u (unsigned int* pEncSize);
```

Parameters

pEncSize	Pointer to the output memory size in bytes.

Description

The function `ippsEncodeGetStateSize_G726` is declared in `ippsc.h` file. This function gets information about the amount of memory needed to process the G.726 ADPCM compression.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pEncSize* pointer is `NULL`.

EncodeInit_G726

Initializes the memory for the ADPCM encoder.

Syntax

```
IppStatus ippsEncodeInit_G726_16s8u (IppsEncoderState_G726_16s* pEncMem,
    IppSpchBitRate rate);
```

Parameters

pEncMem	Pointer to the input memory buffer of size needed to properly initialize the G.726 encoder.
rate	Encode bit rate of the G.726 encoder, must be one of `IPP_SPCHBR_16000`, `IPP_SPCHBR_24000`, `IPP_SPCHBR_32000`, or `IPP_SPCHBR_40000`.

Description

The function `ippsEncodeInit_G726` is declared in `ippsc.h` file. This function initializes the memory given by the pointer *pEncMem* to enable G.726 ADPCM compression starting from the reset state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when *rate* is not equal to one of the admissible encoding bit rates: `IPP_SPCHBR_16000`, `IPP_SPCHBR_24000`, `IPP_SPCHBR_32000`, or `IPP_SPCHBR_40000`.
`IppStsNullPtrErr`	Indicates an error when the *pEncMem* pointer is `NULL`.

Encode_G726

Performs ADPCM compression of the uniform PCM input.

Syntax

```
IppStatus ippsEncode_G726_16s8u (IppsEncoderState_G726_16s* pEncMem,
    const Ipp16s* pSrc, Ipp8u* pDst, unsigned int len);
```

Parameters

pEncMem	Pointer to the memory buffer that has been initialized for ADPCM encode.
pSrc	Pointer to the uniform PCM input speech vector.
pDst	Pointer to the ADPCM bit-stream output vector.
len	The length of input/output vectors.

Description

The function `ippsEncode_G726` is declared in `ippsc.h` file. This function performs ADPCM compression of the 14-bit uniform PCM speech input (Recommendation G.726, Annex A) with the bit rate on which the G.726 encoder (with memory pointed to by *pEncMem*) was initialized to operate. Each byte of the output vector contains ADPCM compressed value of two, three, four, or five binary digits for 16, 24, 32 or 40 Kbit/s bit-rate ADPCM compression, respectively.

The Mu-Law or A-Law PCM input should be expanded to 14-bit uniform PCM prior to ADPCM compression (see Recommendation G.726). This expansion may be done, for example, by first applying the functions `ippsMuLawToLin_8u16s` or `ippsALawToMuLaw_8u16s` which expand 8-bit Mu-Law or A-Law PCM, respectively, into linear 16-bit PCM.

The linear 16-bit PCM input must be shifted two bits to the right (divided by four) to achieve the 14-bit uniform PCM input appropriate for the `ippsEncode_G726` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when the len is less or equal to 0.
`IppStsNullPtrErr`	Indicates an error when the *pEncMem*, *pSrc*, or *pDst* pointer is NULL.

DecodeGetStateSize_G726

Informative function, returns the number of bytes needed for decoder memory.

Syntax

```
IppStatus ippsDecodeGetStateSize_G726_8u16s (unsigned int* pDecSize);
```

Parameters

pDecSize	Pointer to the output memory size in bytes.

Description

The function `ippsDecodeGetStateSize_G726` is declared in `ippsc.h` file. This function reports the amount of memory needed to process the G.726 ADPCM decompression.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pDecSize* pointer is `NULL`.

DecodeInit_G726

Initializes the memory for the G726 decoder.

Syntax

```
IppStatus ippsDecodeInit_G726_8u16s (IppsDecoderState_G726_16s* pDecMem,
    IppSpchBitRate rate, IppPCMLaw law);
```

Parameters

pDecMem	Pointer to the input memory buffer of size needed to properly initialize the G.726 decoder.
rate	Decode bit rate of the G.726 decoder, must be one of `IPP_SPCHBR_16000`, `IPP_SPCHBR_24000`, `IPP_SPCHBR_32000`, or `IPP_SPCHBR_40000`.
law	Output speech PCM law: must be one of `IPP_PCM_MULAW`, `IPP_PCM_ALAW`, or `IPP_PCM_LINEAR`.

Description

The function `ippsDecodeInit_G726` is declared in `ippsc.h` file. This function initializes the memory given by the pointer *pDecMem* to enable ADPCM decompression starting from the reset state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when *rate* is not equal to one of the admissible decoding bit rates: `IPP_SPCHBR_16000`, `IPP_SPCHBR_24000`, `IPP_SPCHBR_32000`, or `IPP_SPCHBR_40000`; or *law* is not equal to one of the acceptable output PCM values `IPP_PCM_MULAW`, `IPP_PCM_ALAW`, or `IPP_PCM_LINEAR`.
`IppStsNullPtrErr`	Indicates an error when the *pDecMem* pointer is `NULL`.

Decode_G726

Performs decompression of the ADPCM bit-stream.

Syntax

```
IppStatus ippsDecode_G726_8u16s (IppsDecoderState_G726_16s* pDecMem, const
    Ipp8u* pSrc, Ipp16s* pDst, unsigned int len)
```

Parameters

pDecMem	Pointer to the memory buffer that has been initialized for G.726 ADPCM decoder.
pSrc	Pointer to the input vector that contains two, three, four, or five binary digits per byte for 16, 24, 32, or 40 Kbit/s ADPCM bit-stream, respectively.
pDst	Pointer to the 16-bit linear PCM speech output vector.
len	The length of input/output vectors.

Description

The function `ippsDecode_G726` is declared in `ippsc.h` file. This function performs decompression of the ADPCM bit-stream input (Recommendation G.726) into 16-bit linear PCM. The input bit-stream must be ADPCM compressed on the bit rate for which the G.726 decoder was initialized to operate.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsNullPtrErr`	Indicates an error when the *pDecMem*, *pSrc*, or *pDst* pointer is NULL.

G.728 Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following ITU-T* recommendation G.728 with Annexes I, H.

These functions are used in the Intel IPP *G.728 Speech Encoder-Decoder* sample downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

The list of these functions is given in the following table:

Table 9-13 Intel IPP G.728 Related Functions

Function Base Name	Operation
IIRGetStateSize_G728	Gets the size of IIR state structure to be used.
IIR_Init_G728	Initializes the IIR state structure.
IIR_G728	Applies IIR filter to multiple samples.
SynthesisFilterGetStateSize_G728	Gets the size of synthesis filter state structure.
SynthesisFilterInit_G728	Initializes the synthesis filter state structure.
SyntesisFilter_G728	Applies the synthesis filter to multiple samples.
CombinedFilterGetStateSize_G728	Gets the size of combined filter state structure.
CombinedFilterInit_G728	Initializes the combined filter state structure.
CombinedFilter_G728	Applies the combined filter to multiple samples.
PostFilterGetStateSize_G728	Gets the size of the post filter state structure.
PostFilterInit_G728	Initializes the post filter state structure.
PostFilter_G728	Applies the post filter to multiple samples.
PostFilterAdapterGetStateSize_G728	Gets the size of the `IppsPostFilterAdapterState` structure to be used.
PostFilterAdapterStateInit_G728	Initializes the `IppsPostFilterAdapterState` structure.
LPCInverseFilter_G728	Computes the LPC prediction residual.
PitchPeriodExtraction_G728	Extracts pitch period from the LPC prediction residual.
WinHybridGetStateSize_G728	Gets the size of hybrid windowing module state structure.
WinHybridInit_G728	Initializes the hybrid windowing module state structure.
WinHybrid_G728	Applies the hybrid windowing.

Table 9-13 Intel IPP G.728 Related Functions (continued)

Function Base Name	Operation
LevinsonDurbin_G728	Calculates LP coefficients from the autocorrelation coefficients.
CodebookSearch_G728	Searches the codebook for the best code vector.
ImpulseResponseEnergy_G728	Implements shape codebook vector convolution and energy calculation.

IIRGetStateSize_G728

Gets the size of IIR state structure to be used.

Syntax

```
IppStatus ippsIIR16sGetStateSize_G728_16s (int* pSize);
```

Parameters

pSize	Pointer to the output IIR state size value.

Description

The function `ippsIIR16sGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the IIR filter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSize* pointer is `NULL`.

IIR_Init_G728

Initializes the IIR state structure.

Syntax

```
IppStatus ippsIIR16sInit_G728_16s (IppsIIRState16s_G728_16s* pMem );
```

Parameters

pMem	Pointer to the memory allocated for IIR filter.

Description

The function `ippsIIR16sInit_G728` is declared in `ippsc.h` file. This function initializes the IIR state structure using the given memory block.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMem* pointer is `NULL`.

IIR_G728

Applies IIR filter to multiple samples.

Syntax

```
IppStatus ippsIIR16s_G728_16s (const Ipp16s* pCoeffs,
    const Ipp16s* pSrcQntSpeech, Ipp16s* pDstWgtSpeech, int len,
    IppsIIRState16s_G728_16s* pMem );
```

Parameters

pCoeffs	Pointer to the filter coefficients vector [20]: b_0 , ..., b_9 , a_0 , ..., a_9 (in Q14).
pSrcQntSpeech	Pointer to the source vector [*len*].
pDstWgtSpeech	Pointer to the destination vector [*len*].
len	The number of source and destination samples.
pMem	Pointer to the IIR filter state structure.

Description

The function `ippsIIRState16s_G728` is declared in `ippsc.h` file. This function calculates the synthesized speech output by filtering the input quantized speech through the IIR filter one at a time according to the transfer function:

$$\frac{1+\sum_{i=0}^{9} b_i \cdot z^{-i}}{1+\sum_{i=0}^{9} a_i \cdot z^{-i}}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the *pCoeffs*, *pSrcQntSpeech*, *pDstWgtSpeech* or *pMem* pointers is NULL.

SynthesisFilterGetStateSize_G728

Gets the size of synthesis filter state structure.

Syntax

```
IppStatus ippsSynthesisFilterGetStateSize_G728_16s (int* pSize);
```

Parameters

pSize	Pointer to the output size value of the synthesis filter state structure.

Description

The function `ippsSynthesisFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the synthesis filter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSize* pointer is NULL.

SynthesisFilterInit_G728

Initializes the synthesis filter state structure.

Syntax

```
IppStatus ippsSynthesisFilterInit_G728_16s (IppsSynthesisFilterState_G728_16s*
    pMem);
```

Parameters

pMem	Pointer to the memory allocated for synthesis filter.

Description

The function `ippsSynthesisFilterInit_G728` is declared in `ippsc.h` file. This function initializes synthesis filter state structure using the given memory block.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMem* pointer is `NULL`.

SyntesisFilter_G728

Applies the synthesis filter to multiple samples.

Syntax

```
IppStatus ippsSyntesisFilterZeroInput_G728_16s (const Ipp16s* pCoeffs, Ipp16s*
    pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech, Ipp16s* pSpeechSfs,
    IppsSynthesisFilterState_G728_16s* pMem);
```

Parameters

pCoeffs	Pointer to the filter coefficients vector [51]: a_0 , ..., a_{50} in Q14.
pSrcDstExc	Pointer to the input/output gain-scaled excitation vector [5].
excSfs	The input scale of the previous gain-scaled excitation vector.
pDstSpeech	Pointer to the output quantized speech vector [5].
pSpeechSfs	The output scale of the quantized speech vector.

pMem	Pointer to the synthesis filter state structure.

Description

The function `ippsSyntesisFilterZeroInput_G728` is declared in `ippsc.h` file. This function computes the decoded speech vector as the sum of the zero-input response and the zero-state response of the synthesis filter according to the transfer function:

$$\frac{1}{1+\sum_{i=1}^{50} a_i \cdot z^{-i}}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the *pCoeffs*, *pSrcDstExc*, *pDstSpeech*, *pSpeechSfs* or *pMem* pointer is `NULL`.

CombinedFilterGetStateSize_G728

Gets the size of combined filter state structure.

Syntax

```
IppStatus ippsCombinedFilterGetStateSize_G728_16s (int* pSize);
```

Parameters

pSize	Pointer to the output size value of the combined filter state structure.

Description

The function `ippsCombinedFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the combined filter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSize* pointer is `NULL`.

CombinedFilterInit_G728

Initializes the combined filter state structure.

Syntax

```
IppStatus ippsCombinedFilterInit_G728_16s (IppsCombinedFilterState_G728_16s*
    pMem);
```

Parameters

pMem	Pointer to the memory allocated for the combined filter.

Description

The function `ippsCombinedFilterInit_G728` is declared in `ippsc.h` file. This function initializes combined filter state structure using the given memory block.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMem* pointer is `NULL`.

CombinedFilter_G728

Applies the combined filter to multiple samples.

Syntax

```
IppStatus ippsCombinedFilterZeroInput_G728_16s(const Ipp16s* pSyntCoeff, const
    Ipp16s* pWgtCoeff, Ipp16s* pDstWgtZIR, IppsCombinedFilterState_G728_16s*
    pMem);
IppStatus ippsCombinedFilterZeroState_G728_16s(const Ipp16s* pSyntCoeff, const
    Ipp16s* pWgtCoeff, Ipp16s* pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech,
    Ipp16s* pSpeechSfs, IppsCombinedFilterState_G728_16s* pMem);
```

Parameters

pSyntCoeff	Pointer to the filter coefficients vector [50]: a_1 , ..., a_{50} in Q14.

pWgtCoeff	Pointer to the filter coefficients vector [20]: B_1 , ..., B_{10} , A_1 , ..., A_{10} in Q14.
pSrcDstExc	Pointer to the output gain-scaled excitation vector [5].
excSfs	The input scale of the previous gain-scaled excitation vector.
pDstWgtZIR	Pointer to the output zero input response vector [5] of the combined filter.
pDstSpeech	Pointer to the output quantized speech vector [5].
pSpeechSfs	The output scale of the quantized speech vector.
pMem	Pointer to the combined filter state structure.

Description

The functions `ippsCombinedFilterZeroInput_G728` and `ippsCombinedFilterZeroState_G728` are declared in `ippsc.h` file.

`ippsCombinedFilterZeroInput_G728_16s`. This function calculates the zero-input response of the combined filter by superposing two filters, specifically, the 50s-order synthesis filter and the 10s-order IIR filter according to the transfer function

$$\frac{1}{1+\sum_{i=1}^{50} a_i \cdot z^{-i}} \cdot \frac{1+\sum_{i=1}^{10} B_i \cdot z^{-i}}{1+\sum_{i=1}^{10} A_i \cdot z^{-i}}$$

`ippsCombinedFilterZeroState_G728_16s`. This function first performs filtering of the gain-scaled excitation vector through the zero-state combined filter (see the above function). The memory of combined filter is then updated by adding zero-state responses of the synthesis and the IIR filters which it is combined of. The quantized speech output vector is obtained as a by-product of the memory updates.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSyntCoeff*, *pWgtCoeff*, *pSrcDstExc*, *pDstWgtZIR*, *pDstSpeech*, *pSpeechSfs* or *pMem* pointer is `NULL`.

PostFilterGetStateSize_G728

Gets the size of the post filter state structure.

Syntax

```
IppStatus ippsPostFilterGetStateSize_G728_16s (int* pSize);
```

Parameters

pSize	Pointer to the output size value of the post filter state structure.

Description

The function `ippsPostFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the post filter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSize* pointer is `NULL`.

PostFilterInit_G728

Initializes the post filter state structure.

Syntax

```
IppStatus ippsPostFilterInit_G728_16s (IppsPostFilterState_G728_16s* pMem);
```

Parameters

pMem	Pointer to the memory allocated for the post filter.

Description

The function `ippsPostFilterInit_G728` is declared in `ippsc.h` file. This function initializes the post filter state structure using the given memory block.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMem* pointer is NULL.

PostFilter_G728

Applies the post filter to multiple samples.

Syntax

```
IppStatus ippsPostFilter_G728_16s (Ipp16s gl, Ipp16s glb, Ipp16s kp, Ipp16s
    tiltz, const Ipp16s* pCoeffs, const Ipp16s* pSrc, Ipp16s* pDst,
    IppsPostFilterState_G728_16s* pMem);
```

Parameters

gl	The LTP scaling factor.
glb	The LTP product term.
kp	The LTP lag, pitch period of the current frame.
tiltz	The STP tilt-compensation coefficient.
pCoeffs	Pointer to the post filter coefficients vector [20]: B_1 , ..., B_{10} , A_1 , ..., A_{10}.
pSrc	Pointer to the input speech vector [5]; elements -*kp*,...,-1 must be given as memory of the LTP filter.
pDst	Pointer to the output post-filtered speech vector [5].
pMem	Pointer to the post filter state structure.

Description

The function ippsPostFilter_G728 is declared in ippsc.h file. This function performs filtering of input samples by one at a time according to the transfer function that is comprised of LTP filter and STP filter parts:

$$gl \cdot (1 - \mathrm{glb} \cdot z^{-kp}) \cdot \frac{1 - \sum_{i=1}^{10} B_i \cdot z^{-i}}{1 - \sum_{i=1}^{10} A_i \cdot z^{-i}} \cdot (1 + tiltz \cdot z^{-1})$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pCoeffs*, *pSrc*, *pDst* or *pMem* pointer is NULL.

PostFilterAdapterGetStateSize_G728

Gets the size of the IppsPostFilterAdapterState *structure to be used.*

Syntax

```
IppStatus  ippsPostFilterAdapterGetStateSize_G728 (int* pSize);
```

Parameters

pSize	Pointer to the output IppsPostFilterAdapterState structure size value.

Description

The function ippsPostFilterAdapterGetStateSize_G728 is declared in ippsc.h file. This function returns the minimal size of memory to be allocated for proper use of the postfilter adapter.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSize* pointer is NULL.

PostFilterAdapterStateInit_G728

Initializes the `IppsPostFilterAdapterState` *structure.*

Syntax

```
IppStatus  ippsPostFilterAdapterStateInit_G728(
    IppsPostFilterAdapterState* pMem);
```

Parameters

pMem	Pointer to the memory allocated for postfilter adapter.

Description

The function `ippsPostFilterAdapterStateInit_G728` is declared in `ippsc.h` file. This function initializes the `IppsPostFilterAdapterState` structure using the memory block given by *pMem*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMem* pointer is `NULL`.

LPCInverseFilter_G728

Computes the LPC prediction residual.

Syntax

```
IppStatus ippsLPCInverseFilter_G728_16s (const Ipp16s* pSrcSpeech, const
    Ipp16s* pCoeffs, Ipp16s* pDstResidual, IppsPostFilterAdapterState_G728*
    pMem);
```

Parameters

pSrcSpeech	Pointer to the quantized speech buffer [-240…0…4].
pCoeffs	Pointer to the 10th-order LPC filter coefficients [10].
pDstResidual	Pointer to the LPC prediction residual vector [-140…0…99].

pMem	Pointer to the postfilter adapter memory.

Description

The function `ippsLPCInverseFilter_G728_16s` is declared in `ippsc.h` file. This function computes the LPC prediction residual vector for the current decoded speech vector. The 10th-order LPC inverse filter implements the following function:

$$A(z) = 1 - \sum_{i=0}^{9} a_i \cdot z^{-(i+1)}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.

PitchPeriodExtraction_G728

Extracts pitch period from the LPC prediction residual.

```
IppStatus ippsPitchPeriodExtraction_G728_16s (const Ipp16s* pSrcResidual,
    int* pPitchPeriod, IppsPostFilterAdapterState_G728* pMem);
```

Parameters

pSrcResidual	Pointer to the LPC undecimated residual vector d[-140,99].
pPitchPeriod	Pointer to the pitch period of the previous frame $\bar{p}$.
pMem	Pointer to the postfilter adapter memory.

Description

The function `ippsPitchPeriodExtraction_G728_16s` is declared in `ippsc.h` file. The function extracts the pitch period once per frame. It assumes that the input vector contains the LPC prediction residual of previous frames (samples [-139,80]) and of four vectors of the current frame stored, respectively, in elements [80,84], [85,89], [90,94], and [95,99]. The prediction history for the previous frame is stored in the elements [-139,80].

Calculation is as follows:

1. The last 20 elements of the residual computed for the current frame [80,99] are filtered by the 3rd-order elliptic filter and then down-sampled by a factor of 4. The resulting five decimated LPC residual samples are added to the end of decimated LPC residual buffer $\bar{d}[\cdot]$ by shifting other 55 decimated LPC residual samples that were calculated for previous frames.
2. The time lag p_0 is identified which gives the maximum among seven autocorrelations of the undecimated LPC residual

$$\sum_{i=0}^{99} d[i] \cdot d[i-p], \quad p = 4\tau - 3, 4\tau - 2, \ldots, 4\tau + 3$$

where the time lag τ is defined as a position where the maximum of the autocorrelations for the decimated LPC residual is held:

$$\sum_{i=0}^{24} \bar{d}[i] \cdot \bar{d}[i-\tau], \tau = 5, .., 35$$

3. The time lag p_1 is identified that gives the maximum of thirteen autocorrelations of the undecimated LPC residual around the pitch period of the previous frame $\bar{p}$:

$$\sum_{i=0}^{99} d[i] \cdot d[i-p], p = \bar{p} - 6, \ldots, \bar{p} + 6$$

The two lags p_0, p_1 are checked to determine which of them should be set as a

fundamental pitch period. First, the values β_0 and β_1 are calculated as

$$\beta_0 = \frac{\sum_{i=0}^{99} d[i] \cdot d[i - p_0]}{\sum_{i=0}^{99} d[i - p_0] \cdot d[i - p_0]}$$

$$\beta_1 = \frac{\sum_{i=0}^{99} d[i] \cdot d[i - p_1]}{\sum_{i=0}^{99} d[i - p_1] \cdot d[i - p_1]}$$

Then the output pitch period is chosen as follows:

$$pitchPeriod = \begin{cases} p_0 & if \;\; \beta_1 \le 0.4 \cdot \beta_0 \\ p_1 & if \;\; \beta_1 > 0.4 \cdot \beta_0 \end{cases}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.

WinHybridGetStateSize_G728

Gets the size of hybrid windowing module state structure.

Syntax

```
IppStatus ippsWinHybridGetStateSize_G728_16s (int M, int L, int N, int DIM,
    int* pSize);
```

Parameters

M	The input length of the LPC window.
L	The input adaptation cycle size in samples.
N	The input number of non-recursive window samples.
DIM	The input block size used for block scaling of the input speech by the `ippsWinHybrid_G728` function.
pSize	Pointer to the output size value of the hybrid windowing module state structure.

Description

The function `ippsWinHybridGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the hybrid windowing module according to the given window parameters.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSize* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *L*, *M* or *N* are less than or equal to zero.

WinHybridInit_G728

Initializes the hybrid windowing module state structure.

Syntax

```
IppStatus ippsWinHybridInit_G728_16s (const Ipp16s* pWinTab, int M, int L,
    int N, int DIM, Ipp16s a2L, IppsWinHybridState_G728_16s* pMem);
```

Parameters

pWinTab	The input vector of windowing coefficients.
M	The input length of LPC window. Must be not less than 10.
L	The input adaptation cycle size in samples.
N	The input number of non-recursive window samples.

a2L	The a^{2L} multiple used in calculation of the recursive component in hybrid windowing module.
N	The input number of non-recursive window samples.

Description

The function `ippsWinHybridInit_G728` is declared in `ippsc.h` file. This function initializes the hybrid windowing module state structure using the given memory block. The recursive component and the previous speech samples are zeroed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMem* pointer is `NULL`.

WinHybrid_G728

Applies the hybrid windowing.

Syntax

```
IppStatus ippsWinHybridBlock_G728_16s(Ipp16s bfi, const Ipp16s* pSrc, const
    Ipp16s* pSrcSfs, Ipp16s* pDst, IppsWinHybridState_G728_16s* pMem);
IppStatus ippsWinHybrid_G728_16s(Ipp16s bfi, const Ipp16s* pSrc, const
    Ipp16s* pSrcSfs, Ipp16s* pDst, IppsWinHybridState_G728_16s* pMem);
```

Parameters

bfi	The input bad frame indicator: “1” signifies a bad frame, any other value signifies a good frame.
pSrc	Pointer to the input speech vector [20].
pSrcSfs	Pointer to the input vector [5] of scale factors for elements of the input speech vector. Each element of the *pSrcSfs* vector may specify a separate scale factor for a block of input speech vector elements. Thus, *pSrcSfs*[0] is the scale factor of the elements *pSrc*[0],..*pSrc*[*DIM*-1]; *pSrcSfs*[1] is the scale factor for *pSrc*[*DIM*],..*pSrc*[2**DIM*-1], and so on, where *DIM* is a block length that must be defined by the `WinHybridInit_G728()` function.

pDst	Pointer to the output reflection coefficients vector [M+1], where M is the length of the LPC window defined in WinHybridInit_G728() function.
pMem	Pointer to the post filter state structure.

Description

The function ippsWinHybrid_G728 is declared in ippsc.h file. This function first applies the window to the input speech vector and then calculates the autocorrelation coefficients needed in LPC analysis by the formulae:

$$R_m(i) = r_m(i) + \sum_{k=m-N}^{m-1} s_m(k)s_m(k-i), m = 0,..,M$$

where the recursive component of adaptation cycle is calculated as follows:

$$r_m(i) = \alpha^{2L} \cdot r_{m-L}(i) + \sum_{k=m-L-N}^{m-N-1} s_m(k)s_m(k-i)$$

The recursive part is calculated using the data calculated in previous adaptation cycle and stored in module memory.

A *white noise correction* is applied to autocorrelation coefficients by increasing the energy as follows:

$$r_m(0) = \frac{257}{256} \cdot r_m(0)$$

The recursive component and previous speech samples are stored in the module memory and may be used in the next adaptation cycle.

If the bad frame indicator is on, then only 10 autocorrelation coefficients are calculated and output.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcSfs*, *pSrc*, *pDst* or *pMem* pointer is NULL.

LevinsonDurbin_G728

Calculates LP coefficients from the autocorrelation coefficients.

Syntax

```
IppStatus ippsLevinsonDurbin_G728_16s_Sfs(const Ipp16s* pSrcAutoCorr, int
    order, Ipp16s* pDstLPC, Ipp16s* pDstResidualEnergy, Ipp16s*
    pDstScaleFactor);
IppStatus ippsLevinsonDurbin_G728_16s_ISfs(const Ipp16s* pSrcAutoCorr, int
    numSrcLPC, int order, Ipp16s* pSrcDstLPC, Ipp16s* pSrcDstResidualEnergy,
    Ipp16s* pSrcDstScaleFactor);
```

Parameters

pSrcAutoCorr	Pointer to the input autocorrelation coefficients vector [*order*+1].
order	The input number of LP coefficients to calculate.
numSrcLPC	The input number of pre-calculated LPCs.
pDstLPC	Pointer to the output LPC vector [*order*].
pSrcDstLPC	Pointer to the input/output LPC vector [*order*].
pDstResidualEnergy	Pointer to the output residual energy.
pSrcDstResidualEnergy	Pointer to the input/output residual energy of the pre-calculated LPC.
pDstScaleFactor	Pointer to the output scale factor of the LPC vector.
pSrcDstScaleFactor	Pointer to the input scale factor of the pre-calculated LPC and the output scale factor of the output LPC vector.

Description

The functions `ippsLevinsonDurbin_G728_16s_Sfs` and `ippsLevinsonDurbin_G728_16s_ISfs` are declared in `ippsc.h` file.

ippsLevinsonDurbin_G728_16s_Sfs. This function may be used to calculate Linear Prediction (LP) coefficients by solving the following set of linear equations:

$$\sum_{i=0}^{order-1} a_i \cdot r(|i-k|) = r(k), k = 1,2,\ldots,order$$

where a_i , i = 0,1,…,*order*-1 are the LP coefficients to be calculated and stored in the output LPC vector. The description of the Levinson-Durbin algorithm used by this function may be found in ippsLevinsonDurbin_G729 function.

Both ippsLevinsonDurbin_G728 and ippsLevinsonDurbin_G729B calculate mathematically the same, but not bit exact LPCs. The difference is that the ippsLevinsonDurbin_G729B function outputs LPCs in Q12, while the ippsLevinsonDurbin_G728 function automatically rescales the LPCs if overflow occurs.

ippsLevinsonDurbin_G728_16s_Isfs. This function may be used to continue Levinson-Durbin recursion for bigger *order*. The LPCs, their scale factor and the residual energy of the previous recursion and the additional autocorrelation coefficients are used to resume recursion.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the input or output pointers is NULL.
ippStsRangeErr	Indicates an error when *order* is less than or equal to 0, or when *order* < *numSrcLPC* for the in-place function.

CodebookSearch_G728

Searches the codebook for the best code vector.

Syntax

```
IppStatus ippsCodebookSearch_G728_16s(const Ipp16s* pSrcCorr, const Ipp16s*
    pSrcEnergy, int* pDstShapeIdx, int* pDstGainIdx, short* pDstCodebookIdx,
    IppSpchBitRate rate);
```

Parameters

pSrcCorr	Pointer to the input vector [5] of time-reversed convolution of the target signal.
pSrcEnergy	Pointer to the input vector [128] of the energy of convolved shape codevector.
pDstShapeIdx	Pointer to the output best 7-bit shape codebook index.
pDstGainIdx	Pointer to the output best 3-bit gain codebook index.
pDstCodebookIdx	Pointer to the output best codebook index to be transmitted.
rate	Input coding bit rate.

Description

The function `ippsCodebookSearch_G728` is declared in `ippsc.h` file. This function implements the “Error calculator and best codebook index selector” block used to search through the gain codebook and the shape codebook for the best combination of the gain and shape codebook indexes.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcCorr*, *pSrcEnergy*, *pDstShapeIdx*, *pDstGainIdx* or *pDstCodebookIdx* pointer is `NULL`.
IppStsRangeErr	Indicates an error when *rate* is not one of the acceptable values `IPP_SPCHBR_16000`, `IPP_SPCHBR_12800` or `IPP_SPCHBR_9600` for the 16, 12.8 or 9.6 Kbit/s coding bit rates, respectively.

ImpulseResponseEnergy_G728

Implements shape codebook vector convolution and energy calculation.

Syntax

```
IppStatus ippsImpulseResponseEnergy_G728_16s(const Ipp16s* pSrcImpResp, Ipp16s*
    pDstEnergy);
```

Parameters

pSrcImpResp	Pointer to the input impulse response vector [5] of the synthesis and weighted filter.
pDstEnergy	Pointer to the output energy vector [128] of the convolved shape codevector.

Description

The function `ippsImpulseResponseEnergy_G728` is declared in `ippsc.h` file. This function implements the "Shape codevector convolution and energy calculator" block used to calculate the energy of the convolved shape codevector.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcImpResp* or *pDstEnergy* pointer is `NULL`.

Echo Canceller Related Functions

Intel IPP functions described in this section implement building blocks that can be used to create an acoustic and network echo cancellers compliant to the appropriate ITU-T Recommendations: G.168-2000, and G.167 (superseded by G.161, P.340).

The list of Intel IPP Echo Canceller related functions is given in the following table:

Table 9-14 Intel IPP Echo Canceller Related Functions

Function Base Name	Operation
SubbandProcessGetSize	Returns size of the subband process state structure.
SubbandProcessInit	Initializes the subband process state structure.
SubbandAnalysis	Decomposes a frame into a complex subband representation
SubbandSynthesis	Reconstructs frame from a complex subband representation.
SubbandControllerGetSize_EC	Returns size of the subband controller state structure.
SubbandControllerInit_EC	Initializes the subband controller state structure.
SubbandControllerReset_EC	Resets subband controller state.

Table 9-14 Intel IPP Echo Canceller Related Functions (continued)

Function Base Name	Operation
SubbandController_EC	Updates filter coefficients and returns output gain coefficients.
SubbandControllerUpdate_EC	Updates controller state and returns the step sizes.
SubbandControllerDTGetSize_EC	Returns size of the subband DT controller state structure.
SubbandControllerDTInit_EC	Initializes the subband DT controller state structure.
SubbandControllerDTReset_EC	Resets subband DTcontroller state.
SubbandControllerDT_EC	Updates filter coefficients and returns output gain coefficients.
SubbandControllerDTUpdate_EC	Updates DT controller state and returns the step sizes.
ToneDetectGetStateSize_EC	Returns size of the tone detector state structure.
ToneDetectInit_EC	Initializes the tone detector state structure.
ToneDetect_EC	Detects the signal of 2100 Hz frequency with every 450 ms phase reversal.
FullbandControllerGetSize_EC	Returns size of the fullband controller state structure.
FullbandControllerInit_EC	Initializes the fullband controller state structure.
FullbandControllerUpdate_EC	Updates the fullband controller state and returns the step sizes.
FullbandController_EC	Updates filter coefficients and returns output gain coefficients.
FullbandControllerReset_EC	Resets fullband controller state.
FIR_EC	Computes FIR filter results.
FIRSubband_EC, FIRSubbandLow_EC	Computes the frequency-domain adaptive filter output
FIRSubbandCoeffUpdate_EC, FIRSubbandLowCoeffUpdate_EC	Updates the adaptive filter coefficients
NLMS_EC	Performs FIR filtering with coefficients update.

SubbandProcessGetSize

Returns size of the subband process state structure.

Syntax

```
IppStatus ippsSubbandProcessGetSize_32f(int order, int windowLen, int*
    pStateSize, int* pInitBufSize, int* pBufSize);
IppStatus ippsSubbandProcessGetSize_16s(int order, int windowLen, int*
    pStateSize, int* pInitBufSize, int* pBufSize);
```

Parameters

order	Number of subbands is equal to $2^{order-1} + 1$.
windowLen	Length of window.
pStateSize	Pointer to the computed value of buffer size for the subband process state structure.
pInitBufSize	Pointer to the computed buffer size for use in the initialization function.
pBufSize	Pointer to the computed size of work buffer.

Description

The functions `ippsSubbandProcessGetSize` are declared in the `ippsc.h` file. These functions return the sizes of memory buffers required for the functions `ippsSubbandAnalysis` and `ippsSubbandSynthesis` to operate. The memory must be allocated and initialized by means of the `ippsSubbandProcessInit` function prior to using the functions `ippsSubbandAnalysis` or `ippsSubbandSynthesis` .

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pStateSize*, *pInitBufSize*, or *pBufSize* is `NULL`.
`ippStsBadArgErr`	Indicates an error when *order* is less than or equal to 0 or when *windowLen* is not divisible by 2^{order}.

SubbandProcessInit

Initializes the subband process state structure.

Syntax

```
IppStatus ippsSubbandProcessInit_32f(IppsSubbandProcessState_32f* pState, int
    order, int frameSize, int windowLen, const Ipp32f* pWindow, Ipp8u*
    pInitBuf);
IppStatus ippsSubbandProcessInit_16s(IppsSubbandProcessState_16s* pState, int
    order, int frameSize, int windowLen, const Ipp32s* pWindow, Ipp8u*
    pInitBuf);
```

Parameters

pState	Pointer to the subband process state structure to be created.
order	Number of subbands is equal to $2^{order-1} + 1$.
frameSize	Size of frame. Must be in range from 1 to 2^{order}.
windowLen	Window length.
pWindow	Pointer to window coefficients . May be NULL if the *order*, *frameSize* and *windowLen* are equal to one of the predefined sets: (5, 24, 128) or (6, 44, 256). In this case the predefined window is used.
pInitBuf	Pointer to the initialized buffer.

Description

The functions ippsSubbandProcessInit are declared in the ippsc.h file. These functions initialize the state structure and internal data of the subband process algorithm. The ippsSubbandAnalysis and ippsSubbandSynthesis functions use this memory in their operation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pInitBuf* or *pState* is NULL.

ippStsBadArgErr	Indicates an error when *order* is less or equal to 0, or when *windowLen* is not divisible by 2^{order}, or when *frameSize* is less than or equal to 0, or when *pWindow* is NULL and the appropriate internal window does not exist.

SubbandAnalysis

Decomposes a frame into a complex subband representation.

Syntax

```
IppStatus ippsSubbandAnalysis_32f32fc(const Ipp32f* pSignal, Ipp32fc*
    pSubbands, IppsSubbandProcessState_32f* pState, Ipp8u* pBuf);
IppStatus ippsSubbandAnalysis_16s32sc_Sfs(const Ipp16s* pSignal, Ipp32sc*
    pSubbands, IppsSubbandProcessState_16s* pState, int scaleFactor, Ipp8u*
    pBuf);
```

Parameters

pState	Pointer to the subband process state stucture.
pSignal	Pointer to the source vector. Vector length must be equal to the frame size which was used to initialize the subband process algorithm by ippsSubbandProcessInit function.
pSubbands	Pointer to the resulting subband vector. Vector length is equal to the number of subbands calculated as $2^{order-1} + 1$, where *order* is specified in the function ippsSubbandProcessInit.
pBuf	Pointer to the work buffer of size specified by the parameter *pBufSize* in the ippsSubbandProcessGetSize function. This buffer must be allocated by the user.

Description

The functions ippsSubbandAnalysis are declared in the ippsc.h file. These functions perform subband analysis using the oversampled DFT filter bank.

Computation steps are as follows:

1) Calculate the weighted sum as

$$X(j) = \sum_{i=0}^{windowLen/\ fftLen-1} window(i \cdot fftLen + j) \cdot x(i \cdot fftLen + j)$$

for $j = 0,\ldots, fftLen - 1$, $fftLen = 2^{order}$.

Here x denotes the input sample buffer of size $windowLen$.

2) Calculate and return the Fast Fourier Transform of X.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.

SubbandSynthesis

Reconstructs frame from a complex subband representation.

Syntax

```
IppStatus ippsSubbandSynthesis_32fc32f(const Ipp32fc* pSubbands, Ipp32f*
    pSignal, IppsSubbandProcessState_32f* pState, Ipp8u* pBuf);
IppStatus ippsSubbandSynthesis_32sc16s_Sfs(const Ipp32sc* pSubbands, Ipp16s*
    pSignal, IppsSubbandProcessState_16s* pState, int scaleFactor, Ipp8u* pBuf);
```

Parameters

pState	Pointer to the subband process state stucture.
pSubbands	Pointer to the resulting subband vector. Vector length is equal to the number of subbands calculated as $2^{order-1} + 1$, where *order* is specified in the function ippsSubbandProcessInit.
pSignal	Pointer to the destination signal vector. Vector length is equal to frame size which was used to initialize the subband process algorithm by ippsSubbandProcessInit function.
pBuf	Pointer to the work buffer of size computed by the function ippsSubbandProcessGetSize (returned via *pBufSize* parameter).

Description

The functions `ippsSubbandSynthesis` are declared in the ippsc.h file.
These functions perform subband synthesis using the oversampled DFT filter bank.

Computation steps are as follows:

1) Calculates *IX* as the inverse Fast Fourier Transform of the input subband sample *X*. Transform length is *fftLen* = 2^{order}.

2) Shifts internal buffer by *frameSize* and sets to zero the undefined buffer values.

3) Updates buffer values as

$$x(j) := x(j) + window(j) * IX(j \% fftLen),$$

$$for\ j = 0, \ldots, windowLen - 1$$

4) Returns most recent *frameSize* samples from the buffer.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.

SubbandControllerGetSize_EC

Returns size of the subband controller state structure.

Syntax

```
IppStatus ippsSubbandControllerGetSize_EC_32f(int numSubbands, int frameSize,
    int numSegments, ippECFrequency sampleFreq, int* pSize);
IppStatus ippsSubbandControllerGetSize_EC_16s(int numSubbands, int frameSize,
    int numSegments, ippECFrequency sampleFreq, int* pSize);
```

Parameters

numSubbands	Number of subbands.
frameSize	Size of the frame.
numSegments	Number of segments.
sampleFreq	Sample frequency.

pSize	Pointer to the computed buffer size value.

Description

The functions `ippsSubbandControllerGetSize_EC` are declared in the `ippsc.h` file. These functions return the sizes of memory required for the `ippsSubbandControllerUpdate_EC` and `ippsSubbandController_EC` functions to operate. The memory must be allocated and initialized by means of the `ippsSubbandControllerInit_EC` function prior to using the functions `ippsSubbandControllerUpdate_EC` and `ippsSubbandController_EC`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSize* pointer is `NULL`.
`ippStsBadArgErr`	Indicates an error when *numSubbands* is less than or equal to 0, or *numSegment* is less than or equal to 0, or *frameSize* is less than or equal to 0.

SubbandControllerInit_EC

Initializes the subband controller state structure.

Syntax

```
IppStatus ippsSubbandControllerInit_EC_32f(IppsSubbandControllerState_EC_32f*
    pState, int numSubbands, int frameSize, int numSegments, ippECFrequency
    sampleFreq);
IppStatus ippsSubbandControllerInit_EC_16s(IppsSubbandControllerState_EC_16s*
    pState, int numSubbands, int frameSize, int numSegments, ippECFrequency
    sampleFreq);
```

Parameters

pState	Pointer to the subband controller state structure to be created.
numSubbands	Number of subbands.
frameSize	Size of the frame.
numSegments	Number of segments.
sampleFreq	Sample frequency.

Description

The functions ippsSubbandControllerInit_EC are declared in the ippsc.h file. These functions initialize the state structure and the internal data of the subband controller algorithm. The functions ippsSubbandControllerUpdate_EC and ippsSubbandController_EC use this memory in their operation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsBadArgErr	Indicates an error when *numSubbands* is less than or equal to 0, or *numSegments* is less than or equal to 0, or *frameSize* is less than or equal to 0.

SubbandControllerUpdate_EC

Updates controller state and returns the step sizes.

Syntax

```
IppStatus ippsSubbandControllerUpdate_EC_32f(const Ipp32f* pSrcRin, const
    Ipp32f* pSrcSin, const Ipp32fc** ppSrcRinSubbandsHistory, const Ipp32fc*
    pSrcSinSubbands, double* pDstStepSize, IppsSubbandControllerState_EC_32f*
    pState);
IppStatus ippsSubbandControllerUpdate_EC_16s(const Ipp16s* pSrcRin, const
    Ipp16s* pSrcSin, const Ipp32sc** ppSrcRinSubbandsHistory, const Ipp32sc*
    pSrcSinSubbands, IppAECScaled32s* pDstStepSize,
    IppsSubbandControllerState_EC_16s* pState);
```

Parameters

pState	Pointer to the subband controller state structure.
pSrcRin	Pointer to receive-in signal frame.
pSrcSin	Pointer to send-in signal frame. Frame size is specified in ippsSubbandControllerInit_EC function.
ppSrcRinSubbandsHistory	Pointer to an array of pointers to the most recent receive-in blocks. Size of the array is equal to *numSegments* specified in ippsSubbandControllerInit_EC function.

pSrcSinSubbands	Pointer to subband representation of send-in signal frame (or NULL). Size of the array is equal to *numSubbands* specified in ippsSubbandControllerInit_EC function.
pDstStepSize	Pointer to the vector of step sizes. Vector length is equal to *numSubbands* specified in ippsSubbandControllerInit_EC function.

Description

The functions ippsSubbandControllerUpdate_EC are declared in the ippsc.h file. These functions update the internal state of the subband controller and return the step size values for further use in the coefficients' adaptation process.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pState*, *pRin*, *pSin*, *pRinSubbands*, or *pStepSize* is NULL.

SubbandController_EC

Updates filter coefficients and returns output gain coefficients.

Syntax

```
IppStatus ippsSubbandController_EC_32f(const Ipp32fc* pSrcAdaptiveFilterErr,
    const Ipp32fc* pSrcFixedFilterErr, Ipp32fc** ppDstAdaptiveCoefs, Ipp32fc**
    ppDstFixedCoefs, Ipp32f* pDstSGain, IppsSubbandControllerState_EC_32f*
    pState);
IppStatus ippsSubbandController_EC_16s(const Ipp32sc* pSrcAdaptiveFilterErr,
    const Ipp32sc* pSrcFixedFilterErr, Ipp32sc** pDstpAdaptiveCoefs, Ipp32sc**
    ppFixedCoefs, Ipp32s* pDstSGain, IppsSubbandControllerState_EC_16s* pState);
```

Parameters

pState	Pointer to SubbandController_EC state structure.
pSrcAdaptiveFilterErr	Pointer to the adaptive filter error vector. Vector length is equal to *numSubbands*, specified in ippsSubbandControllerInit_EC function.

pSrcFixedFilterErr	Pointer to the fixed filter error vector. Vector length is equal to *numSubbands*.
ppDstAdaptiveCoefs	Pointer to an array of pointers to the adaptive filter coefficients vectors. Size of the array is equal to *numSegments* specified in `ippsSubbandControllerInit_EC` function.
ppDstFixedCoefs	Pointer to an array of pointers to the fixed filter coefficients vectors. Size of the array is equal to *numSegments* specified in `ippsSubbandControllerInit_EC` function.
pDstSGain	Pointer to the send gain coefficient.

Description

The functions `ippsSubbandController_EC` are declared in the `ippsc.h` file.

These functions compare powers of adaptive filter error and fixed filter error and track changes of adaptive filter coefficients power.

If the adaptive filter has a significantly smaller error and is stable (which is indicated by coefficients power changing slowly), its coefficients are copied to the fixed filter. This case corresponds to “no double-talk” mode.

If the fixed filter has a significantly smaller error, its coefficients are copied to the adaptive filter. This case corresponds to the double-talk mode.

Send gain coefficient calculation (non-linear processor technology) is based on presence of the double-talk mode and changes of receive-in and send-in signal powers.

The subband controller state staurctue *pState* must be initialized by the function `ippsSubbandControllerInit_EC` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.

SubbandControllerReset_EC

Resets the subband controller state.

Syntax

```
IppStatus ippsSubbandControllerReset_EC_32f(IppsSubbandControllerState_EC_32f*
    pState);
IppStatus ippsSubbandControllerReset_EC_16s(IppsSubbandControllerState_EC_16s*
    pState);
```

Parameters

pState	Pointer to the subband controller state structure.

Description

The functions `ippsSubbandControllerReset_EC` are declared in the `ippsc.h` file. These functions reset the subband controller state. You can use these functions after an interruption in adaptation process occurred.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* pointer is `NULL`.

SubbandControllerDTGetSize_EC

Returns the size of the subband DT controller state structure.

Syntax

```
IppStatus ippsSubbandControllerDTGetSize_EC_16s(int numSubbands, int frameSize,
    int numSegments, ippPCMFrequency sampleFreq, int* pSize);
```

Parameters

numSubbands	Number of subbands.
frameSize	Size of the frame.

numSegments	Number of segments.
sampleFreq	Sample frequency.
pSize	Pointer to the computed buffer size value.

Description

The function `ippsSubbandControllerDTGetSize_EC` is declared in the `ippsc.h` file. This function returns the size of memory required for the state structure and the internal data of the subband DT controller algorithm. This state structure is used by the functions `ippsSubbandControllerDTUpdate_EC` and `ippsSubbandControllerDT_EC`, and must be allocated and initialized by means of the `ippsSubbandControllerDTInit_EC` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSize* pointer is `NULL`.
`ippStsBadArgErr`	Indicates an error when *numSubbands* is less than or equal to 0, or *numSegment* is less than or equal to 0, or *frameSize* is less than or equal to 0.

SubbandControllerDTInit_EC

Initializes the subband DT controller state structure.

Syntax

```
IppStatus ippsSubbandControllerDTInit_EC_16s(IppsSubbandControllerDTState_EC_16s*
    pState, int numSubbands, int frameSize, int numSegments, ippPMCFrequency
    sampleFreq);
```

Parameters

pState	Pointer to the subband controller DT state structure to be created.
numSubbands	Number of subbands.
frameSize	Size of the frame.
numSegments	Number of segments.
sampleFreq	Sample frequency.

Description

The functions `ippsSubbandControllerDTInit_EC` are declared in the `ippsc.h` file.

This function initializes the state structure and the internal data of the subband DT controller algorithm. The size of the required memory should be computed using the function `ippsSubbandControllerDTGetSize_EC` beforehand.

This state structure is used by the functions `ippsSubbandControllerDT_EC` and `ippsSubbandControllerDTUpdate_EC`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointer *pState* is `NULL`.
`ippStsBadArgErr`	Indicates an error when *numSubbands* is less than or equal to 0, or *numSegments* is less than or equal to 0, or *frameSize* is less than or equal to 0.

SubbandControllerDTReset_EC

Resets the subband DT controller state.

Syntax

```
IppStatus ippsSubbandControllerDTReset_EC_16s(IppsSubbandControllerState_EC_16s*
    pState);
```

Parameters

pState	Pointer to the subband controller DT state structure.

Description

The function `ippsSubbandControllerDTReset_EC` is declared in the `ippsc.h` file. This function resets the subband controller state. It can be used after an interruption in adaptation process occurred.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointer *pState* is `NULL`.

SubbandControllerDT_EC

Updates filter coefficients and returns output gain coefficients.

Syntax

```
IppStatus ippsSubbandControllerDT_EC_16s(const Ipp32sc* pSrcAdaptiveFilterErr,
    const Ipp32sc* pSrcFixedFilterErr, Ipp32sc** ppDstAdaptiveCoefs, Ipp32sc**
    ppDstFixedCoefs, Ipp64s* pSrcDstFilterPwr, int* pDstStsAdapt, Ipp64s
    pwrDelta, int filterUpdateEnabled, int adaptationEnabled, int startSubband,
    IppsSubbandControllerDTState_EC_16s* pState);
```

Parameters

pState	Pointer to state structure of the subband DT controller.
pSrcAdaptiveFilterErr	Pointer to the input adaptive filter error vector. Vector length is equal to *numSubbands*, specified in the function ippsSubbandControllerDTInit_EC.
pSrcFixedFilterErr	Pointer to the input fixed filter error vector. Vector length is equal to *numSubbands*.
ppDstAdaptiveCoefs	Pointer to an array of pointers to the adaptive filter coefficients vectors. Size of the array is equal to *numSegments* specified in ippsSubbandControllerDTInit_EC function.
ppDstFixedCoefs	Pointer to an array of pointers to the fixed filter coefficients vectors. Size of the array is equal to *numSegments* specified in ippsSubbandControllerDTInit_EC function.
pSrcDstFilterPwr	Pointer to an input/output power of adaptive filter coefficient.
pDstStsAdapt	Pointer to output flag which shows the state of the controller. *pDstStsAdapt* = -1 - coefficients of the adaptive filter have been copied to the fixed filter; *pDstStsAdapt* = 0 - coefficients of the adaptive and fixed filters are not changed; *pDstStsAdapt* = 1 - coefficients of the fixed filter have been copied to the fixed filter.
pwrDelta	Difference between the power of the adaptive filter coefficients of the current frame and the power of the adaptive filter coefficients of the previous frame.

filterUpdateEnabled	Difference between the power of the adaptive filter coefficients of the current frame and the power of the adaptive filter coefficients of the previous frame.
adaptationEnabled	Indicates if an adaptation is enabled or not.
startSubband	Indicates the number of subband the filtering starts from (0 ≤ *startSubband* < *numSubbands*).

Description

The function ippsSubbandControllerDT_EC is declared in the ippsc.h file.

This function compares powers of the adaptive filter error and fixed filter error, and tracks changes of the adaptive filter coefficients power.

If the adaptive filter has a significantly smaller error and is stable (which is indicated by coefficients power changing slowly), its coefficients are copied to the fixed filter. If the fixed filter has a significantly smaller error or adaptive filter is not stable (which is indicated by coefficients power changing rapidly), its coefficients are copied to the adaptive filter. Output flag *pDstStsAdapt* shows if there were conservations or restorations of the adaptive filter coefficients.

The subband DT controller state staurctue *pState* must be initialized by the function ippsSubbandControllerDTInit_EC beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsBadArgErr	Indicates an error when when *startSubband* is less than 0 or greater than or equal to the *numSubbands*.

SubbandControllerDTUpdate_EC

Updates the state of the DT controller and returns the step sizes.

Syntax

```
IppStatus ippsSubbandControllerDTUpdate_EC_16s(const Ipp32sc**
    ppSrcRinSubbandsHistory, const Ipp32sc* pSrcSinSubbands, const Ipp32sc*
    pSrcFilterErr, Ipp32s_EC_Sfs* pDstStepSize, int* pIsDT, int* pDisabledNLP,
    int startSubband, IppsSubbandControllerDTState_EC_16s* pState);
```

Parameters

pState	Pointer to state structure of the subband DT controller.
ppSrcRinSubbandsHistory	Pointer to an array of pointers to the most recent receive-in blocks. Size of the array is equal to *numSegments* specified in the function ippsSubbandControllerDTInit_EC.
pSrcSinSubbands	Pointer to subband representation of send-in signal frame (or NULL). Size of the array is equal to *numSubbands* specified in the function ippsSubbandControllerDTInit_EC.
pSrcFilterErr	Pointer to the input filter error vector. Vector length is equal to *numSubbands* specified in the function ippsSubbandControllerDTInit_EC.
pDstStepSize	Pointer to the vector of step sizes. Vector length is equal to *numSubbands* specified in the function ippsSubbandControllerDTInit_EC.
isDT	Pointer to the indicator that indicates the double-talk condition.
disabledNLP	Pointer to the indicator that indicates whether NLP is disable. If it equals 1 - NLP is disabled, if it equals 0 - NLP is not disabled.
startSubband	Indicates the number of subband the filtering starts from ($0 \le startSubband < numSubbands$).

Description

The functions ippsSubbandControllerDTUpdate_EC are declared in the ippsc.h file.

This function updates the internal state of the subband DT controller and return the step size values for further use in the coefficients' adaptation process.

The subband DT controller state staurctue *pState* must be initialized by the function ippsSubbandControllerDTInit_EC beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsBadArgErr	Indicates an error when when *startSubband* is less than 0 or greater than or equal to the *numSubbands*.

ToneDetectGetStateSize_EC

Returns size of the tone detector state structure.

Syntax

```
IppStatus ippsToneDetectGetStateSize_EC_16s(ippECFrequency sampleFreq, int*
    pSize);
IppStatus ippsToneDetectGetStateSize_EC_32f(ippECFrequency sampleFreq, int*
    pSize);
```

Parameters

sampleFreq	Sample frequency.
pSize	Pointer to the computed buffer size value.

Description

The functions ippsToneDetectGetStateSize_EC are declared in the ippsc.h file. These functions return the size of memory buffers required for the ippsToneDetect_EC functions to operate.

The appropriate memory must be and initialized by means of the ippsToneDetectInit_EC function prior to using the ippsToneDetect_EC function .

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSize* pointer is NULL.
ippStsRangeErr	Indicates an error when *sampleFreq* is not a valid element of the enumerated type IppFrequency.

ToneDetectInit_EC

Initializes the tone detector state structure.

Syntax

```
IppStatus ippsToneDetectInit_EC_16s(IppsToneDetectState_EC_16s* pState,
    ippECFrequency sampleFreq);
IppStitus ippsToneDetectInit_EC_32f(IppsToneDetectState_EC_32f* pState,
    ippECFrequency sampleFreq);
```

Parameters

pState	Pointer to the memory buffer to be initialized as the tone detector state structure. The memory size must be computed by the ippsToneDetectGetStateSize_EC function and returned via *pSize* parameter.
sampleFreq	Sample frequency.

Description

The functions ippsToneDetectInit_EC are declared in the ippsc.h file. These functions initialize the internal data of the tone detect algorithm. The ippsToneDetect_EC functions use this memory in their operation.

Return Values

ippStsNoErr	Indicates no error.
ippStsRangeErr	Indicates an error when *sampleFreq* is not a valid element of the enumerated type IppFrequency.

ToneDetect_EC

Detects the signal of 2100 Hz frequency with every 450 ms phase reversal.

```
IppStatus ippsToneDetect_EC_16s(const Ipp16s* pSignal, int len, int* pResult,
    IppsToneDetectState_EC_16s* pState);
IppStatus ippsToneDetect_EC_32f(const Ipp32f* pSignal, int len, int* pResult,
    IppsToneDetectState_EC_32f* pState);
```

Parameters

pState	Pointer to the tone detector state structure.
pSignal	Pointer to signal vector.
len	Number of samples in signal vector.
pResult	Pointer to the result value. If the value is not zero, this means that the tone was detected.

Description

The functions `ippsToneDetect_EC` are declared in the `ippsc.h` file.

These functions apply an infinite impulse response (IIR) band-pass filter centered at 2100 Hz frequency. Functions compare powers of the signal before and after the filter.

High-level signal after the filter with periodical short blips is considered as the tone detected.

NOTE. Detection can occur when summary length of processed signal vectors is equal to or greater than 900 ms.

Return Values

`ippStsNoErr`	Indicates no error.

FullbandControllerGetSize_EC

Returns size of the fullband controller state structure.

Syntax

```
IppStatus ippsFullbandControllerGetSize_EC_32f(int frameSize, int tapLen,
    ippECFrequency sampleFreq, int* pSize);
IppStatus ippsFullbandControllerGetSize_EC_16s(int frameSize, Int tapLen,
    ippECFrequency sampleFreq, int* pSize);
```

Parameters

frameSize	Size of the frame.
tapLen	Number of tap values.
sampleFreq	Sample frequency.
pSize	Pointer to the computed buffer size value.

Description

The functions `ippsFullbandControllerGetSize_EC` are declared in the `ippsc.h` file. These functions return the sizes of memory buffers required for the functions `ippsFullbandControllerUpdate_EC` and `ippsFullbandController_EC` to operate. The appropriate memory must be allocated and initialized by calling the function `ippsFullbandControllerInit_EC` prior to using the `ippsFullbandControllerUpdate_EC`, `ippsFullbandControllerReset_EC` or `ippsFullbandController_EC` functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSize* pointer `NULL`.
`ippStsBadArgErr`	Indicates an error when *frameSize* is less than or equal to 0, or *tapLen* is less than or equal to 0.

FullbandControllerInit_EC

Initializes the fullband controller state structure.

Syntax

```
IppStatus ippsFullbandControllerInit_EC_32f(IppsFullbandControllerState_EC_32f*
    pState, int frameSize, int tapLen, ippECFrequency sampleFreq);
IppStatus ippsFullbandControllerInit_EC_16s(IppsFullbandControllerState_EC_16s*
    pState, int frameSize, int tapLen, ippECFrequency sampleFreq);
```

Parameters

pState	Pointer to the memory buffer required for the fullband controller state structure to be initialized. The size of the memory buffer must be computed by ippsFullbandControllerGetSize_EC function and returned via *pSize* parameter.
frameSize	Size of the frame.
tapLen	Number of tap values.
sampleFreq	Sample frequency.

Description

The functions ippsFullbandControllerInit_EC are declared in the ippsc.h file. These functions initialize the internal data of the fullband controller algorithm. The ippsFullbandControllerUpdate_EC and ippsFullbandController_EC functions use this memory in their operation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pState* pointer is NULL.
ippStsBadArgErr	Indicates an error when *frameSize* is less than or equal to 0, or *tapLen* is less than or equal to 0.

FullbandControllerUpdate_EC

Updates the fullband controller state and returns the step sizes.

Syntax

```
IppStatus ippsFullbandControllerUpdate_EC_32f(const Ipp32f* pSrcRin, const
    Ipp32f* pSrcSin, Ipp32f* pDstStepSize, IppsFullbandControllerState_EC_32f*
    pState);
IppStatus ippsFullbandControllerUpdate_EC_16s(const Ipp16s* pSrcRin, const
    Ipp16s* pSrcSin, IppAECScaled32s* pDstStepSize,
    IppsFullbandControllerState_EC_16s* pState);
```

Parameters

pState	Pointer to the fullband controller state structure.
pSrcRin	Pointer to the receive-in signal history. The history length is *tapLen* + *frameSize*, where *tapLen* and *frameSize* are specified in the function ippsFullbandControllerInit_EC
pSrcSin	Pointer to the send-in signal frame. The frame size is specified in the function ippsFullbandControllerInit_EC.
pDstStepSize	Pointer to the vector of step sizes. Vector length is equal to *frameSize* specified in the function ippsFullbandControllerInit_EC.

Description

The functions ippsFullbandControllerUpdate_EC are declared in the ippsc.h file. These functions update the internal state of fullband controller and return the step size values for use in the coefficients' adaptation process.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pState*, *pRin*, *pSin*, or *pStepSize* pointer is NULL.

FullbandController_EC

The main fullband controller function which updates filter coefficients and returns output gain coefficients.

Syntax

```
IppStatus ippsFullbandController_EC_32f(const Ipp32f* pAdaptiveFilterErr, const
    Ipp32f* pFixedFilterErr, Ipp32f* pAdaptiveCoefs, Ipp32f* pFixedCoefs,
    Ipp32f* pSGain, IppsFullbandControllerState_EC_32f* pState);
IppStatus ippsFullbandController_EC_16s(const Ipp16s* pAdaptiveFilterErr, const
    Ipp16s* pFixedFilterErr, Ipp16s* pAdaptiveCoefs, Ipp16s* pFixedCoefs,
    Ipp32s* pSGain, IppsFullbandControllerState_EC_16s* pState);
```

Parameters

pState	Pointer to the fullband controller state structure.
pAdaptiveFilterErr	Pointer to the vector of adaptive filter output. Vector length is equal to *frameSize*, specified in ippsFullbandControllerInit_EC.
pFixedFilterErr	Pointer to the vector of fixed filter output. Vector length is equal to *frameSize*, specified in ippsFullbandControllerInit_EC.
pAdaptiveCoefs	Pointer to the adaptive filter coefficients vectors. Vector length is equal to *tapLen*, specified in ippsFullbandControllerInit_EC.
pFixedCoefs	Pointer to the fixed filter coefficients vectors. Vector length is equal to *tapLen*, specified in ippsFullbandControllerInit_EC.
pSGain	Pointer to the send gain coefficient.

Description

The functions ippsFullbandController_EC are declared in the ippsc.h file.

These functions compare powers of adaptive filter error and fixed filter error and track changes of adaptive filter coefficients power.

If the adaptive filter has a significantly smaller error and is stable (which is indicated by coefficients power changing slowly), its coefficients are copied to the fixed filter. This case corresponds to “no double-talk” mode.

If the fixed filter has a significantly smaller error, its coefficients are copied to the adaptive filter. This case corresponds to the double-talk mode.

Send gain coefficient calculation (non-linear processor technology) is based on presence of the double-talk mode and changes of receive-in and send-in signal powers.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.

FullbandControllerReset_EC

Resets fullband controller state.

Syntax

```
IppStatus
    ippsFullbandControllerReset_EC_32f(IppsFullbandControllerState_EC_32f*
    pState);
IppStatus
    ippsFullbandControllerReset_EC_16s(IppsFullbandControllerState_EC_16s*
    pState);
```

Parameters

pState	Pointer to the fullband controller state structure.

Description

The functions ippsFullbandControllerReset_EC are declared in the ippsc.h file. These functions reset the fullband controller state. You can use these functions after an interruption in adaptation process occurred.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* pointer is NULL.

FIR_EC

Computes FIR filter results.

Syntax

```
IppStatus ippsFIR_EC_16s(const Ipp16s* pSrcSpchRef, const Ipp16s* pSrcSpch,
    Ipp16s* pDstSpch, int len, Ipp16s* pSrcTaps, Int tapsLen);
IppStatus ippsFIR_EC_32f(const Ipp32f* pSrcSpchRef, const Ipp32f* pSrcSpch,
    Ipp32f* pDstSpch, int len, Ipp32f* pSrcTaps, Int tapsLen);
```

Parameters

pSrcSpchRef	Pointer to the source original receive-out signal (R_{out}) of length (*tapsLen* + *len*) .
pSrcSpch	Pointer to the send-in signal (S_{in}) with echo path.
pDstSpch	Pointer to the destination send-out signal (S_{out}) which is echo-free.
len	Length of source and destination signals.
pSrcTaps	Vector of FIR filter taps .
tapsLen	Number of taps in the FIR filter.

Description

The functions `ippsFIR_EC` are declared in the `ippsc.h` file.

These function perform finite impulse response (FIR) filtering.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.

FIRSubband_EC, FIRSubbandLow_EC

Computes the frequency-domain adaptive filter output

Syntax

```
IppStatus ippsFIRSubband_EC_32fc(Ipp32fc** ppSrcSignalIn, Ipp32fc** ppSrcCoefs,
    Ipp32fc* pDstSignalOut, int numSegments, int len);
IppStatus ippsFIRSubband_EC_32sc_Sfs(Ipp32sc** ppSrcSignalIn, Ipp32sc**
    ppSrcCoefs, Ipp32sc* pDstSignalOut, int numSegments, int len, int
    scaleFactor);
IppStatus ippsFIRSubbandLow_EC_32sc_Sfs(const Ipp32sc** ppSrcSignal, const
    Ipp32sc** ppCoefs, int numSegments, Ipp32sc* pDstSignal, int startSubband,
    int numSubbands, int scaleFactor);
```

Parameters

ppSrcSignalIn, ppSrcSignal	Pointer to the two-dimensional vector of size [*numSegments*]*[*len*] containing the pointers to the most recent complex-valued FFT spectra in input audio signal.
ppSrcCoefs, *ppCoefs*	Pointer to the two-dimensional vector of size [*numSegments*]*[*len*] containing the pointers to the filter coefficients vector of size [*len*].
pDstSignalOut, *pDstSignal*	Pointer to the complex-valued filter output vector.
numSegments	Number of filter segments (0 < *numSegments* < 256).
len, numSubbands	Number of filter subbands (0 < *len*, *numSubbands* < 4097).
startSubband	Number of subbands to skip before filtering (0≤*startSubband* < *numSubbands*).
scaleFactor	Saturation fixed scale factor (-32 < *scaleFactor* < 32), for `ippsFIRSubbandLow_EC_32sc_Sfs` function (0 ≤*scaleFactor* < 32).

Description

The functions `ippsFIRSubband_EC` and `ippsFIRSubbandLow_EC` are declared in the `ippsc.h` file.These functions perform FIR filtering of the input two-dimensional spectra vector.

The `ippsFIRSubbandLow_EC_32sc_Sfs` function performs filtering without 64-bit integer overflow check. For *startSubband*=0, if no overflow occurs this function is equivalent to the function `ippsFIRSubband_EC_32sc_Sfs` with *len* equals to *numSubbands*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error when *len* has an illegal value.
`ippStsRangeErr`	Indicates an error when *numSegments*, *startSubband* or *scaleFactor* is out of the specified range.

FIRSubbandCoeffUpdate_EC, FIRSubbandLowCoeffUpdate_EC

Updates the adaptive filter coefficients

Syntax

```
IppStatus ippsFIRSubbandCoeffUpdate_EC_32fc_I(const double* pSrcStepSize, const
    Ipp32fc** ppSrcFilterInput, const Ipp32fc* pSrcError, Ipp32fc**
    ppSrcDstCoefs, int numSegments, int len);
IppStatus ippsFIRSubbandCoeffUpdate_EC_32sc_I(const Ipp32s_EC_Sfs*
    pSrcStepSize, const Ipp32sc** ppSrcFilterInput, const Ipp32sc* pSrcError,
    Ipp32sc** ppSrcDstCoefsQ15, int numSegments, int len, int scaleFactorCoef);
IppStatus ippsFIRSubbandLowCoeffUpdate_EC_32sc_I(const Ipp32sc**
    ppSrcFilterInput,  const Ipp32sc*  pSrcError, Ipp32sc** ppSrcDstCoefsQ15,
    int numSegments, Ipp32sc* pDstProdStepErrQ, const Ipp32s_EC_Sfs*
    pSrcAdaptStepSize,  int startSubband, int numSubbands, int scaleFactorCoef);
```

Parameters

pSrcStepSize, pSrcAdaptStepSize Pointer to the adaptive filter step size vector of size *len* or *numSubbands*. For the integer functions step size vector elements are represented as a scaled

	integer values where variable *x* of type `Ipp32s_EC_Sfs` corresponds to the floating point value `x.val*2^x.sf`.
ppSrcFilterInput	Pointer to the array of pointers to the most recent input blocks (for example, Xn, Xn-1, …, Xn-L+1). These are the complex-valued vectors that contain the FFT of the input signal. The dimension of *ppSrcFilterInput* is [*numSegments*]*[*len*] or [*numSegments*]*[*numSubbands*].
pSrcError	Pointer to the complex-valued vector containing the filter error. Its dimension is [*len*] or [*numSubbands*].
ppSrcDstCoefs, ppSrcDstCoefsQ15	Pointer to the array of pointers to the filter coefficient vectors. They are the complex-valued vectors containing the filter coefficients. The dimension of *ppSrcDstCoefs* is [*numSegments*]*[*len*] or [*numSegments*]*[*numSubbands*].
startSubband	Number of subbands to skip before filtering (0≤*startSubband* < *numSubbands*).
numSegments	Number of filter segments (L) (0 < *numSegments* < 256).
len, numSubbands	Number of adaptive filter subbands and length of the input and output vectors.
scaleFactorCoef	Fixed scale factor for filter coefficients (0< *scaleFactorCoef* < 32).
pDstProdStepErrQ	Pointer to the output vector of filter error and step size product.

Description

The functions `ippsFIRSubbandCoeffUpdate_EC` and `FIRSubbandLowCoeffUpdate_EC` are declared in the `ippsc.h` file. These functions update the adaptive filter coefficients according to the given step size and filter error independently in each frequency subband.

The function `ippsFIRSubbandLowCoeffUpdate_EC_32sc_I` computes the product between step size, error and coefficients without 64-bit integer overflow check. For *startSubband*=0, if no overflow occurs the function `ippsFIRSubbandLowCoeffUpdate_EC_32sc_I` is equivalent to `ippsFIRSubbandCoeffUpdate_EC_32sc_I` with *len* equal to *numSubbands*.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error when *len* has an illegal value.
ippStsRangeErr	Indicates an error when *pSrcStepSize*[i] < 0, or when *numSegments* or *scaleFactorCoef* is out of the specified range.

NLMS_EC

Performs FIR filtering with coefficients update.

Syntax

```
IppStatus ippsNLMS_EC_16s(const Ipp16s* pSrcSpchRef, const Ipp16s* pSrcSpch,
    const Ipp32s* pStepSize, Ipp16s* pSrcDstErr, Ipp16s* pDstSpch, int len,
    Ipp16s* pSrcDstTaps, int tapsLen);
IppStatus ippsNLMS_EC_32f(const Ipp32f* pSrcSpchRef, const Ipp32f* pSrcSpch,
    const Ipp32f* pStepSize, Ipp32f* pSrcDstErr,  Ipp32f* pDstSpch, int len,
    Ipp32f* pSrcDstTaps, int tapsLen);
```

Parameters

pSrcSpchRef	Pointer to the source original receive-out signal (R_{out}) of length (*tapsLen* + *len*) .
pSrcSpch	Pointer to the send-in signal (S_{in}) with echo path.
pStepSize	Pointer to the vector of step sizes. Vector length is equal to *len*.
pSrcDstErr	Pointer to the last error value.
pDstSpch	Pointer to the destination send-out signal (S_{out}) which is echo-free.
len	Length of source and destination signals.
pSrcDstTaps	Source and destination vector of FIR filter taps.
tapsLen	Number of taps.

Description

The functions ippsNLMS_EC are declared in the ippsc.h file.

These function perform FIR filtering with normalized Least Mean Square (LMS) coefficients adaptation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.

G722 Sub-Band ADPCM Speech Codec Related Functions

The Intel IPP functions described later in this section if properly combined may be used to construct the G.722 speech codec compliant to the reference implementation given by ITU-T library (Rec.192, Clause 7: *The ITU-T 64, 56 and 48 kbps wide-band speech coding algorithm*). The description of G.722 speech codec also known as Sub-Band ADPCM compression may be found in ITU-T G.722 Recommendation: *7KHz coding within 64 kbps*.

The Intel IPP functions described below are primarily concerned with the well-defined, computationally expensive core operations that comprise the G.722 subband codec.

The list of Intel IPP Subband ADPCM related functions is given in the following table:

Table 9-15 Intel IPP Subband ADPCM Related Functions

Function Base Name	Operation
SBADPCMEncodeStateSize_G722	Returns the memory size required for G.722 Sub-Band ADPCM encoder.
SBADPCMEncodeInit_G722	Initializes the memory buffer for the Sub-Band ADPCM encoder.
SBADPCMEncode_G722	Performs Sub-Band ADPCM encoding of the synthesis QMF samples.
QMFEncode_G722	Performs QMF analysis of the uniform PCM input.
ippsSBADPCMDecodeStateSize_G722	Returns the memory size required for Sub-Band ADPCM decoder.
SBADPCMDecodeInit_G722	Initializes the memory buffer for the Sub-Band ADPCM decoder.
SBADPCMDecode_G722	Performs decoding of the Sub-Band ADPCM compressed bit-stream.
QMFDecode_G722	Performs QMF synthesis of recovered samples.

SBADPCMEncodeStateSize_G722

Returns the memory size required for G.722 Sub-Band ADPCM encoder.

Syntax

```
IppStatus ippsSBADPCMEncodeStateSize_G722_16s (int* pEncMemSize)
```

Parameters

pEncMemSze	Pointer to the output memory size in bytes.

Description

The function `ippsSBADPCMEncodeStateSize_G722` is declared in the `ippsc.h` file.

This function gets the information about the amount of memory (in bytes) needed to process the G.722 Sub-Band ADPCM compression.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pEncMemSize* pointer is `NULL`.

SBADPCMEncodeInit_G722

Initializes the memory buffer for the Sub-Band ADPCM encoder.

Syntax

```
IppStatus ippsSBADPCMEncodeInit_G722_16s (IppsEncoderState_G722_16s* pEncMem)
```

Parameters

pEncMem	Pointer to the input memory buffer of size required to properly initialize the G.722 Sub-Band ADPCM encoder.

Description

The function `ippsSBADPCMEncodeInit_G722_16s` is declared in the `ippsc.h` file.

This function initializes the memory buffer referenced by the pointer *pEncMem* . This buffer is required to enable G.722 Sub-Band ADPCM compression (encoding) starting from the reset state.

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when the *pEncMem* pointer is NULL.

SBADPCMEncode_G722

Performs Sub-Band ADPCM encoding of the synthesis QMF samples.

Syntax

```
IppStatus ippsSBADPCMEncode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int
    len, IppsEncoderState_G722_16s* pEncMem)
```

Parameters

pSrc	Pointer to the input vector of synthesis QMF samples.
pDst	Pointer to the ADPCM bit-stream output vector.
len	The length of input vector, must be a multiple of two.
pEncMem	Pointer to the state memory of the Sub-Band ADPCM encoder.

Description

The function ippsSBADPCMEncode_G722_16s is declared in the ippsc.h file.

This function performs the Sub-Band ADPCM compression (encoding) of the synthesized Quadrature Mirror Filter (QMF) samples within a bit rate of 64 kbps. The function uses the state memory referenced by *pEncMem* to encode the input data and update it accordingly. Before using this function, you must initialize the state memory by calling the ippsSBADPCMEncodeInit_G722 function.
The function ippsSBADPCMEncode_G722_16s splits the frequency band of the input signal into two sub-bands (higher and lower) and encodes the signals in each sub-band using ADPCM compression. Each element of the output vector contains six binary digits of the lower sub-band and two binary digits of the higher sub-band for the 64 kbps bit rate ADPCM encoding.

Return Values

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error when *len* is less than or equal to 0.
ippStsSizeErr	Indicates an error when *len* is not a multiple of two.
IppStsNullPtrErr	Indicates an error when the *pMemUpdated*, *pSrc*, or *pDst* pointer is NULL.

QMFEncode_G722

Performs QMF analysis of the uniform PCM input.

Syntax

```
IppStatus ippsQMFEncode_G722_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s* delay)
```

Parameters

pSrc	Pointer to the uniform PCM input speech vector.
pDst	Pointer to the synthesis QMF samples output vector.
len	The length of input/output vectors, must be a multiple of two.
delay	Pointer to the buffer of a delay line.

Description

The function ippsQMFEncode_G722_16s is declared in the ippsc.h file.

This function uses Quadrature Mirror Filtering (QMF) of the 14-bit uniform PCM speech input to compute the lower and higher sub-band signal components for the Sub-Band ADPCM encoder. Computations are done in accordance with ITU-T Recommendation G.722.

Return Values

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error when *len* is less than or equal to 0.
ippStsSizeErr	Indicates an error when *len* is not a multiple of two.
IppStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *delay* pointer is NULL.

SBADPCMDecodeStateSize_G722

Returns the memory size required for Sub-Band ADPCM decoder.

Syntax

```
IppStatus ippsSBADPCMDecodeStateSize_G722_16s (int* pDecMemSize)
```

Parameters

pDecMemSize	Pointer to the output size of the memory needs to perform G.722 SB-ADPCM decoder.

Description

The function `ippsSBADPCMDecodeStateSize_G722` is declared in the `ippsc.h` file.

This function gets the information about the amount of memory (in bytes) needed to process the G.722 Sub-Band ADPCM decompression.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pDecMemSize* pointer is `NULL`.

SBADPCMDecodeInit_G722

Initializes the memory buffer for the Sub-Band ADPCM decoder.

Syntax

```
IppStatus ippsSBADPCMDecodeInit_G722_16s (IppsDecoderState_G722_16s* pDecMem)
```

Parameters

pDecMem	Pointer to the input memory buffer of size required to properly initialize the G.722 Sub-Band ADPCM decoder.

Description

The function `ippsSBADPCMDecodeInit_G722_16s` is declared in the `ippsc.h` file.

This function initializes the memory buffer referenced by the pointer *pDecMem* . This buffer is required to enable G.722 Sub-Band ADPCM decompression (decoding) starting from the reset state.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pDecMem* pointer is `NULL`.

SBADPCMDecode_G722

Performs decoding of the Sub-Band ADPCM compressed bit-stream.

Syntax

```
IppStatus ippsSBADPCMDecode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s mode, IppsDecoderState_G722_16s* pDecMem)
```

Parameters

pSrc	Pointer to the input vector that contains the even number of samples. The even element of this vector contains six binary digits of the lower sub-band and the odd element of this vector contains two binary digits of the higher sub-band.
pDst	Pointer to the output vector that contains decoded low-band and high-band portions of the recovered samples.
len	Length of the input vector.
mode	Decode bit rate of the G.722 decoder. The values of *mode* have the following meaning: *mode* = 1 - sets 64 kbps bit rate *mode* = 2 - sets 56 kbps bit rate *mode* = 3 - sets 48 kbps bit rate
pDecMem	Pointer to the state memory of the Sub-Band ADPCM decoder.

Description

The function `ippsSBADPCMDecode_G722_16s` is declared in the `ippsc.h` file.

This function decodes the lower-band and upper-band portions of the Sub-Band ADPCM compressed bit-stream into the low-band and upper-band portions of the recovered samples, respectively. This operation is the reverse of the Sub-Band ADPCM compression (see `ippsSBADPCMEncode_G722`). Note that the effective audio coding bit rate varies between 64, 56, or 48 kbps depending on the mode of operation.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsNullPtrErr`	Indicates an error when the *pDecMem* , *pSrc*, or *pDst* pointer is NULL.

QMFDecode_G722

Performs QMF synthesis of recovered samples.

Syntax

```
IppStatus ippsQMFDecode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s* delay);
```

Parameters

pSrc	Pointer to the input vector that contains decoded low-band and high-band portions of the recovered samples.
pDst	Pointer to the 16-bit linear PCM speech output vector.
len	The length of input/output vectors, must be a multiple of two.
delay	Pointer to the buffer of a delay line.

Description

The function `ippsQMFDecode_G722_16s` is declared in the `ippsc.h` file.

This function performs Quadrature Mirror Filter (QMF) synthesis of the low-band and high-band portions of the recovered samples to reconstruct the output signal. Computations are done in accordance with ITU-T Recommendation G.722.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when *len* is than less or equal to 0.
`ippStsSizeErr`	Indicates an error when *len* is not a multiple of two.
`IppStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst* or *delay* pointer is `NULL`.

Companding Functions

The functions described in this section perform an operation of data compression by using a logarithmic encoder-decoder, referred to as companding. Companding allows you to maintain a constant percentage error by logarithmically spacing the quantization levels [Rab78].

The Intel IPP companding functions perform the following conversion operations of signal samples:

- From 8-bit μ-law encoded format to PCM-linear or vice-versa.
- From 8-bit A-law encoded format to PCM-linear or vice-versa.
- From 8-bit μ-law encoded format to A-law encoded or vice-versa.

Samples encoded in μ-law or A-law format are non-uniformly quantized. The quantization functions used by these formats are designed to reduce the dependency of signal-to-noise ratio on the magnitude of the encoded signal. This is achieved by quantization (companding) at a finer resolution near zero, and at a coarse resolution at larger positive or negative levels. The output values are normalized to be in the range `[-1; +1]`.

These functions perform the μ-law and A-law companding in compliance with the CCITT G.711 specification. For the conversion rules and more details, refer to [CCITT].

Example 9-1 shows how to use companding functions.

MuLawToLin

Decodes samples from 8-bit μ-law encoded format to linear samples.

Syntax

```
IppStatus ippsMuLawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMuLawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector, which stores 8-bit μ-law encoded signal samples to be decoded.
pDst	Pointer to the destination vector, which stores the linear sample results.

len	Number of samples in the vector.

Description

The function `ippsMuLawToLin` is declared in the `ipps.h` file. This function decodes the 8-bit μ-law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

LinToMuLaw

Encodes the linear samples using 8-bit μ-law format and stores them in a vector.

Syntax

```
IppStatus ippsLinToMuLaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToMuLaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

Parameters

pSrc	Pointer to the vector that holds the signal samples (normalized to be less than 1.0) to be encoded.
pDst	Pointer to the vector that holds the output of the function `ippsLinToMuLaw`.
len	Number of samples in the vector.

Description

The function `ippsLinToMuLaw` is declared in the `ipps.h` file. This function encodes the PCM-linear samples in the vector *pSrc* using 8-bit μ-law format and stores them in the vector *pDst*.

The formula for μ-law companding is as follows:

$$|C_\mu(x)| = \frac{\ln(1 + 255 \cdot |x|)}{\ln(256)} \cdot 128, \quad -1 \le x \le 1$$

where x is the linear signal sample and $C_\mu(x)$ is the μ-law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

Application Notes

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of μ-law format is usually performed using look-up Tables 2a/G.711 and 2b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Example 9-1 shows how to use the function `ippsLinToMuLaw_32f8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

ALawToLin

Decodes the 8-bit A-law encoded samples to linear samples.

Syntax

```
IppStatus ippsALawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsALawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the vector that holds the signal samples to be converted.
pDst	Pointer to the vector that holds the output of the function `ippsALawToLin`.
len	Number of samples in the vector.

Description

The function `ippsALawToLin` is declared in the `ipps.h` file. This function decodes the 8-bit A-law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*. Example 9-1 shows how to use the function `ippsALawToLin_8u32f`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

LinToALaw

Encodes the linear samples using 8-bit A-law format and stores them in an array.

Syntax

```
IppStatus ippsLinToALaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToALaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

Parameters

pSrc	Pointer to the vector that holds the signal samples to be encoded.
pDst	Pointer to the vector that holds the output of the function `ippsLinToALaw`.
len	Number of samples in the vector.

Description

The function `ippsLinToALaw` is declared in the `ipps.h` file. This function encodes the PCM-linear samples in the vector *pSrc* using 8-bit A-law format and stores them in the vector *pDst*.

The formula for A-law companding is as follows:

$$|C_A(x)| = \begin{cases} \dfrac{87.56|x|}{1+\ln 87.56} \cdot 128, & 0 \le |x| \le \dfrac{1}{87.56} \\ \dfrac{1+\ln(87.56|x|)}{1+\ln 87.56} \cdot 128, & \dfrac{1}{87.56} < |x| \le 1 \end{cases},$$

where x is the linear signal sample and $C_A(x)$ is the A-law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

Application Notes

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of A-law format is usually performed using look-up Tables 1a/G.711 and 1b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

MuLawToALaw

Converts samples from 8-bit μ-law encoded format to 8-bit A-law encoded format.

Syntax

```
IppStatus ippsMuLawToALaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector, which stores 8-bit μ-law encoded signal samples.
pDst	Pointer to the destination vector, which stores the 8-bit A-law encoded samples.
len	Number of samples in the vector.

Description

The function `ippsMuLawToALaw` is declared in the `ipps.h` file. This function converts signal samples from 8-bit μ-law encoded format in the vector *pSrc* to 8-bit A-law encoded format and stores them in the vector *pDst*.

Application Notes

The conversion of μ-law format to A-law format is usually performed using look-up Table 3/G.711shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Example 9-1 shows how to use the function `ippsMuLawToALaw_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

ALawToMuLaw

Converts samples from 8-bit A-law encoded format to 8-bit μ-law encoded format.

Syntax

```
IppStatus ippsALawToMuLaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector, which stores 8-bit A-law encoded signal samples.
pDst	Pointer to the destination vector, which stores the 8-bit μ-law encoded samples.
len	Number of samples in the vector.

Description

The function `ippsMuLawToALaw` is declared in the `ipps.h` file. This function converts signal samples from 8-bit A-law encoded format in the vector *pSrc* to 8-bit μ-law format and stores them in the vector *pDst*.

Application Notes

The conversion of A-law format to μ-law format is usually performed using look-up Table 4/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 9-1 Using Companding Functions

```
void compand( void ) {
     Ipp32f x[4] = { 0.1f, 0.2f, 0.3f, 0.4f };
     Ipp8u m[4], a[4];
     ippsLinToMuLaw_32f8u( x, m, 4 );
     ippsMuLawToALaw_8u( m, a, 4 );
     ippsALawToLin_8u32f( a, x, 4 );
     // now x must be close to original
     printf_32f("x =", x, 4, ippStsNoErr);
}
Output:
    x =  0.099609 0.207031 0.304688 0.398438
```

RT Audio Related Functions

The Intel IPP functions described in this section if properly combined can be used to construct the speech codec compliant to the Microsoft's Real-Time (RT) Audio codec. The primitives are primarily designed to implement the well-defined, computationally expensive core operations that comprise the codec portion of the RT Audio system. The RT Audio codec comprises a fixed and variable rate algorithms intended for encoding 16-bit uniform PCM at the sampling rate of 8 and 16 KHz at bit rates 8800 and 18000 bps respectively.

The list of these functions is given in the following table:

Table 9-16 Intel MSRT Audio Related Functions

Function Base Name	Operation
LPCToLSP_RTA	Converts LP coefficients to LSP coefficients
LSPToLPC_RTA	Converts LSP coefficients to LP coefficients
LevinsonDurbin_RTA	Calculates LP coefficients from the autocorrelation coefficients
QMFGetStateSize_RTA	Calculates the size of the QMF filter state memory
QMFInit_RTA	Initializes the QMF filter state memory
QMFDecode_RTA	Performs QMF synthesis
QMFEncode_RTA	Performs QMF analysis
PostFilterGetStateSize_RTA	Calculates the size of the post filter state memory
PostFilterInit_RTA	Initializes the post filter state memory
PostFilter_RTA	Restores speech signal from the residual
AdaptiveCodebookSearch_RTA	Searches for the adaptive codebook index and the lag, and computes the adaptive vector
FixedCodebookSearch_RTA FixedCodebookSearchRandom_RTA	Searches for the fixed codebook vector
LSPQuant_RTA	Performs quantization of LSP coefficients
HighPassFilter_RTA	Performs high-pass filtering
BandPassFilter_RTA	Performs band pass filtering

LPCToLSP_RTA

Converts LP coefficients to LSP coefficients.

Syntax

```
IppStatus ippsLPCToLSP_RTA_32f(const Ipp32f* pSrcLPC, Ipp32f* pDstLSP, int
    order);
```

Parameters

pSrcLPC	Pointer to the input vector of length *order* that contains LP coefficients.
pDstLSP	Pointer to the output vector of length *order* that contains LSP coefficients.
order	Order of the LP filter (number of LP coefficients).

Description

The function `ippsLPCToLSP_RTA` is declared in the `ippsc.h` file.

This function converts LP coefficients to LSP coefficients.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`IppStsRangeErr`	Indicates an error when *order* is less than or equal to 0, or greater then 16, or is not even number.

LSPToLPC_RTA

Converts LSP coefficients to LP coefficients.

Syntax

```
IppStatus ippsLSPToLPC_RTA_32f(const Ipp32f* pSrcLSP, Ipp32f* pDstLPC, int
    order);
```

Parameters

pSrcLSP	Pointer to the input vector of length *order* that contains LSP coefficients.
pDstLPC	Pointer to the output vector of length *order* that contains LP coefficients.
order	Order of the LP filter (number of LP coefficients).

Description

The function `ippsLSPToLPC_RTA` is declared in the `ippsc.h` file.

This function converts LSP coefficients to LP coefficients.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`IppStsRangeErr`	Indicates an error when *order* is less than or equal to 0, or greater then 16, or is not even number.

LevinsonDurbin_RTA

Calculates LP coefficients from the autocorrelation coefficients.

Syntax

```
IppStatus ippsLevinsonDurbin_RTA_32f(const Ipp32f* pSrcAutoCorr, Ipp32f*
    pDstLPC, Ipp32f* pDstRC, int order);
```

Parameters

pSrcAutoCorr	Pointer to the autocorrelation vector.
pDstLPC	Pointer to the LPC output vector.
pDstRC	Pointer to the RC output vector.
order	Order of the LP filter.

Description

The function `ippsLevinsonDurbin_RTA` is declared in the `ippsc.h` file.

This function calculates linear prediction (LP) coefficients for the LP filter with the given *order* using the autocorrelation coefficients and Levinson-Durbin algorithm.

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
IppStsRangeErr	Indicates an error when *order* is less than or equal to 0.

QMFGetStateSize_RTA

Calculates the size of the QMF filter state memory.

Syntax

```
IppStatus ippsQMFGetStateSize_RTA_32f(int* pSize);
```

Parameters

pSize	Pointer to the size of the the QMF filter memory.

Description

The function ippsQMFGetStateSize_RTA is declared in the ippsc.h file.

This function calculates the size of the Quadrature Mirror Filter (QMF) structure that must be initialized by the function ippsQMFInit_RTA and used by the functions ippsQMFEncode_RTA and ippsQMFDecode_RTA.

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when *pSize* pointer is NULL.

QMFInit_RTA

Initializes the QMF filter state memory.

Syntax

```
IppStatus ippsQMFInit_RTA_32f(IppQMFState_RTA_32f* pQMFMem, int nTaps);
```

Parameters

pQMFMem	Pointer to the QMF filter state structure.
nTaps	Number of taps, possible values are: 24, 48.

Description

The function `ippsQMFInit_RTA` is declared in the `ippsc.h` file.

This function sets the number of filter taps *nTaps* and initializes Quadrature Mirror Filter (QMF) state memory *pQMFMem* that is used by the functions ippsQMFEncode_RTA and ippsQMFDecode_RTA . The size of the memory must be calculated by calling the function ippsQMFGetStateSize_RTA beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when *pQMFMem* pointer is `NULL`.
`IppStsRangeErr`	Indicates an error when *nTaps* is not equal to 24 or 48.

QMFDecode_RTA

Performs QMF synthesis.

Syntax

```
IppStatus ippsQMFDecode_RTA_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
    int len, Ipp32f* pDst, IppQMFState_RTA_32f* pQMFMem);
```

Parameters

pSrcLow	Pointer to the input low-band vector, its length is *len*.

pSrcHigh	Pointer to the input high-band vector, its length is *len*.
len	Number of samples in input vectors.
pDst	Pointer to the output restored vector, its length is 2*len.
pQMFMem	Pointer to the QMF filter state memory.

Description

The function ippsQMFDecode_RTA is declared in the ippsc.h file.

This function decodes the input vectors with QMF synthesis filter. The QMF filter state memory must be initializes by calling the function ippsQMFInit_RTA beforehand.

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
IppStsSizeErr	Indicates an error when *len* is less than or equal to 0, or is greater than 320.

QMFEncode_RTA

Performs QMF analysis.

Syntax

```
IppStatus ippsQMFEncode_RTA_32f(const Ipp32f* pSrc, int len, Ipp32f* pDstLow,
    Ipp32f* pDstHigh, IppQMFState_RTA_32f* pQMFMem);
```

Parameters

pSrc	Pointer to the input vector, its length is *len*.
len	Number of samples in input vectors.
pDstLow	Pointer to the output low-band vector, its length is *len*/2.
pDstHigh	Pointer to the output high-band vector, its length is *len*/2.
pQMFMem	Pointer to the QMF filter state memory.

Description

The function ippsQMFEncode_RTA is declared in the ippsc.h file.

This function encodes the input vectors with QMF analysis filter. The QMF filter state memory must be initializes by calling the function ippsQMFInit_RTA beforehand.

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
IppStsSizeErr	Indicates an error when *len* is less than or equal to 0, or is greater than 320, or is not even number.

PostFilterGetStateSize_RTA

Calculates the size of the short term post filter state memory.

Syntax

```
IppStatus ippsPostFilterGetStateSize_RTA_32f(int* pSize);
```

Parameters

pSize	Pointer to the size of the filter memory.

Description

The function ippsPostFilterGetStateSize_RTA is declared in the ippsc.h file.

This function calculates the size of post filter memory that is initialized by the function ippsPostFilterInit_RTA and used by the function ippsPostFilter_RTA.

Return Values

ippStsNoErr	Indicates no error.
IppStsNullPtrErr	Indicates an error when *pSize* pointer is NULL.

PostFilterInit_RTA

Initializes the post filter state memory.

Syntax

```
IppStatus ippsPostFilterInit_RTA_32f(IppPostFilterState_RTA_32s pStateMem);
```

Parameters

pStateMem	Pointer to the post filter state memory.

Description

The function `ippsPostFilterInit_RTA` is declared in the `ippsc.h` file.

This function initializes the post state memory *pStateMem*. The size of the memory must be calculated by calling the function `ippsPostFilterGetStateSize_RTA` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when *pStateMem* pointer is `NULL`.

PostFilter_RTA

Restores speech signal from the residual.

Syntax

```
IppStatus ippsPostFilter_RTA_32f_I(const Ipp32f* pSrcLPC, int order, Ipp32f*
    pSrcDstSpeech, int len, int pfType, Ipp32f* pMem[33],
    IppPostFilterState_RTA_32f* pStateMem);
```

Parameters

pSrcLPC	Pointer to the input vector with LPC coefficients.
order	Order of the LP filter.
pSrcDstSpeech	Pointer to the input/output speech vector.
len	Number of samples.

pfType	Specifies the type of thepost filter. Possible values: 0 - harmonic and full band coding; 1, 3, 4 - subband coding respectively; 2, 5, 6 - streaming coding respectively.
pMem	Pointer for the post filter memory vector.
pStateMem	Pointer for the post filter state memory.

Description

The function `ippsPostFilter_RTA` is declared in the `ippsc.h` file.

This function restores speech signal from the residual. The function updates *pMem* and *pStateMem* memory appropriately. The post filter state memory must be initializes by calling the function `ippsPostFilterInit_RTA` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`IppStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsRangeErr`	Indicates an error when *order* is less than or equal to 1, or greater than 16, or *pfType* is less than 0 or greater than 7.

AdaptiveCodebookSearch_RTA

Searches for the adaptive codebook index and the lag, and computes the adaptive vector.

Syntax

```
IppStatus ippsAdaptiveCodebookSearch_RTA_32f(const Ipp32f* pSrcAdptTarget,
    const Ipp32f* pSrcImpulseResponse, int subFrameSize, const Ipp32f*
    pSrcBoundary[4], Ipp32f* pSrcDstExc[372], Ipp32f* pSrcDstLag, int*
    pDstAdptIndex, Ipp32f* pDstAdptContribution, int deltaSearchRange, int
    searchFlag);
```

Parameters

pSrcAdptTarget	Pointer to the input target signal [*subFrameSize*].

pSrcImpulseResponse	Pointer to the input impulse response [*subFrameSize*].
subFrameSize	Size of the subframe.
pSrcBoundary	Pointer to the input adaptive codebook search boundary.
pSrcDstExc	Pointer to the input/output excitation buffer.
pSrcDstLag	Pointer to the input/output lag.
pDstAdptIndex	Pointer to the output adaptive codebook search index.
pDstAdptContribution	Pointer to the output adaptive codebook search contribution [*subFrameSize*].
deltaSearchRange	Delta search range value.
searchFlag	Search flag, possible values: 0 - full search. 1 - search around open-loop pitch.

Description

The function `ippsAdaptiveCodebookSearch_RTA` is declared in the `ippsc.h` file.

This function searches for the adaptive codebook index and the lag, using the target signal and the past filtered excitation, and computes the adaptive vector. The result is applied to the subframe.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`IppStsSizeErr`	Indicates an error when *subFrameSize* is less than or equal to 0; or *searchFlag* is not equal to 0 or 1.

FixedCodebookSearch_RTA FixedCodebookSearchRandom_RTA

Searches for the fixed codebook vector.

Syntax

```
IppStatus ippsFixedCodebookSearch_RTA_32f(const Ipp32f* pSrcFixedTarget, const
    Ipp32f* pSrcImpulseResponse, Ipp32f* pDstFixedVector, int subFrameSize, int*
    pDstFixedIndex, Ipp32f* pDstGain, int numPulses,int complexity);
```

```
IppStatus ippsFixedCodebookSearchRandom_RTA_32f(const Ipp32f* pSrcFixedTarget,
    const Ipp32f* pSrcImpulseResponse, Ipp32f* pDstFixedVector, int
    subFrameSize, int* pDstFixedIndex, Ipp32f* pDstGain, int codebookSize, int
    nStage, int gainLimit);
```

Parameters

pSrcFixedTarget	Pointer to the input target signal [*subFrameSize*].
pSrcImpulseResponse	Pointer to the input impulse response [*subFrameSize*].
pDstFixedVector	Pointer to the output fixed codebook vector [*subFrameSize*].
subFrameSize	Size of the subframe.
pDstFixedIndex	Pointer to the output fixed codebook search index value.
pDstGain	Pointer to the output optimal gain value.
numPulses	Number of pulses.
complexity	Complexity control parameter, possible values are: 0, 1, 2, 3.
codebookSize	Size of the random codebook, possible values: 64, 160, 256.
nStage	Number of stage, possible values: 1, 2, 2.
gainLimit	Flag that specifies the gain constrain limit, possible values: 0, 1, 2, 3, 4.

Description

The functions `ippsFixedCodebookSearch_RTA` and `ippsFixedCodebookSearchRandom_RTA` are declared in the `ippsc.h` file.

These functions search for the fixed-codebook vector and corresponding vector index in the fixed and random fixed codebooks respectively.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`IppStsSizeErr`	Indicates an error when *subFrameSize* is less than or equal to 0, is greater than 80.
`IppStsRangeErr`	Indicates an error when *complexity* is less than 0 or greater than 3; or *nStage* is not equal to 0, 1, or 2; or *gainLimit* is not equal to 0, 1, 2, 3, 4; or *codebookSize* is less than 1 or greater than 256; or *numPulses* is less than 5 or greater than 10.

LSPQuant_RTA

Performs quantization of LSP coefficients.

Syntax

```
IppStatus ippsLSPQuant_RTA_32f(const Ipp32f* pSrcLSP, const Ipp32f* pSrcLPC,
    Ipp32f* pDstLSP, int order, int* pDstLSPIndex, IppPCMFrequency frequency,
    int complexity);
```

Parameters

pSrcLSP	Pointer to the input unquantized LSP vector.
pSrcLPC	Pointer to the input unquantized LPC vectorþ
pDstLSP	Pointer to the output quantized LSP vector.
order	LP coefficients filter order, possible values are: 4, 6, 8, 10.
pDstLSPIndex	Pointer to the output vector of quantized LSP indexes.
frequency	Sampling frequency, possible values are:
	`IPP_PCM_FREQ_8000` 8000 Hz,
	`IPP_PCM_FREQ_16000` 16000 Hz,
	`IPP_PCM_FREQ_22050` 22050 Hz,
	`IPP_PCM_FREQ_32000` 32000 Hz.
complexity	Complexity control parameter, possible values are: 0, 1, 2, 3.

Description

The function `ippsLSPQuant_RTA` is declared in the `ippsc.h` file.

This function performs quantization of LSP coefficients.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`IppStsRangeErr`	Indicates an error when *complexity* is less than 0 or greater than 3; or *order* is not equal to 4, 6, 8, 10; or *frequency* is not equal to the possible values.

HighPassFilter_RTA

Performs high-pass filtering.

Syntax

```
IppStatus ippsHighPassFilter_RTA_32f(Ipp32f* pSrc, Ipp32f* pDst, int len, int
    hpfOrder, IppPCMFrequency frequency, Ipp32f* pMem);
```

Parameters

pSrc	Pointer to the source vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Number of elements in the source and destination vectors.
hpfOrder	Filter order, possible values: 4, 5.
frequency	Sampling frequency, possible values are:
	`IPP_PCM_FREQ_8000` 8000 Hz,
	`IPP_PCM_FREQ_16000` 16000 Hz,
	`IPP_PCM_FREQ_22050` 22050 Hz,
	`IPP_PCM_FREQ_32000` 32000 Hz.
pMem	Pointer to the memory allocated for the filter [*hpfOrder*].

Description

The function `ippsHighPassFilter_RTA` is declared in the `ippsc.h` file.

This function performs high-pass filtering and may be applied to the input or synthesized audio signal.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`IppStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsRangeErr`	Indicates an error when *hpfOrder* is not equal to 4 or 5; or *frequency* is not equal to the possible values.

BandPassFilter_RTA

Performs band pass filtering.

Syntax

```
IppStatus ippsBandPassFilter_RTA_32f_I(Ipp32f* pSrcDst, int len, Ipp32f* pMem);
```

Parameters

pSrcDst	Pointer to the source and destination vector [*len*].
len	Number of elements in the source and destination vectors.
pMem	Pointer to the memory allocated for the band pass filter [*len*+2].

Description

The function `ippsBandPassFilter_RTA` is declared in the `ippsc.h` file.

This function performs frequency enhancement around half of the bandwidth by cascading lowpass and highpass first order filters with the unit gain at stop band, and maximum gain ~ 3.56 db at about 8 kHz with 16kHz sampling rate.

Return Values

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`IppStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Audio Coding Functions

10

Intel® IPP for audio coding includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio encoder and decoder (see [ISO14496]), MP3 encoder and decoder. These functions implement pipeline blocks with large computational complexity.

The current set of functions is sufficient to implement a portable optimized MPEG-4 AAC Main profile decoder and a portable optimized MPEG-1, 2 Layer III encoder (see [ISO11172] and [ISO13818]).

Also this chapter includes Spectral Band Replication (SBR) functions as an audio coding bandwidth extension technology, which is standardized in ISO/IEC 14496-3:2001/Amd.1:2003 [ISO14496A].

The full list of functions is given in Table 10-1.

Table 10-1 Intel IPP Audio Coding Functions

Function Base Name	Operation
Interleaved to Multi-Row Format Conversion Functions	
`Interleave`	Converts signal from non-interleaved to interleaved format.
`Deinterleave`	Converts signal from interleaved to non-interleaved format.
Spectral Data Prequantization Functions	
`Pow34`	Raises a vector to the power of 3/4.
`Pow43`	Raises a vector to the power of 4/3.
`Pow43Scale`	Raises a vector to the power of 4/3 with scaling.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
Scale Factors Calculation Functions	
CalcSF	Restores actual scale factors from the bit stream values.
Mantissa Conversion and Scaling Functions	
Scale	Applies scale factors to spectral bands in accordance with spectral bands boundaries.
MakeFloat	Converts mantissa and exponent arrays to float arrays.
Modified Discrete Cosine Transform Functions	
MDCTFwdInitAlloc, MDCTInvInitAlloc	Create and initialize modified discrete cosine transform specification structure.
MDCTFwdInit, MDCTInvInit	Initialize modified discrete cosine transform specification structure.
MDCTFwdFree, MDCTInvFree	Close modified discrete cosine transform specification structure.
MDCTFwdGetSize, MDCTInvGetSize	Get the size of MDCT specification structure, initialization, and work buffer.
MDCTFwdGetBufSize, MDCTInvGetBufSize	Get the size of MDCT work buffer.
MDCTFwd, MDCTInv	Compute forward or inverse MDCT of a signal.
Block Filtering Functions	
FIRBlockInitAlloc	Initializes FIR block filter state.
FIRBlockFree	Closes FIR block filter state.
FIRBlockOne	Filters vector of sample through FIR block filter.
Frequency Domain Prediction Functions	
FDPInitAlloc	Creates and initializes predictor state.
FDPInit	Initializes FDP state.
FDPFree	Closes FDP state.
FDPGetSize	Gets size of FDP state structure.
FDPReset	Resets predictors for all spectral lines.
FDPResetSfb	Resets predictor-specific information in some scale factor bands.
FDPResetGroup	Resets predictors for group of spectral lines.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
FDPFwd	Performs frequency domain prediction procedure and calculates prediction error.
FDPInv	Retrieves input signal from prediction error, using frequency domain prediction procedure.
VLC Functions	
VLCDecodeEscBlock_MP3	Parses the bitstream and decodes variable length code for MP3 using signed VLC tables.
VLCDecodeEscBlock_AAC	Parses the bitstream and decodes variable length code for AAC using signed VLC tables.
VLCDecodeUTupleEscBlock_MP3	Parses the bitstream and decodes variable length code for MP3 using unsigned multi-tupled VLC tables.
VLCDecodeUTupleEscBlock_AAC	Parses the bitstream and decodes variable length code for AAC using unsigned multi-tupled VLC tables.
VLCCountEscBits_MP3	Calculates the number of bits necessary for encoding in MP3 format.
VLCCountEscBits_AAC	Calculates the number of bits necessary for encoding in AAC format.
VLCEncodeEscBlock_MP3	Encodes an array of values into destination bitstream in MP3 format and advances bitstream pointer.
VLCEncodeEscBlock_AAC	Encodes an array of values into destination bitstream in AAC format and advances bitstream pointer.
Psychoacoustic Functions	
Spread	Computes spreading function.
Vector Quantization Functions	
VQCodeBookInitAlloc	Creates and initializes the codebook structure.
VQCodeBookInit	Initializes the codebook structure.
VQCodeBookFree	Closes the `IppsVQCodeBookState_32f` structure created by the function `VQCodeBookInitAlloc`.
VQCodeBookGetSize	Gets size of the codebook structure.
VQPreliminarySelect	Selects candidates for the nearest code vector of codebooks.
VQMainSelect	Finds optimal indexes with minimal distortion.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
VQIndexSelect	Finds optimal vector set for specified number of codebooks.
VQReconstruction	Reconstructs vectors from indexes.
MP3 Encoder Functions	
AnalysisPQMF_MP3	Implements stage 1 of MP3 hybrid analysis filterbank.
AnalysisFilterInit_PQMF_MP3	Initializes PQMF MP3 analysis specification structure.
AnalysisFilterInitAlloc_PQMF_MP3	Allocates memory for PQMF MP3 analysis specification structure and initializes it.
AnalysisFilterGetSize_PQMF_MP3	Returns size of PQMF MP3 analysis specification structure.
AnalysisFilterFree_PQMF_MP3	Frees memory allocated for PQMF MP3 analysis specification structure.
AnalysisFilter_PQMF_MP3	Tranforms PQMF MP3-processed subband signals into time domain samples.
MDCTFwd_MP3	Implements stage 2 of the MP3 hybrid analysis filterbank.
PsychoacousticModelTwo_MP3	Implements the ISO/IEC 11172-3 psych-acoustic model recommendation 2 to estimate the masked threshold and perceptual entropy associated with a block of PCM audio input.
JointStereoEncode_MP3	Transforms independent left and right channel spectral coefficient vectors into combined mid/side (MS) and/or intensity (IS) mode coefficient vectors suitable for quantization.
Quantize_MP3	Quantizes the spectral coefficients generated by the analysis filterbank.
PackScaleFactors_MP3	Applies noiseless coding to the scale factors and then packs the output into the bitstream buffer.
HuffmanEncode_MP3	Applies lossless Huffman encoding to the quantized samples and packs the output into the bitstream buffer.
PackFrameHeader_MP3	Packs the content of the frame header into the bitstream.
PackSideInfo_MP3	Packs the side information into the bitstream buffer.
BitReservoirInit_MP3	Initializes all elements of the bit reservoir state structure.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
MP3 Decoder Functions	
UnpackFrameHeader_MP3	Unpacks the audio frame header.
UnpackSideInfo_MP3	Unpacks the side information from the input bitstream for use during the decoding of the associated frame.
UnpackScaleFactors_MP3	Unpacks scale factors.
HuffmanDecode_MP3 HuffmanDecodeSfb_MP3 HuffmanDecodeSfbMbp_MP3	Decodes Huffman data.
ReQuantize_MP3 ReQuantizeSfb_MP3	Requantizes the decoded Huffman symbols.
MDCTInv_MP3	Performs the first stage of hybrid synthesis filter bank.
SynthPQMF_MP3	Performs the second stage of hybrid synthesis filter bank.
SynthesisFilterInit_PQMF_MP3	Initializes PQMF MP3 synthesis specification structure.
SynthesisFilterInitAlloc_PQMF_MP3	Allocates memory for PQMF MP3 synthesis specification structure and initializes it.
SynthesisFilterGetSize_PQMF_MP3	Returns size of PQMF MP3 synthesis specification structure.
SynthesisFilterFree_PQMF_MP3	Frees memory allocated for PQMF MP3 synthesis specification structure.
SynthesisFilter_PQMF_MP3	Transforms PQMF MP3-processed subband signals into time domain samples.
AAC Decoder Functions	
UnpackADIFHeader_AAC	Gets the AAC ADIF format header.
UnpackADTSFrameHeader_AAC	Gets ADTS frame header from the input bitstream.
DecodePrgCfgElt_AAC	Gets program configuration element from the input bitstream.
DecodeChanPairElt_AAC	Gets *channel_pair_element* from the input bitstream.
NoiselessDecoder_LC_AAC	Decodes all the data for one channel.
DecodeDatStrElt_AAC	Gets data stream element from the input bitstream.
DecodeFillElt_AAC	Gets the fill element from the input bitstream.
QuantInv_AAC	Performs inverse quantization of Huffman symbols for current channel in-place.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
DecodeMsStereo_AAC	Processes MS stereo for pair channels in-place.
DecodeIsStereo_AAC	Processes intensity stereo for pair channels.
DeinterleaveSpectrum_AAC	Deinterleaves the coefficients for short block.
DecodeTNS_AAC	Decodes for Temporal Noise Shaping in-place.
MDCTInv_AAC_32s16s	Maps the time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples.
MDCTInv_AAC_32s_I	Computes inverse modified discrete cosine transform (MDCT), windowing and overlapping of signals.
DecodeMainHeader_AAC	Gets main header information and main layer information from bit stream.
DecodeExtensionHeader_AAC	Get extension header information and extension layer information from bit stream.
DecodePNS_AAC	Implements perceptual noise substitution coding within an ICS.
DecodeMsPNS_AAC	Implements perceptual noise substitution coding in the case of joint coding.
DecodeChanPairElt_MP4_AAC	Gets *channel_pair_element* from the input bitstream
LongTermReconstruct_AAC	Uses Long Term Reconstruct to reduce the redundancy of a signal between successive coding frames.
MDCTFwd_AAC_32s	Generates spectrum coefficient of PCM samples.
MDCTFwd_AAC_32s_I	Computes forward modified discrete cosine transform (MDCT) of windowed signals.
EncodeTNS_AAC	Performs reversion of TNS in the Long Term Reconstruct loop in-place.
LongTermPredict_AAC	Gets the predicted time domain signals in the Long Term Reconstruct (LTP) loop.
NoiselessDecode_AAC	Performs noiseless decoding.
LtpUpdate_AAC	Performs required buffer update in the Long Term Reconstruct (LTP) loop.
SBR Audio Encoder Functions	
DetectTransient_SBR	Estimates stationarity of signal by calculating deviation of spectrum audio signal.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
EstimateTNR_SBR	Measures Tonality-to-Noise Ratio of complex QMF subband samples.
AnalysisFilterEncGetSize_SBR	Returns size of analysis FilterSpec_SBR specification structures, init and work buffers.
AnalysisFilterEncInit_SBR	Initializes analysis specification structure.
AnalysisFilterEnc_SBR	Performs subband filtering of the input audio signal.
AnalysisFilterEncInitAlloc_SBR	Creates and initializes analysis specification structure.
AnalysisFilterEncFree_SBR	Closes analysis filter specification structure.
SBR Audio Decoder Functions	
AnalysisFilterInitAlloc_SBR	Allocates memory and initializes analysis SBR specification structure for real and complex signals.
SynthesisFilterInitAlloc_SBR	Allocates memory and initializes synthesis SBR specification structure for real and complex signals.
SynthesisDownFilterInitAlloc_SBR	Allocates memory and initializes downsample synthesis SBR specification structure for real and complex signals.
AnalysisFilterFree_SBR	Closes analysis SBR specification structure for real and complex signals.
SynthesisFilterFree_SBR	Closes synthesis SBR specification structure for real and complex signals.
SynthesisDownFilterFree_SBR	Closes downsample synthesis SBR specification structure for real and complex signals.
AnalysisFilterGetSize_SBR	Returns size of analysis FilterSpec_SBR specification structures, init and work buffers.
SynthesisFilterGetSize_SBR	Returns size of synthesis FilterSpec_SBR specification structures, init and work buffers.
SynthesisDownFilterGetSize_SBR	Returns size of downsample synthesis FilterSpec_SBR specification structures, init and work buffers.
AnalysisFilterInit_SBR	Initializes analysis specification structure.
SynthesisFilterInit_SBR	Initializes synthesis specification structure.
SynthesisDownFilterInit_SBR	Initializes downsample synthesis specification structure.
AnalysisFilter_SBR	Transforms time domain signal output from the core decoder into frequency subband signals.

Table 10-1 Intel IPP Audio Coding Functions (continued)

Function Base Name	Operation
SynthesisFilter_SBR	Transforms SBR-processed subband signals into time domain samples.
SynthesisDownFilter_SBR	Transforms SBR-processed subband signals into time domain samples and performs downsampling at the same time.
PredictCoef_SBR	Obtains prediction filter coefficients using covariance method.
PredictOneCoef_SBR	Obtains one prediction filter coefficient using covariance method
DTS Audio Coding Functions	
SynthesisFilterInit_DTS	Initializes DTS synthesis filter specification structure.
SynthesisFilterInitAlloc_DTS	Allocates memory for DTS synthesis filter specification structure and initilaizes it.
SynthesisFilterGetSize_DTS	Returns size of DTS synthesis filter specification structure.
SynthesisFilterFree_DTS	Frees memory allocated for DTS synthesis filter specification structure.
SynthesisFilter_DTS	Transforms QMF DTS-processed subband signals into time domain samples.

Interleaved to Multi-Row Format Conversion Functions

This section describes functions that enable you to perform transformations between interleaved and non-interleaved forms of multichannel signal. A signal in the interleaved form is a vector with interleaved samples for different channels. In the non-interleaved form, samples for each channel are stored in a separate vector.

NOTE. Although you may use both aligned and unaligned memory for arrays, you should expect slower performance when memory is not aligned.

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm .

Interleave

Converts signal from non-interleaved to interleaved format.

Syntax

```
IppStatus ippsInterleave_16s(const Ipp16s** pSrc, int ch_num, int len,
    Ipp16s* pDst);
IppStatus ippsInterleave_32f(const Ipp32f** pSrc, int ch_num, int len,
    Ipp32f* pDst);
```

Parameters

pSrc	Array of pointers to the vectors [*len*] containing samples for particular channels.
ch_num	Number of channels.
len	Number of samples in each channel.
pDst	Pointer to the destination vector [*ch_num* * *len*] in interleaved format.

Description

This function is declared in the `ippac.h` header file. The function `ippsInterleave` transforms the signal from the non-interleaved to interleaved format according to the formula

$$pDst[i] = pSrc[i \text{ div } ch_num][i \bmod ch_num],\ 0 \le i < ch_num * len,$$

where `div` is the integer part of the quotient and `mod` is the remainder.

See Example 10-1 of using `ippsInterleave_16s` function.

Example 10-1 `ippsInterleave_16s` Usage

```
IppStatus interleave(void)
{
  Ipp16s i0[5] = {0,1,2,3,4};
  Ipp16s i1[5] = {10,11,12,13,14};
  Ipp16s pDst[10];
  Ipp16s *pSrc[2];
  IppStatus st;
  int i;

  pSrc[0] = i0; pSrc[1] = i1;
  st = ippsInterleave_16s(pSrc, 2, 5, pDst);

  printf("\n pDst = ");
  for (i = 0; i < 10; i++)
    printf("%i ", pDst[i]);
  printf("\n");
  return st;
}

//Output:
//  pDst = 0 10 1 11 2 12 3 13 4 14
```

The function `ippsInterleave_32f` is used in the float-point version of MP3 encoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsNumChannelsErr`	Indicates an error when *ch_num* is less than or equal to 0.

ippStsMisalignedBuf	Indicates misaligned data. Supply aligned data for better performance.

Deinterleave

Converts signal from interleaved to non-interleaved format.

Syntax

```
IppStatus ippsDeinterleave_16s(const Ipp16s* pSrc, int ch_num, int len,
    Ipp16s** pDst);
IppStatus ippsDeinterleave_32f(const Ipp32f* pSrc, int ch_num, int len,
    Ipp32f** pDst);
```

Parameters

pSrc	Pointer to vector [*ch_num* * *len*] of interleaved samples.
ch_num	Number of channels.
len	Number of samples in each channel.
pDst	Array of pointers to the vectors [*len*] to be filled with samples of particular channels.

Description

This function is declared in the `ippac.h` header file. The function `ippsDeinterleave` transforms the input signal from interleaved to non-interleaved format according to the formula

$pDst[i][j] = pSrc[i + j * ch_num], 0 \le i < ch_num, 0 \le j < len.$

See Example 10-2 of using `ippsDeinterleave_16s` function.

Example 10-2 `ippsDeinterleave_16s` Usage

```
IppStatus deinterleave(void)
{
  Ipp16s i0[5];
  Ipp16s i1[5];
  Ipp16s pSrc[10] = {0,1,2,3,4,5,6,7,8,9};
  Ipp16s *pDst[2];
  IppStatus st;
  int i;
```

Example 10-2 ippsDeinterleave_16s Usage (continued)

```
  pDst[0] = i0; pDst[1] = i1;
  st = ippsDeinterleave_16s(pSrc, 2, 5, pDst);

  printf("\n pDst[0] = ");
  for (i = 0; i < 5; i++)
    printf("%i ", pDst[0][i]);
  printf("\n pDst[1] = ");

  for (i = 0; i < 5; i++)
    printf("%i ", pDst[1][i]);
  printf("\n");
  return st;
}

//Output:
//  pDst[0] = 0 2 4 6 8
//  pDst[1] = 1 3 5 7 9
```

The function ippsInterleave_16s is used in the AAC and MP3 encoders and the function ippsInterleave_32f is used in the AC3 decoder (float-point version), and AAC and MP3 encoders (float-point version) included into IPP Samples. See introduction to this section.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* or *ch_num* is less than or equal to 0.
ippStsMisalignedBuf	Indicates misaligned data. Supply aligned data for better performance.

Spectral Data Prequantization Functions

MPEG-1, 2 Layer III and MPEG-4 AAC audio encoders raise the spectral data to the power of 3/4 before quantization to provide a more consistent signal-to-noise ratio over the range of quantized values. The re-quantizers in the decoders linearize the values by raising their output to the power of 4/3.

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from
http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm .

Pow34

Raises a vector to the power of 3/4.

Syntax

```
IppStatus ippsPow34_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int len);
IppStatus ippsPow34_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPow34_16s_Sfs(const Ipp16s* pSrc, int inscaleFactor, int
    Ipp16s* pDst, int scaleFactor, int len);
```

Parameters

pSrc	Pointer to the input data vector [*len*].
inscaleFactor	Input data scalefactor value.
pDst	Pointer to the output data vector [*len*].
scaleFactor	Scalefactor value.
len	Number of elements in the input and output vectors.

Description

This function is declared in the `ippac.h` header file. The function `ippsPow34` performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = |pSrc[i]|^{\frac{3}{4}}, \; 0 \le i < len$$

and stores the result in *pDst*.

For example, in MPEG-1 Layer III the quantization of the complete vector of spectral values is done according to the following formula:

$$ix(i) = nint\left(\left(\frac{|xr(i)|}{\sqrt[4]{2^{qquant+quantanf}}}\right)^{0.75} - 0.0946\right)$$, where

ix is the array of quantized values, *i* is the number of values in the array, *nint* is the function used to round non-integer values to the nearest integer value, *xr* is the vector of the magnitudes of the spectral values, *qquant* is the quantizer step size information, and *quantanf* is a constant that depends on the spectral flatness measure.

You can use the function `ippsPow34_32f16s` for this operation. However, if the maximum of all quantized values is outside the table range or the overall bit sum exceeds the available bits, you should repeat this operation several times with a new *qquant* value and the same *xr* array.

Alternatively, you can use the function `ippsPow34_32f` to calculate the power of 3/4 for *xr* only once before the inner iteration loop and use it during every quantization iteration with the constant multiplier and convert the resulting values to `short`.

$$multiplier = \left(\frac{1}{\sqrt[4]{2^{qquant+quantanf}}}\right)^{0.75}.$$

This operation is supported by the function `ippsMulC_Low_32f16s`.

See Example 10-3 of using `ippsPow34_32f` and `ippsPow34_16s_Sfs`.

Example 10-3 `ippsPow34` Usage

```
IppStatus pow34(void)
{
  Ipp32f pSrc[5] = {-1234.56, 0, 45.89, -3.896, 45328.562};
  Ipp32f pDst[5], pDst32[5];
  Ipp16s pSrc16[5], pDst16[5];
  int inscaleFactor = 1;
  int scaleFactor = -3;
  IppStatus st;
  int i;

 st = ippsPow34_32f(pSrc, pDst, 5);
  if (st < ippStsOk) return st;
  printf("\n pDst = ");
  for (i = 0; i < 5; i++)
    printf("%f ", pDst[i]);
  printf("\n");
```

Example 10-3 `ippsPow34` Usage (continued)

```
  st = ippsConvert_32f16s_Sfs(pSrc, pSrc16, 5, ippRndNear,
inscaleFactor);
  if (st < ippStsOk) return st;
  st = ippsPow34_16s_Sfs(pSrc16, inscaleFactor, pDst16, scaleFactor, 5);
  if (st < ippStsOk) return st;
  st = ippsConvert_16s32f_Sfs(pDst16, pDst32, 5, -scaleFactor);

  printf("Before scaling\n pDst16 = ");
  for (i = 0; i < 5; i++)
    printf("%i ", pDst16[i]);
  printf("\n");

  printf("After scaling\n pDst16 = ");
  for (i = 0; i < 5; i++)
    printf("%f ", pDst32[i]);
  printf("\n");

  return st;
}

//Output:
//   pDst = 208.273575 0.000000 17.631470 2.773092 3106.554443
//   Before scaling
//   pDst16 = 1666 0 141 23 24853
//   After scaling
//   pDst16 = 208.250000 0.000000 17.625000 2.875000 3106.625000
```

The function `ippsPow34` is used in the float-point versions of AAC and MP3 encoders included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsMisalignedBuf`	Indicates misaligned data. Supply aligned data for better performance.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Pow43

Raises a vector to the power of 4/3.

Syntax

```
IppStatus ippsPow43_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
```

Parameters

pSrc	Pointer to the input data vector [*len*].
pDst	Pointer to the output data vector [*len*].
len	Number of elements in the input and output vectors.

Description

This function is declared in the `ippac.h` header file. The function `ippsPow43` performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = sign(pSrc[i])*|pSrc[i]|^{\frac{4}{3}}, 0 \le i < len$$

and stores the result in *pDst*.

The function `ippsPow43` is used in the float-point versions of AAC and MP3 decoders included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsMisalignedBuf`	Indicates misaligned data. Supply aligned data for better performance.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

NOTE. The input values in the *pSrc* array should not be greater than 8206. The function does not check if the input values meet this requirement in order not to cause performance slowdown. If the input values in the array exceed the limit, the function may operate incorrectly. No error code is returned in this case.

Pow43Scale

Raises a vector to the power of 4/3 with scaling.

Syntax

```
IppStatus ippsPow43Scale_16s32s_Sf(const Ipp16s* pSrc, Ipp32f* pDst,
    const Ipp16s *pScaleFactor, const Ipp32s *pBandOffset, int offset,
    int bandsNumber, int groupLen, int scalef);
```

Parameters

pSrc	Pointer to the input data vector.
pDst	Pointer to the output data vector.
pScaleFactor	Pointer to the data array containing scale factors. The size of the array must be not less than *bandsNumber*.
pBandOffset	Pointer to the vector of band offsets. The size of array must be not less than *bandsNumber* + 1.
offset	Scale factors offset.
bandsNumber	Number of bands to which scale factors are applied.
groupLen	Number of windows in the current group.
scalef	Scale factor of the output data.

Description

This function is declared in the `ippac.h` header file. The `ippsPow43Scale` function performs the following operation:

$$pDst[i] = pSrc[i]^{\frac{4}{3}} * 2^{\frac{1}{4}(pScalefactor[band] - offset)}.$$

This function does not perform any saturation so the user should consider overflow possibility. Absolute values of *pSrc*[i] should be less than 2^{14}.

See Example 10-4 of using `ippsPow43Scale` function.

Example 10-4 `ippsPow43Scale` Usage

```
#undef SF_OFFSET
#define SF_OFFSET 100

IppStatus pow43scale(void)
{
  Ipp16s pSrc[40];
  Ipp32s pDst0[40], pDst1[40];
  Ipp16s pScaleFactor[5] = {65, 128, 163, 58, 100};
  Ipp32s pBandOffset0[6] = {0,1,3,8,13,20};
  Ipp32s pBandOffset1[6];
  int groupLen = 2;
  int bandsNumber = 5;
  int i;
  IppStatus st;

  for (i = 0; i < 40; i++)
    pSrc[i] = (Ipp16s)(100 * i);

  /* The are two ways of using ippsPow43Scale_16s32s_Sf when groupLen */
  /* is not equal to 1                                                */
  /* First way is to use function as is                               */
  st = ippsPow43Scale_16s32s_Sf(pSrc, pDst0, pScaleFactor, pBandOffset0,
                                SF_OFFSET, bandsNumber, groupLen, 0);
  if (st != ippStsOk) return st;

  /* The second way is to recalculate pBandOffset */
  pBandOffset1[0] = pBandOffset0[0];
  for (i = 0; i < bandsNumber; i++)
    pBandOffset1[i+1] = pBandOffset1[i] +
         (pBandOffset0[i+1] - pBandOffset0[i]) * groupLen;

  /* and use function ippsPow43Scale_16s32s_Sf with groupLen is equal to
1 */
  /* The result will be the same                                      */

  st = ippsPow43Scale_16s32s_Sf(pSrc, pDst1, pScaleFactor, pBandOffset1,
                                SF_OFFSET, bandsNumber, 1, 0);

  printf("\n pDst0 = ");
  for (i = 0; i < 40; i++)
    printf("%i ", pDst0[i]);
  printf("\n");

  printf("\n pDst1 = ");
  for (i = 0; i < 40; i++)
    printf("%i ", pDst1[i]);
```

Example 10-4 `ippsPow43Scale` Usage (continued)

```
  printf("\n");
  return st;
}

//Output:
//   pDst1 = 0 1 149709 257062 377244 507968 278884317
//           342520430 409269211 478864278 551089873
//           625767026 702744444 781892234 863097398
//           946260634 12 14 15 16 17 18 19 20 22 23
//           35751 37596 39464 41355 43267 45201 47155
//           49130 51125 53140 55174 57227 59298 61388
//
//   pDst1 = 0 1 149709 257062 377244 507968 278884317
//           342520430 409269211 478864278 551089873
//           625767026 702744444 781892234 863097398
//           946260634 12 14 15 16 17 18 19 20 22 23
//           35751 37596 39464 41355 43267 45201 47155
//           49130 51125 53140 55174 57227 59298 61388
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any pointer passed to the function is `NULL`.
`ippStsSizeErr`	Indicates an error when *`groupLen`* or *`bandsNumber`* is less than 0.

Scale Factors Calculation Functions

In MPEG-2, 4 GA AAC decoder scale factors extracted from the bitstream require an additional restoring procedure. The function `CalcSF` restores actual scale factors from values transmitted in the bit stream.

CalcSF

Restores actual scale factors from the bit stream values.

Syntax

```
IppStatus ippsCalcSF_16s32f(const Ipp16s* pSrc, int offset, Ipp32s* pDst,
    int len);
```

Parameters

pSrc	Pointer to the input data array.
offset	Scale factors offset.
pDst	Pointer to the output data array.
len	Number of elements in the vector.

Description

This function is declared in the `ippac.h` header file. The function `ippsCalcSF` restores actual scale factors from the values *pSrc* transmitted in the bit stream, using the common scale factor offset. Computation is performed according to the following formula

$$pDst[i] = 2^{\frac{1}{4}(pSrc[i]-offset)}, \quad 0 \le i < len.$$

Restored scale factors are written into *pDst*.

See Example 10-5 of using `ippsCalcSF_16s32f` function.

Example 10-5 `ippsCalcSF` Usage

```
#undef SF_OFFSET
#define SF_OFFSET 100
IppStatus calcSF(void)
{
  Ipp16s pSrc[5] = {65, 128, 163, 58, 100};
```

Example 10-5 `ippsCalcSF` Usage (continued)

```
  Ipp32f pDst[5];
  IppStatus st = ippsCalcSF_16s32f(pSrc, SF_OFFSET, pDst, 5);
  int i;

  printf("\n pDst = ");
  for (i = 0; i < 5; i++)
    printf("%f ", pDst[i]);
  printf("\n");
}

//Output:
//   pDst = 0.002323 128.000000 55108.988281 0.000691 1.000000
```

The function `ippsCalcSF` is used in the float-point versions of AAC and MP3 decoders included into IPP Samples downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm .

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

NOTE. The input values in the *pSrc* array should be in the range $-512 \le (pSrc[i] - offset) < 512$. Otherwise the function may operate incorrectly. No error message is returned in this case.

Mantissa Conversion and Scaling Functions

The scale application procedure is necessary for MPEG-2, 4 GA AAC decoder to restore the original spectral values from a set of the inversely quantized spectral coefficients. The whole spectrum is divided into a set of scale factor bands. Since the widths differ from band to band, the band positions are defined by the offset vector.

The mantissa conversion procedure is necessary for the AC3 decoder to restore the original spectral values from a set of the exponents and mantissas in the bitstream.

Scale

Applies scale factors to spectral bands in accordance with spectral bands boundaries.

Syntax

```
IppStatus ippsScale_32f_I(Ipp32f* pSrcDst, const Ipp32f* pSF,
    const int* pBandOffset, int bandsNumber);
```

Parameters

pSrcDst	Pointer to the input and output data array. The size of array must be not less than *pBandOffset*[*bands_number*].
pSF	Pointer to the data array containing scale factors. The size of the array must be not less than *bands_number*.
pBandOffset	Pointer to the vector of band offsets. The size of array must be not less than *bands_number* + 1.
bandsNumber	Number of bands to which scale factors are applied.

Description

This function is declared in the `ippac.h` header file. The function `ippsScale_32f_I` computes scaled values from the input vector, the vector of scale factors, and the vector of band offsets. Operations are performed in-place.

This function applies the set of scale factors *pSF* with *bandsNumber* elements to the bands constructed from the input vector *pSrc*. Band boundaries are defined by the vector of the band offsets *pBandOffset*. All values in each band are multiplied by the corresponding scale factor.

NOTE. The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of *pSrcDst* vector is contained in the *pBandsOffset*[*bandsNumber*] element.

See Example 10-6 of using ippsScale_32f_I.

Example 10-6 ippsScale Usage

```
IppStatus scale(void)
{
  Ipp32f pSrcDst[20];
  Ipp32f pSF[5] = {0.002323, 128, 55108.988281, 0.000691, 1};
  int pBandOffset[6] = {0,1,3,8,13,20};
  IppStatus st;
  int i;

  for (i = 0; i < 20; i++)
    pSrcDst[i] = (Ipp32f)i;

  st = ippsScale_32f_I(pSrcDst, pSF, pBandOffset, 5);

  printf("\n pSrcDst = ");
  for (i = 0; i < 20; i++)
    printf("%f ", pSrcDst[i]);
  printf("\n");
}

//Output:
//   pDst = 0.000000 128.000000 256.000000 165326.968750 220435.953125
//          275544.937500 330653.937500 385762.906250 0.005528 0.006219
//          0.006910 0.007601 0.008292 13.000000 14.000000
//          15.000000 16.000000 17.000000 18.000000 19.000000
```

The function ippsScale is used in the float-point versions of AAC and MP3 decoders included into IPP Samples downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm .

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrcDst* or *pBandsOffset* pointer is NULL.
ippStsSizeErr	Indicates an error when *bandsNumber* is less than or equal to 0.

MakeFloat

Converts mantissa and exponent arrays to float arrays.

Syntax

```
IppStatus ippsMakeFloat_16s32f (Ipp16s* inmant, Ipp16s* inexp, Ipp32s
    size, Ipp32f* outfloat);
```

Parameters

inmant	Array of mantissas.
inexp	Array of exponents.
size	Number of array elements.
outfloat	Array of resulting float arrays.

Description

This function is declared in the `ippac.h` header file. This function converts the mantissa and exponent arrays decoded from the bitstream to the float array of spectral samples by the formula:

$$outfloat[i] = inmant[i] \times 2^{-inexp[i]-15},$$

where $i = 0...size$.

The conversion serves to improve application performance when decoding bitstreams in the AC3 format.

See Example 10-7 of using `ippsMakeFloat_16s32f`.

Example 10-7 `ippsMakeFloat` Usage

```
IppStatus makefloat(void)
{
  Ipp16s inmant[5] = {1234, 56, 25907, 3498, 27854};
  Ipp16s inexp[5] = {-2, -1, 0, 1, 2};
  Ipp32f outfloat[5];
  IppStatus st = ippsMakeFloat_16s32f(inmant, inexp, 5, outfloat);
```

Example 10-7 `ippsMakeFloat` Usage (continued)

```
  int i;

  printf("\n outfloat = ");
  for (i = 0; i < 5; i++)
    printf("%f ", outfloat[i]);
  printf("\n");
}

//Output:
//   outfloat = 0.150635 0.003418 0.790619 0.053375 0.212509
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *inmant*, *inexp* or *outfloat* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *size* is less than or equal to 0.

Modified Discrete Cosine Transform Functions

This section describes Intel IPP functions that compute the modified discrete cosine transform (MDCT) of a signal. MDCT is a lapped orthogonal transform widely used in different audio codecs, such as MPEG-1, MPEG-2, AC-3, and AAC.

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm .

MDCTFwdInitAlloc, MDCTInvInitAlloc

Create and initialize modified discrete cosine transform specification structure.

Syntax

```
IppStatus ippsMDCTFwdInitAlloc_32f(IppsMDCTFwdSpec_32f** ppMDCTSpec,
    int len);
```

```
IppStatus ippsMDCTFwdInitAlloc_16s(IppsMDCTFwdSpec_32s** ppMDCTSpec,
    int len);
IppStatus ippsMDCTInvInitAlloc_32f(IppsMDCTInvSpec_32f** ppMDCTSpec,
    int len);
```

Parameters

ppMDCTSpec	Pointer to pointer to MDCT specification structure to be created.
len	Number of samples in MDCT. Since this set of functions was designed specially for audio coding, only the following values of length are supported: 12, 36, and 2^k, where $k \geq 5$. These values are the only values that appear in audio coding.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdInitAlloc` and `ippsMDCTInvInitAlloc` create and initialize MDCT specification structure *pMDCTSpec* with the specified transform length *len*.

The function `ippsMDCTFwdInitAlloc` initializes the forward MDCT specification structure. The function `ippsMDCTInvInitAlloc` initializes the inverse MDCT specification structure.

`ippsMDCTFwdInitAlloc_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders,
`ippsMDCTFwdInitAlloc_16s` is used in the fixed-point version of AAC encoder,
`ippsMDCTInvInitAlloc_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *ppMDCTSpec* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* does not belong to the above set of admissible values.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

NOTE. `ippsMDCTFwd_16s_Sfs` does not support the length values of 12 and 36.

MDCTFwdInit, MDCTInvInit

Initialize modified discrete cosine transform specification structure.

Syntax

```
IppStatus ippsMDCTFwdInit_32f(IppsMDCTFwdSpec_32f** ppMDCTSpec, int len,
    Ipp8u* pMemSpec, Ipp8u* pMemInit);
IppStatus ippsMDCTFwdInit_16s(IppsMDCTFwdSpec_32f** ppMDCTSpec, int len,
    Ipp8u* pMemSpec, Ipp8u* pMemInit);
IppStatus ippsMDCTInvInit_32f(IppsMDCTInvSpec_32f** ppMDCTSpec, int len,
    Ipp8u* pMemSpec, Ipp8u* pMemInit);
```

Parameters

ppMDCTSpec	Pointer to pointer to MDCT specification structure to be created.
len	Number of samples in MDCT. Since this set of functions was designed specially for audio coding, only the following values of length are supported: 12, 36, and 2^k, where $k \geq 5$. These values are the only values that appear in audio coding.
pMemSpec	Pointer to the area for MDCT specification structure.
pMemInit	Pointer to the buffer that is used for initialization.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdInit` and `ippsMDCTInvInit` initialize MDCT specification structure *pMDCTSpec* with the specified transform length *len*.

The function `ippsMDCTFwdInit` initializes the forward MDCT specification structure.
The function `ippsMDCTInvInit` initializes the inverse MDCT specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *ppMDCTSpec*, *pMemSpec* or *pMemInit* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* does not belong to the above set of admissible values.

NOTE. ippsMDCTFwd_16s_Sfs does not support the length values of 12 and 36.

MDCTFwdFree, MDCTInvFree

Closes modified discrete cosine transform specification structure.

Syntax

```
IppStatus ippsMDCTFwdFree_32f(IppsMDCTFwdSpec_32f* pMDCTSpec);
IppStatus ippsMDCTFwdFree_16s(IppsMDCTFwdSpec_16s* pMDCTSpec);
IppStatus ippsMDCTInvFree_32f(IppsMDCTInvSpec_32f* pMDCTSpec);
```

Parameters

pMDCTSpec	Pointer to the MDCT specification structure to be closed.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdFree` and `ippsMDCTInvFree` close the MDCT structure *pMDCTSpec* by freeing all memory associated with the specification created by `ippsMDCTFwdInitAlloc` or `ippsMDCTInvInitAlloc` functions.

Call either `ippsMDCTFwdFree` or `ippsMDCTInvFree` after the transform is completed.

The function `ippsMDCTFwdFree` closes the forward MDCT specification structure.

The function `ippsMDCTInvFree` closes the inverse MDCT specification structure.

`ippsMDCTFwdFree_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders,
`ippsMDCTFwdFree_16s` is used in the fixed-point version of AAC encoder,
`ippsMDCTInvFree_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into IPP Samples. See introduction to this section.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMDCTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pMDCTSpec* is incorrect.

MDCTFwdGetSize, MDCTInvGetSize

Get the sizes of MDCT specification structure, MDCT initialization, and MDCT work buffer.

Syntax

```
IppStatus ippsMDCTFwdGetSize_32f(int len, int* pSizeSpec, int*
    pSizeInit, int* pSizeBuf);
IppStatus ippsMDCTFwdGetSize_16s(int len, int* pSizeSpec, int*
    pSizeInit, int* pSizeBuf);
IppStatus ippsMDCTInvGetSize_32f(int len, int* pSizeSpec, int*
    pSizeInit, int* pSizeBuf);
```

Parameters

len	Number of samples in MDCT. Since this set of functions is designed specially for audio coding, only the following values of length are supported: 12, 36, and 2^k, where $k \geq 5$. These values are the only values that appear in audio coding.
pSizeSpec	Address of the MDCT specification structure size value in bytes.
pSizeInit	Address of the MDCT initialization buffer size value in bytes.
pSizeBuf	Address of the MDCT work buffer size value in bytes.

Description

These functions are declared in the ippac.h header file. The functions ippsMDCTFwdGetSize and ippsMDCTInvGetSize get the specification structure, initialization, and work buffer sizes of the MDCT described by the length and store the result in *pSizeSpec*, *pSizeInit*, and *pSizeBuf*.

The function `ippsMDCTFwdGetSize` gets the size of the work buffer for the forward MDCT. The function `ippsMDCTInvGetSize` gets the size of the work buffer for the inverse MDCT.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSizeSpec*, *pSizeInit*, or *pSizeBuf* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* does not belong to the above set of admissible values.

NOTE. `ippsMDCTFwd_16s_Sfs` does not support the length values of 12 and 36.

MDCTFwdGetBufSize, MDCTInvGetBufSize

Gets the size of MDCT work buffer.

Syntax

```
IppStatus ippsMDCTFwdGetBufSize_32f(const IppsMDCTFwdSpec_32f*
    pMDCTSpec, int* pSize);
IppStatus ippsMDCTFwdGetBufSize_16s(const IppsMDCTFwdSpec_16s*
    pMDCTSpec, int* pSize);
IppStatus ippsMDCTInvGetBufSize_32f(const IppsMDCTInvSpec_32f*
    pMDCTSpec, int* pSize);
```

Parameters

pMDCTSpec	Pointer to the MDCT specification structure.
pSize	Address of the MDCT work buffer size value in bytes.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdGetBufSize` and `ippsMDCTInvGetBufSize` get the work buffer size of the MDCT described by the specification structure *pMDCTSpec* and store the result in *pSize*.

The function `ippsMDCTFwdGetBufSize` gets the size of the work buffer for the forward MDCT.

The function `ippsMDCTInvGetBufSize` gets the size of the work buffer for the inverse MDCT.

`ippsMDCTGetBufSize_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders,
`ippsMDCTFwdGetBufSize_16s` is used in the fixed-point version of AAC encoder,
`ippsMDCTInvGetBufSize_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMDCTSpec* pointer or *pSize* value is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification structure *pMDCTSpec* is invalid.

MDCTFwd, MDCTInv

Computes forward or inverse modified discrete cosine transform (MDCT) of a signal.

Syntax

```
IppStatus ippsMDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsMDCTFwdSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsMDCTInvSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsMDCTFwdSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f_I(Ipp32f* pSrcDst, const IppsMDCTInvSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsMDCTFwdSpec_32f* pMDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Parameters

pSrc	Pointer to the input data array.
pDst	Pointer to the output data array.
pSrcDst	Pointer to the input and output data array for the in-place operations.
pMDCTSpec	Pointer to the MDCT specification structure.
pBuffer	Pointer to the MDCT work buffer.
scaleFactor	Scalefactor value.

Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwd` and `ippsMDCTInv` compute the forward and inverse modified discrete cosine transform (MDCT), respectively.

In the following definition of MDCT, N denotes the length and $n_0 = (N/2+1)/2$.

For the forward MDCT, $x(n)$ is *pSrc*[*n*] and $y(k)$ is *pDst*[*k*], whereas for the inverse MDCT $x(n)$ is *pDst*[*n*] and $y(k)$ is *pSrc*[*k*].

The forward MDCT is defined by the formula:

$$y(k) = 2 \cdot \sum_{n=0}^{N-1} x(n)\cos\left(\frac{\pi}{N}(n+n_0)(2k+1)\right), \text{ for } 0 \le k < \frac{N}{2}.$$

The inverse MDCT is defined as

$$= \frac{2}{N}?1_\text{Space}??\text{mul}? \sum_{k=0}^{\frac{N}{2}-1} y(k)\cos\left(\frac{\pi}{N}(n+n_0)(2k, \text{ for } 0 \le n < N.\right.$$

The *pBuffer* argument provides the MDCT functions with the necessary work memory and helps to avoid memory allocation within the functions. The buffer may also increase the performance if the MDCT functions use the result of the previous operation stored in cache as an input array.

See Example 10-8 of using `ippsMDCTFwd_32f`.

Example 10-8 `ippsMDCTFwd_32f` Usage

```
IppStatus mdct(void)
{
```

Example 10-8 `ippsMDCTFwd_32f` Usage (continued)

```
  Ipp32f pSrc[32];
  Ipp32f pDst[16];
  Ipp8u* pBuffer;
  IppsMDCTFwdSpec_32f* pMDCTSpec;
  IppStatus st;
  int size, i;

  for (i = 0; i < 32; i++)
    pSrc[i] = (float)cos(IPP_2PI * i / 32);

  st = ippsMDCTFwdInitAlloc_32f(&pMDCTSpec, 32);
  if (st != ippStsOk) return st;

  st = ippsMDCTFwdGetBufSize_32f(pMDCTSpec, &size);
  if (st != ippStsOk) return st;

  if (size != 0) pBuffer = ippsMalloc_8u(size);

  st = ippsMDCTFwd_32f(pSrc, pDst, pMDCTSpec, pBuffer);

  printf("\n pDst = ");
  for (i = 0; i < 16; i++)
    printf("%f ", pDst[i]);
  printf("\n");

  return st;
}

//Output:
//   pDst = 11.430438 -16.110317 7.151322 4.418863
//            -3.528530 -2.781422 2.448268 2.108223
//            -1.940894 -1.757418 1.664175 1.560623
//            -1.509103 -1.455249 1.432051 1.415278
```

`ippsMDCTFwd_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders,
`ippsMDCTInv_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders,
`ippsMDCTFwd_16s_Sfs` is used in the fixed version of AAC encoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification structure *`pMDCTSpec`* is invalid.

`ippStsMisalignedBuf`	Indicates misaligned data. Supply aligned data for better performance.

Block Filtering Functions

Intel IPP functions described in this section implement the finite impulse response (FIR) block filter. You can use this group of functions to design transform domain adaptive filters. These filters preprocess the signal by decomposing the input vector into orthogonal components, which are subsequently used as inputs to a parallel bank of adaptive subfilters. You may use this approach for implementing frequency domain linear predictors in audio codecs, for example, CELP and AAC.

The filtering function receives a number of vectors (signals). Every call of a filtering function produces one filtered sample for each input signal. The library functions do not perform any particular adaptation method but you can specify the filter taps at each call of a filtering function.

To use the FIR block filter functions, follow these general steps:

1. Call `FIRBlockInitAlloc` to initialize the state structure of a block filter.
2. Call `FIRBlockOne` to filter a vector of samples through a block filter.
3. Call `FIRBlockFree` to free dynamic memory associated with the FIR block filter.

FIRBlockInitAlloc

Initializes FIR block filter state.

Syntax

```
IppStatus ippsFIRBlockInitAlloc_32f(IppsFIRBlockState_32f** pState,
    int order, int len);
```

Parameters

pState	Pointer to the FIR block filter state structure to be created.
order	Number of elements in the array containing the tap values.
len	Number of input signals.

Description

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockInitAlloc` creates and initializes a FIR block filter state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.
`ippStsFIRLenErr`	Indicates an error when *`order`* or *`len`* is less than or equal to 0.

FIRBlockFree

Closes FIR block filter state.

Syntax

```
IppStatus ippsFIRBlockFree_32f(IppsFIRBlockState_32f* pState);
```

Parameters

`pState`	Pointer to the FIR block filter state structure to be closed.

Description

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockFree` closes the FIR block filter state by freeing all memory associated with the filter state created by the function `ippsFIRBlockInitAlloc`.

Call `ippsFIRBlockFree` after filtering is completed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pState`* is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

FIRBlockOne

Filters vector of samples through FIR block filter.

Syntax

```
IppStatus ippsFIRBlockOne_32f(Ipp32f* pSrc, Ipp32f* pDst,
    IppsFIRBlockState_32f* pState, Ipp32f* pTaps);
```

Parameters

pSrc	Pointer to the input vector of samples to be filtered.
pDst	Pointer to the vector of filtered output samples.
pState	Pointer to the FIR filter state structure.
pTaps	Pointer to the vector of filter taps.

Description

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockOne` filters a vector of samples *pSrc* of the length *len* through a filter and stores the result in *pDst*.

The filter taps are specified in the vector *pTaps* of the length *order*. The values of *len* and *order* parameters are specified in the `FIRBlockInitAlloc` call.

In the following definition of the FIR filter, the sample of the input vector i to be filtered with the delay k is denoted x^i_{n-k}, and the taps are denoted h_k. The output value y_i is defined by the following formula:

$$y^i_n = \sum_{k=0}^{order-1} h_k x^i_{n-k}, \quad 0 \le i < len.$$

Before calling the function `ippsFIRBlockOne`, initialize the filter state by calling the function `ippsFIRBlockInitAlloc`. Specify the taps values in the argument *pTaps*.

Example 10-9 Single-Rate Filtering with the `ippsFIRBlockOne` Function

```
IppStatus fir(void)
{
#undef NUMITERS
#define NUMITERS 20
#undef BLOCKSIZE
```

Example 10-9 Single-Rate Filtering with the `ippsFIRBlockOne` Function (continued)

```
#define BLOCKSIZE 20
      int n;
      int i;
      IppStatus status;
      IppsFIRBlockState_32f *fctx;
      Ipp32f x [BLOCKSIZE], y [BLOCKSIZE];
            const float taps[] = {
            0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f,
0.1869f,
               0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0
      };
      ippsFIRBlockInitAlloc_32f( &fctx, 11, BLOCKSIZE );
      for (n =0;n<NUMITERS;++n)
      {
            for (i = 0; i< BLOCKSIZE; i++) x[i]=(Ipp32f)sin(IPP_2PI *
               n *0.2 + i);
            status = ippsFIRBlockOne_32f( x, y, fctx, (Ipp32f*)taps );
            for (i = 0; i< BLOCKSIZE; i++)
               printf("%f", y[i]);
      }
      ippsFIRBlockFree_32f(fctx);
      return status;
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.
`ippStsMisalignedBuf`	Indicates misaligned arrays. Supply aligned data for better performance.
`ippStsFIRLenErr`	Indicates an error when one of the following conditions is true: *pState->len* is less than or equal to 0 *pState->order* is less than or equal to 0 *pState->queue_end* is less than 0 *pState->queue_end* is greater or equal to *pState->order*.

Frequency Domain Prediction Functions

MPEG-2, 4 AAC encoder uses prediction in frequency domain (FDP) to decrease redundancy in the audio signal and ensure more effective coding. For each spectral line, input signals are filtered through the second order adaptive FIR filter called a predictor. Then, instead of processing the signal, the difference between the original signal and the filter output (that is, the prediction error), is passed for further processing. The decoder derives the original signal from the prediction error using a symmetrical block.

You should regularly reset the predictors to their initial state to reduce accumulated calculation error. You may also need to reset the predictors in special cases discussed and described in the ISO-144963 standard. You can reset predictors for the entire spectrum, several scale factor bands, or a selected group of spectral lines.

For more information on the algorithm of filter coefficient adaptation and FDP usage see ISO-144963, clause 6.5.3.2.

To use the FDP prediction tool functions described in this section, follow these steps:

1. Call the function `FDPInitAlloc` to allocate memory and initialize predictor state.
2. Call the function `FDPFwd` for each frame to calculate prediction error or call the function `FDPInv` to retrieve the original signal.
3. Call the functions `FDPReset`, `FDPResetSfb`, or `FDPResetGroup` to reset predictors in certain spectral lines at any time after creating the state.
4. Call the function `FDPFree` to free the memory allocated by `ippsFDPInitAlloc`.

FDPInitAlloc

Creates and initializes predictor state.

Syntax

```
IppStatus ippsFDPInitAlloc_32f(IppsFDPState_32f** ppFDPState, int len);
```

Parameters

ppFDPState	Pointer to pointer to the FDP state structure to be created.
len	Number of spectral lines to be processed.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPInitAlloc_32f` creates and initializes the FDP state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`ppFDPState`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

FDPInit

Initializes predictor state.

Syntax

```
IppStatus ippsFDPInit_32f(IppsFDPState_32f** ppFDPState, int len, Ipp8u*
    pMemSpec);
```

Parameters

`ppFDPState`	Pointer to pointer to the FDP state structure to be initialized.
`len`	Number of spectral lines to be processed.
`pMemSpec`	Pointer to the area for FDP state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPInit_32f` initializes the FDP state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`ppFDPState`* or *`pMemSpec`* is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

FDPFree

Closes FDP state.

Syntax

```
IppStatus ippsFDPFree_32f(IppsFDPState_32f* pFDPState);
```

Parameters

pFDPState	Pointer to the FDP state structure to be closed.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPFree` closes the FDP state by freeing all memory associated with the FDP state structure created by the function `FIRBlockInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pFDPState* pointer is NULL.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

FDPGetSize

Gets size of FDP state structure.

Syntax

```
IppStatus ippsFDPGetSize_32f(int len, int* pSizeState);
```

Parameters

len	Number of spectral lines to be processed.
pSizeState	Address of size value in bytes of the frequency domain predictor state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPGetSize_32f` gets sizes of the frequency domain predictor state structure and stores the results in *`pSizeState`*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSizeState`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

FDPReset

Resets predictors for all spectral lines.

Syntax

```
IppStatus ippsFDPReset_32f(IppsFDPState_32f* pFDPState);
```

Parameters

`pFDPState`	Pointer to the predictor specific state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPReset` resets predictors for all spectral lines.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pFDPState`* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

FDPResetSfb

Resets predictor-specific information in some scale factor bands.

Syntax

```
IppStatus ippsFDPResetSfb_32f (const int* pSfbOffset, int sfbNumber, const Ipp8u*
    pResetFlag, IppsFDPState_32f* pFDPState);
```

Parameters

pSfbOffset	Pointer to the band offset vector.
sfbNumber	Number of bands.
pResetFlag	Array of flags showing whether predictors for spectral lines in a certain scale factor band need to be reset.
pFDPState	Pointer to the predictor specific state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPResetSfb_32f` resets predictors for all spectral lines in each scale factor band *i*, for which *pResetFlag*[*i*] is not equal to 0.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *sfbNumber* is less than or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

FDPResetGroup

Resets predictors for group of spectral lines.

Syntax

```
IppStatus ippsFDPResetGroup_32f (int resetGroupNumber, int step, IppsFDPState_32f*
    pFDPState);
```

Parameters

resetGroupNumber	Number of the group to be reset.
step	Distance between two neighboring spectral lines in the group.
pFDPState	Pointer to the predictor specific state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPResetGroup_32f` resets predictors for each *step*-th spectral line beginning from the start up to the end of spectrum.

See Example 10-10 of using `ippsFDPInv` and `ippsFDPResetGroup_32f` functions.

Example 10-10 `ippsFDPInv` and `ippsFDPResetGroup` Usage

```
/* 1 step - initialization */
IppsFDPState_32f* pFDPState;
ippsFDPInitAlloc_32f(&pFDPState, 1024);

/* 2 step - using inside decoder. Something like this (from AAC)*/

ippsFDPInv_32f_I(p_spectrum, sfb_offset_long_window,
                 pred_max_sfb, prediction_used,
                 pFDPState);

if ((predictor_reset) &&
    predictor_reset_group_number > 0 &&
    predictor_reset_group_number < 31) {

    ippsFDPResetGroup_32f(30, predictor_reset_group_number - 1,
pFDPState);
 }

 /* 3 step - freeing */
 ippsFDPFree_32f(pFDPState);
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFDPState* pointer is NULL.
ippStsSizeErr	Indicates an error when *resetGroupNumber* or *step* is less than or equal to 0.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.

FDPFwd

Performs frequency domain prediction procedure and calculates prediction error.

Syntax

```
IppStatus ippsFDPFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, IppsFDPState_32f*
    pFDPState);
```

Parameters

pSrc	Pointer to the input data array.
pDst	Pointer to the data array to be filled with prediction errors.
pFDPState	Pointer to the predictor specific state structure.

Description

This function is declared in the ippac.h header file. The function ippsFDPFwd applies frequency domain prediction procedure to the input signal *pSrc* and stores prediction errors in the *pDst* vector.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFDPState* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.
ippStsMisalignedBuf	Indicates misaligned arrays. Supply aligned data for better performance.

FDPInv

Retrieves input signal from prediction error, using frequency domain prediction procedure.

Syntax

```
IppStatus ippsFDPInv_32f_I(Ipp32f* pSrcDst, const int* pBandsOffset, int
    predictorBandsNumber, Ipp8u* pPredictionUsed, IppsFDPState_32f* pFDPState);
```

Parameters

pSrcDst	Pointer to the input and output data array for the in-place operation.
pBandsOffset	Pointer to the band offset vector.
predictorBandsNumber	Number of scale factor bands.
pPredictionUsed	Pointer to array of flags showing whether prediction will be used in certain scale factor band.
pFDPState	Pointer to the predictor specific state structure.

Description

This function is declared in the `ippac.h` header file. The function `ippsFDPInv` applies the procedure of frequency domain prediction to specific bands of the input spectral vector *pSrcDst*. Positions of bands are defined by the parameters *predictorBandsNumber* and *pBandsOffset*.

For each scale factor band *i*, if *pPredictionUsed*[*i*] is not equal to 0, all values of *pSrcDst* within the band are treated as a prediction error and the original signal is restored. If *pPredictionUsed*[*i*] is equal to 0, all values of *pSrcDst* within the band are treated as a signal and will be passed without any changes.

Regardless of the *pPredictionUsed* flag, the coefficients of each predictor are updated.

NOTE. The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of *pSrcDst* vector is stored in the *pBandsOffset*[*predictorBandsNumber*] element.

See Example 10-10 of using `ippsFDPInv` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *predictorBandsNumber* is less than or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.
`ippStsMisalignedBuf`	Indicates misaligned arrays. Supply aligned data for better performance.

VLC Functions

Variable Length Coding (VLC) is an audio data compression method that uses statistical modeling to define which values occur more frequently than others to build tables for subsequent encoding and decoding operations. Audio data in the bitstream is encoded with Variable Length Code tables so that the shortest codes correspond to the most frequent values and the longer codes correspond to the less frequent values. Every standard that uses Variable Length Coding has corresponding tables named `inputTable` that list possible codes and their values.

Example 10-11inputTable Format

```
static IppsVLCTable_32s inputTable[]=
{
   {value0, code0, length0};
   {value1, code1, length1};
    …
   {valueN, codeN, lengthN};
}
```

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm .

VLCDecodeEscBlock_MP3

Parses the bitstream and decodes variable length code for MP3 using signed VLC tables.

Syntax

```
IppStatus ippsVLCDecodeEscBlock_MP3_1u16s(const Ipp8u** ppBitStream, int*
    pBitOffset, int linbits, Ipp16s* pData, int len, const
    IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

ppBitStream	Pointer to pointer to the current byte in the bitstream buffer. **ppBitStream* is updated by the function.
pBitOffset	Pointer to the bit position in the byte pointed by **ppBitStream*. **pBitOffset* is updated by the function.
linbits	Length of escape sequence.

pData	Pointer to the array of decoded values.
len	Number of values to decode into array *pData*.
pVLCSpec	Pointer to VLCDecoder specification structure.

Description

The function ippsVLCDecodeEscBlock_MP3_1u16s is declared in the ippac.h header file. The function parses the bitstream and decodes variable length code using the VLCDecoder specification structure built by the function ippsVLCDecodeInit_32s or ippsVLCDecodeInitAlloc_32s from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function ippsVLCDecodeEscBlock_MP3_1u16s, rearrange *inputTable* before initialization as follows:

```
value = (first_sample_value) << 8) + (second_sample_value + 128).
code  = (((code_ini << (first_sample_value != 0)) + (first_sample_value <
0)) << (second_sample_value != 0)) + (first_sample_value < 0).
length = length_ini + (first_sample_value != 0) + (second_sample_value !=
0).
```

MP3Esc VLC codebook is an unsigned two-dimensional codebook with a large absolute value equal to 15.

From one record of the initial *inputTable* you can obtain from one to four records of rearranged *inputTable*.

Example

```
abs(first_sample_value)  = 9;
abs(second_sample_value) = 6;

code_ini = 0xf5
length_ini = 12;

record 1: first_sample_value = -9, second_sample_value = -6;
         value = (-9 << 8) + (-6 + 128) = -2182;
         code = (((0xf5 << 1) + 1) << 1) + 1 = 0x3d7;
         length = 12 + 1 + 1 = 14;
         {-2182, 0x3d7, 14)

record 2: first_sample_value = -9, second_sample_value = 6;
         value = (-9 << 8) + (6 + 128) = -2170;
         code = (((0xf5 << 1) + 1) << 1) + 0 = 0x3d6;
```

```
        length = 12 + 1 + 1 = 14;
        -2170, 0x3d6, 14)

record 3: first_sample_value = 9, second_sample_value = -6;
        value = (9 << 8) + (-6 + 128) = 2426;
        code = (((0xf5 << 1) + 0) << 1) + 1 = 0x3d5;
        length = 12 + 1 + 1 = 14;
        {2426, 0x3d7, 14)

record 4: first_sample_value = 9, second_sample_value = 6;
        value = (9 << 8) + (6 + 128) = 2438;
        code = (((0xf5 << 1) + 0) << 1) + 0 = 0x3d4;
        length = 12 + 1 + 1 = 14;
        {2438, 0x3d7, 14)
```

The function `ippsVLCDecodeEscBlock_MP3_1u16s` is used in the float-point and fixed-point versions of MP3 decoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers passed to the function is `NULL`.
`ippStsBitOffsetErr`	Indicates an error when *`*pBitOffset`* is less than 0 or more than 7.
`ippStsContextMatchErr`	Indicates an error when `pVLCSpec` structure was not created by `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` functions.
`ippStsVLCInputDataErr`	Indicates an error when incorrect input is used. For decode functions it can indicate that bitstream contains code that is not specified inside the used table.

VLCDecodeEscBlock_AAC

Parses the bitstream and decodes variable length code for AAC using signed VLC tables.

Syntax

```
IppStatus ippsVLCDecodeEscBlock_AAC_1u16s(const Ipp8u** ppBitStream, int*
    pBitOffset, Ipp16s* pData, int len, const IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

ppBitStream	Pointer to pointer to the current byte in the bitstream buffer. **ppBitStream* is updated by the function.
pBitOffset	Pointer to the bit position in the byte pointed by **ppBitStream*. **pBitOffset* is updated by the function.
pData	Pointer to the array of decoded values.
len	Number of values to decode into array *pData*.
pVLCSpec	Pointer to `VLCDecoder` specification structure.

Description

The function `ippsVLCDecodeEscBlock_AAC_1u16s` is declared in the `ippac.h` header file. The function parses the bitstream and decodes variable length code using the `VLCDecoder` specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function `ippsVLCDecodeEscBlock_AAC_1u16s`, rearrange *inputTable* before initialization as follows:

```
value = (first_sample_value << 8) + (second_sample_value + 128).
code = (((code_ini << (first_sample_value != 0)) + (first_sample_value <
0)) << (second_sample_value != 0)) + (first_sample_value < 0).
length = length_ini + (first_sample_value != 0) + (second_sample_value !=
0).
```

AACEsc VLC codebook is an unsigned two-dimensional codebook with a large absolute value is equal to 16.

From one record of initial *inputTable* you can obtain from one to four records of rearranged *inputTable*.

Example

```
Initial input table record – {159, 0x1ad, 9}
abs(first_sample_value) = 159/17 = 9;
abs(second_sample_value) = 159 – 17*9 = 6;

code_ini = 0x1ad
length_ini = 9;

record 1: first_sample_value = -9, second_sample_value = -6;
        value = (-9 << 8) + (-6 + 128) = -2182;
        code = (((0x1ad << 1) + 1) << 1) + 1 = 0x6b7;
        length = 9 + 1 + 1 = 11;
        {-2182, 0x6b7, 11)
record 2: first_sample_value = -9, second_sample_value = 6;
        value = (-9 << 8) + (6 + 128) = -2170;
        code = (((0x1ad << 1) + 1) << 1) + 0 = 0x6b6;
        length = 9 + 1 + 1 = 11;
        {-2170, 0x6b6, 11)

record 3: first_sample_value = 9, second_sample_value = -6;
        value = (9 << 8) + (-6 + 128) = 2426;
        code = (((0x1ad << 1) + 0) << 1) + 1 = 0x6b5;
        length = 9 + 1 + 1 = 11;
        {2426, 0x6b7, 11)

record 4: first_sample_value = 9, second_sample_value = 6;
        value = (9 << 8) + (6 + 128) = 2438;
        code = (((0x1ad << 1) + 0) << 1) + 0 = 0x6b4;
        length = 9 + 1 + 1 = 11;
        {2438, 0x6b7, 11)
```

The function `ippsVLCDecodeEscBlock_AAC_1u16s` is used in the float-point and fixed-point versions of AAC decoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers passed to the function is `NULL`.
`ippStsBitOffsetErr`	Indicates an error when *`pBitOffset`* is less than 0 or more than 7.

ippStsContextMatchErr	Indicates an error when pVLCSpec structure was not created by ippsVLCDecodeInit_32s or ippsVLCDecodeInitAlloc_32s functions.
ippStsVLCInputDataErr	Indicates an error when incorrect input is used. For decode functions it can indicate that bitstream contains code that is not specified inside the used table.
ippStsVLCAACEscCodeLengthErr	Indicates an error when bitstream contains AAC-Esc code with the length more than 21.

VLCDecodeUTupleEscBlock_MP3

Parses the bitstream and decodes variable length code for MP3 using unsigned multi-tupled VLC tables.

Syntax

```
IppStatus ippsVLCDecodeUTupleEscBlock_MP3_1u16s(Ipp8u **ppBitStream, int
    *pBitOffset, int linbits, Ipp16s *pData, int len, const
    IppsVLCDecodeUTupleSpec_32s *pVLCSpec);
```

Parameters

ppBitStream	Pointer to pointer to the current byte in the bitstream buffer. **ppBitStream* is updated by the function.
pBitOffset	Pointer to the bit position in the byte pointed by **ppBitStream*. Valid within the range 0 to 7. **pBitOffset* is updated by the function.
linbits	Length of escape sequence.
pData	Pointer to the array of decoded values.
len	Number of samples to decode into *pData* array.
pVLCSpec	Pointer to VLCDecoder specification structure.

Description

The function ippsVLCDecodeUTupleEscBlock_MP3_1u16s is declared in the ippac.h header file. The function parses the bitstream and decodes variable length code using the VLCDecoder specification structure built by the function ippsVLCDecodeUTupleInit_32s or ippsVLCDecodeUTupleInitAlloc_32s from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit

offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function `ippsVLCDecodeUTupleEscBlock_MP3_1u16s`, rearrange *`inputTable`* before initialization as a 2-tuple table.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers passed to the function is `NULL`.
`ippStsBitOffsetErr`	Indicates an error when *`pBitOffset` is less than 0 or more than 7.
`ippStsContextMatchErr`	Indicates an error when `pVLCSpec` structure was not created by `ippsVLCDecodeUTupleInit_32s` or `ippsVLCDecodeUTupleInitAlloc_32s` functions.
`ippStsVLCInputDataErr`	Indicates an error when incorrect input is used. For decode functions, it can indicate that the bitstream contains code that is not specified inside the table used.

VLCDecodeUTupleEscBlock_AAC

Parses the bitstream and decodes variable length code for AAC using unsigned multi-tupled VLC tables.

Syntax

```
IppStatus ippsVLCDecodeUTupleEscBlock_AAC_1u16s(Ipp8u **ppBitStream, int
    *pBitOffset, Ipp16s *pData, int len, const IppsVLCDecodeUTupleSpec_32s
    *pVLCSpec);
```

Parameters

`ppBitStream`	Pointer to pointer to the current byte in the bitstream buffer. *`ppBitStream` is updated by the function.
`pBitOffset`	Pointer to the bit position in the byte pointed by *`ppBitStream`. Valid within the range 0 to 7. *`pBitOffset` is updated by the function.
`pData`	Pointer to the array of decoded values.

len	Number of samples to decode into *pData* array.
pVLCSpec	Pointer to VLCDecoder specification structure.

Description

The function ippsVLCDecodeUTupleEscBlock_AAC_1u16s is declared in the ippac.h header file. The function parses the bitstream and decodes variable length code using the VLCDecoder specification structure built by the function ippsVLCDecodeUTupleInit_32s or ippsVLCDecodeUTupleInitAlloc_32s from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function ippsVLCDecodeUTupleEscBlock_AAC_1u16s, rearrange *inputTable* before initialization as a 2-tuple table.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers passed to the function is NULL.
ippStsBitOffsetErr	Indicates an error when **pBitOffset* is less than 0 or more than 7.
ippStsContextMatchErr	Indicates an error when pVLCSpec structure was not created by ippsVLCDecodeUTupleInit_32s or ippsVLCDecodeUTupleInitAlloc_32s functions.
ippStsVLCInputDataErr	Indicates an error when incorrect input is used. For decode functions, it can indicate that the bitstream contains code that is not specified inside the table used.
ippStsVLCAACEscCodeLengthErr	Indicates an error when the bitstream contains AAC-Esc code with the length exceeding 21.

VLCCountEscBits_MP3

Calculates the number of bits necessary for encoding in MP3 format.

Syntax

```
IppStatus ippsVLCCountEscBits_MP3_16s32s(const Ipp16s* pInputData, int len, int
    linbits, Ipp32s* pCountBits, const IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pInputData	Pointer to the array of source values.
len	Size of values array *pInputData*.
linbits	Length of escape sequence.
pCountBits	Pointer to calculated length in bits to encode *pInputData*.
pVLCSpec	Pointer to `VLCEncoder` specification structure.

Description

The function `ippsVLCCountEscBits_MP3_16s32s` is declared in the `ippac.h` header file. The function calculates the number of bits necessary for encoding source data in *pInputData*, using the `VLCEncoder` specification structure built by the function `ippsVLCEncodeInit_32s` or `ippsVLCEncodeInitAlloc_32s` from data compression domain. To use the function `ippsVLCCountEscBits_MP3_16s32s`, rearrange *inputTable* before initialization in the same way as described in the description of `VLCDecodeEscBlock_MP3` with one exception (because the large absolute value is equal to 15):

```
value= (first_sample_value << 5) + (second_sample_value + 15).
```

The function `ippsVLCCountEscBits_MP3_16s32s` is used in the float-point and fixed-point versions of MP3 encoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.

ippStsContextMatchErr	Indicates an error when pVLCSpec structure was not created by ippsVLCEncodeInit_32s or ippsVLCEncodeInitAlloc_32s functions.

VLCCountEscBits_AAC

Calculates the number of bits necessary for encoding in AAC format.

Syntax

```
IppStatus ippsVLCCountEscBits_AAC_16s32s(const Ipp16s* pInputData, int len,
    Ipp32s* pCountBits, const IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pInputData	Pointer to the array of source values.
len	Size of values array *pInputData*.
pCountBits	Pointer to calculated length in bits to encode *pInputData*.
pVLCSpec	Pointer to VLCEncoder specification structure.

Description

The function ippsVLCCountEscBits_AAC_16s32s is declared in the ippac.h header file. The function calculates the number of bits necessary for encoding source data in *pInputData*, using the VLCEncoder specification structure built by the function ippsVLCEncodeInit_32s or ippsVLCEncodeInitAlloc_32s from data compression domain. To use the function ippsVLCCountEscBits_AAC_16s32s, rearrange *inputTable* before initialization in the same way as described in the description of VLCDecodeEscBlock_AAC with one exception (because the large absolute value is equal to 16):

```
value = (first_sample_value << 6) + (second_sample_value + 16).
```

The function ippsVLCCountEscBits_AAC_16s32s is used in the float-point and fixed-point versions of AAC encoder included into IPP Samples. See introduction to this section.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when pVLCSpec structure was not created by ippsVLCEncodeInit_32s or ippsVLCEncodeInitAlloc_32s functions.

VLCEncodeEscBlock_MP3

Encodes an array of values into destination bitstream in MP3 format and advances bitstream pointer.

Syntax

```
IppStatus ippsVLCEncodeEscBlock_MP3_16s1u(const Ipp16s* pInputData, int len,
    int linbits, Ipp8u** ppBitStream, int* pBitOffset, const
    IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pInputData	Pointer to the array of source values.
len	Size of values array *pInputData*.
linbits	Length of escape sequence.
ppBitStream	Pointer to pointer to the current byte in the bitstream buffer. **ppBitStream* is updated by the function.
pBitOffset	Ppointer to the bit position in the byte pointed by **ppBitStream*. **pBitOffset* is updated by the function.
pVLCSpec	Pointer to VLCEncoder specification structure.

Description

The function ippsVLCEncodeEscBlock_MP3_16s1u is declared in the ippac.h header file. The function encodes an array of values from *pInputData* into destination *Bitstream* and advances the bitstream pointer, using the VLCEncoder specification structure built by the function ippsVLCEncodeInit_32s or ippsVLCEncodeInitAlloc_32s from data compression domain. To use the function ippsVLCCountEscBits_MP3_16s32s, rearrange *inputTable* before initialization in the same way as described in the description of VLCDecodeEscBlock_MP3 with one exception (because the large absolute value is equal to 15):

```
value = (first_sample_value << 5) + (second_sample_value + 15).
```

The function `ippsVLCEncodeEscBlock_MP3_16s1u` is used in the float-point and fixed-point versions of MP3 encoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsBitOffsetErr`	Indicates an error when *`pBitOffset` is less than 0 or more than 7.
`ippStsContextMatchErr`	Indicates an error when `pVLCSpec` structure was not created by `ippsVLCEncodeInit_32s` or `ippsVLCEncodeInitAlloc_32s` functions.

VLCEncodeEscBlock_AAC

Encodes an array of values into destination bitstream in AAC format and advances bitstream pointer.

Syntax

```
IppStatus ippsVLCEncodeEscBlock_AAC_16s1u(const Ipp16s* pInputData, int len,
    Ipp8u** ppBitStream, int* pBitOffset, const IppsVLCEncodeSpec_32s*
    pVLCSpec);
```

Parameters

`pInputData`	Pointer to the array of source values.
`len`	Size of values array *`pInputData`*.
`ppBitStream`	Pointer to pointer to the current byte in the bitstream buffer. **`ppBitStream`* is updated by the function.
`pBitOffset`	Pointer to the bit position in the byte pointed by **`ppBitStream`*. **`pBitOffset`* is updated by the function.
`pVLCSpec`	Pointer to `VLCEncoder` specification structure.

Description

The function `ippsVLCEncodeEscBlock_AAC_16s1u` is declared in the `ippac.h` header file. The function encodes an array of values from *pInputData* into destination *Bitstream* and advance bitstream pointer, using the `VLCEncoder` specification structure built by the function `ippsVLCEncodeInit_32s` or `ippsVLCEncodeInitAlloc_32s` from data compression domain. To use the function `ippsVLCCountEscBits_AAC_16s32s`, rearrange *inputTable* before initialization in the same way as described in the description of `VLCDecodeEscBlock_AAC` with one exception (because the large absolute value is equal to 16):

```
value = (first_sample_value << 6) + (second_sample_value + 16).
```

The function `ippsVLCEncodeEscBlock_AAC_16s1u` is used in the float-point and fixed-point versions of AAC encoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsBitOffsetErr`	Indicates an error when **pBitOffset* is less than 0 or more than 7.
`ippStsContextMatchErr`	Indicates an error when `pVLCSpec` structure was not created by `ippsVLCEncodeInit_32s` or `ippsVLCEncodeInitAlloc_32s` functions.

Psychoacoustic Functions

This section describes psychoacoustic functions.

Spread

Computes spreading function.

Syntax

```
IppStatus ippsSpread_16s_Sfs(Ipp16s* Src1, Ipp16s Src2, int
    inScaleFactor, Ipp16s* pDst);
```

Parameters

Src1	Input data 1.
Src2	Input data 2.
inScaleFactor	Input value scalefactor value.
pDst	Pointer to the output data vector, output data is in Q15 format.

Description

This function is declared in the `ippac.h` file. The function gives a masking threshold produced by a single tone masker frequency *Src2* (in Bark) for neighboring frequency *Src1* (in Bark). This function is widely used in psychoacoustics.

See Example 10-12 of using `ippsSpread_16s_Sfs` function.

Example 10-12 `ippsSpread` Usage

```
// Function ippsSpread_16s_Sfs calculates spread(b1,b2) value
// using the following formula:
// tmpx = (b2 >= b1) ? 3*(b2-b1) : 1.5f*(b2-b1);
// tmpz = 8 * ((tmpx-0.5f)*(tmpx-0.5f) - 2*(tmpx-0.5f));
// if (tmpz > 0) tmpz = 0;
// tmpy = 15.811389f + 7.5f*(tmpx + 0.474f)-
// 17.5f*(float)sqrt(1 + (tmpx+0.474f)*(tmpx+0.474f));
//
// spread = (tmpy < -100 ? 0 : (float)pow(10,(float)(tmpz + tmpy)/10));
// The output of ippsSpread_16s_Sfs in Q15 format.

IppStatus spread(void)
{
   Ipp16s b1 = 18668; /* 9.115234 in Q11 */
```

Example 10-12 ippsSpread Usage (continued)

```
    Ipp16s b2 = 20255; /* 9.890137 in Q11 */
    Ipp16s dst;
    IppStatus st = ippsSpread_16s_Sfs(b1, b2, -11, &dst);

    printf ("dst = %i\n", dst);
    return st;
}

//Output:
//    dst = 547
```

The function ippsSpread_16s_Sfs is used in the fixed-point version of AAC encoder included into IPP Samples downloadable from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm .

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDst* pointer is NULL.

Vector Quantization Functions

This sections describes functions used for vector quantization operations.

VQCodeBookInitAlloc

Creates and initializes the codebook structure.

Syntax

```
IppStatus ippsVQCodeBookInitAlloc_32f(const Ipp32f* pInputTable,
    IppsVQCodeBookState_32f** ppCodeBook, int step, int height);
```

Parameters

pInputTable	Pointer to the codebook table of the size *step* * *height* containing *height* quantization vectors of the length *step*.
ppCodeBook	Pointer to pointer to the codebook structure to be created.

step	Step to the next line in the table *InputTable*, quantization vector length.
height	Table height, number of quantization vectors.

Description

This function is declared in the `ippac.h` header file. This function creates and initializes the structure that contains the codebook and additional information needed for the search operation. This structure is used during the vector quantization operation by the VQPreliminarySelect, VQMainSelect, VQIndexSelect, and VQReconstruction functions.

To free the memory allocated by this function, use the function VQCodeBookFree.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *ppCodeBook* or *pInputTable* pointer is NULL.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.

VQCodeBookInit

Initializes the codebook structure.

Syntax

```
IppStatus ippsVQCodeBookInit_32f(const Ipp32f* pInputTable,
    IppsVQCodeBookState_32f** ppCodeBook, int step, int height, Ipp8u*
    pMemSpec);
```

Parameters

pInputTable	Pointer to the codebook table of the size *step* * *height* containing *height* quantization vectors of the length *step*.
ppCodeBook	Pointer to pointer to the codebook structure to be created.
step	Step to the next line in the table *InputTable*, quantization vector length.
height	Table height, number of quantization vectors.

pMemSpec	Pointer to the area for the codebook structure.

Description

This function is declared in the `ippac.h` header file. This function initializes the structure that contains the codebook and additional information needed for the search operation. This structure is used during the vector quantization operation by the `VQPreliminarySelect`, `VQMainSelect`, `VQIndexSelect`, and `VQReconstruction` functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pInputTable*, *ppCodeBook* or *pMemSpec* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *step* or *height* is less than or equal to 0.

VQCodeBookFree

Closes the `IppsVQCodeBookState_32f` *structure created by the function* `ippsVQCodeBookInitAlloc`*.*

Syntax

```
IppStatus ippsVQCodeBookFree_32f(IppsVQCodeBookState_32f* pCodeBook);
```

Parameters

pCodeBook	Pointer to specified `IppsVQCodeBookState_32f` structure.

Description

This function is declared in the `ippac.h` header file. The function closes the `IppsVQCodeBookState_32f` structure created by `VQCodeBookInitAlloc`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCodeBook* pointer is `NULL`.

VQCodeBookGetSize

Gets the size of the codebook structure.

Syntax

```
IppStatus ippsVQCodeBookGetSize_32f(int step, int height, int*
    pSizeState);
```

Parameters

step	Step to the next line in the codebook table, quantization vector length.
height	Table height, number of quantization vectors.
pSizeState	Address of the codebook structure size value in bytes.

Description

This function is declared in the `ippac.h` header file. This function gets the codebook structure sizes and stores the results in *pSizeState*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSizeState* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when the step or height is less than or equal to 0.

VQPreliminarySelect

Selects candidates for the nearest code vector of codebooks.

Syntax

```
IppStatus ippsVQPreliminarySelect_32f(const Ipp32f* pSrc, const Ipp32f*
    pWeights, int nDiv, const Ipp32s* pLengths, Ipp32s* pIndx, Ipps32s* pSign,
    int nCand, int* pPolbits, IppsVQCodeBookState_32f* pCodeBook);
```

Parameters

pSrc	Source vector to be quantized.
pWeights	Pointer to the vector of weights.
nDiv	Number of fragmentations of the *src* and *weights* vectors.
pLengths	Pointer to an array of lengths of fragmentations.
pIndx	Pointer to the output vector of indexes of the *nCand* minimum candidates.
pSign	Pointer to the output vector of signs of *nCand* minimum candidates. • The value of 1 indicates that the minimal distortion appears when the norm is negative. • The value of 0 indicates that the minimal distortion appears when the norm is positive.
nCand	Number of output candidates.
pPolbits	Pointer to the *polbits* flag vector.
pCodeBook	Pointer to the specified `IppsVQCodeBookState_32f` structure.

Description

This function is declared in the `ippac.h` header file. The function computes indexes and finds *nCand* vectors with the closest values from the codebook.

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

if the *polbits*[*idiv*] is greater than `MAXBIT_SHAPE`, the following distortion are also calculated

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

for *idiv* = 0 to *nDiv* - 1, *icb* is the number of the codebook line.

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCodeBook* initialized in the function VQCodeBookInitAlloc.

The function then selects *nCand* candidates of *dist* and $dist_n$ with the minimum distortion measure.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCodeBook* or *pSrc* pointer is NULL.

VQMainSelect

Finds optimal indexes with minimal distortion.

Syntax

```
IppStatus ippsVQMainSelect_32f(const Ipp32f* pSrc, const Ipp32f* pWeights, int
    nDiv, const Ipp32s* pLengths, int nCand, Ipp32s** ppIndexCand, Ipps32s**
    ppSignCand, Ipp32s** ppIndx, Ipp32s** ppSign, IppsVQCodeBookState_32f**
    ppCodeBooks, int nCodeBooks);
```

Parameters

pSrc	Source vector to be quantized.
pWeights	Pointer to the vector of weights.
nDiv	Number of fragmentations of the *src* and *weights* vectors.
pLengths	Pointer to an array of lengths of fragmentations.

nCand	Number of input candidates.
ppIndexCand	Pointer to pointer to the input vectors of indexes of *nCand* minimum candidates.
ppSignCand	Pointer to pointer to the input vectors of signs of *nCand* minimum candidates.
ppIndx	Pointer to pointer to the output vectors of indexes.
ppSign	Pointer to pointer to the output vectors of signs.
ppCodeBooks	Pointer to pointer to the specified `IppsVQCodeBookState_32f` structures.
nCodeBooks	Number of codebooks.

Description

This function is declared in the `ippac.h` header file. The function restores vectors for all possible combinations of indexes across the specified number of codebooks then computes the distortion against the source vector. The function then returns the combination that provides for the minimal distortion.

The following formula serves for computing of the distortion for the given fragmentation *idiv* with the specified quantization vectors *icb*[*i*], where *i* is within the range of [0,*nCodeBooks*-1].

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp]-pSrcDiv[ismp])^2$$

where

$$rec[ismp] = \frac{\sum_{i=0}^{nCodeBooks-1} pSignCand[idiv*nCand+icb[i]]*ppTable[pIndexCand[idiv*nCand+icb[i]]][ismp]}{nCodeBooks}$$

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

  $$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCodeBook* initialized in the function VQCodeBookInitAlloc.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

VQIndexSelect

Finds optimal vector set for specified number of codebooks.

Syntax

```
IppStatus ippsVQIndexSelect_32f(const Ipp32f* pSrc, const Ipp32f* pWeights, int
    nDiv, const Ipp32s* pLengths,  int nCand, int** ppPolbits, Ipp32s** ppIndx,
    Ipp32s** ppSign, IppsVQCodeBookState_32f** ppCodeBooks, int nCodeBooks);
```

Parameters

pSrc	Source vector to be quantized.
pWeights	Pointer to the vector of weights.
nDiv	Number of fragmentations of the *src* and *weights* vectors.
pLengths	Pointer to an array of lengths of fragmentations.
nCand	Number of input candidates.
ppPolbits	Indicates whether one or two norms should be used to compute the optimal vector set for the specified number of codebooks.
ppIndx	Pointer to pointer to the output vectors of indexes.
ppSign	Pointer to pointer to the output vectors of signs.

- The value of 1 indicates that the minimal distortion appears when the norm is negative.
- The value of 0 indicates that the minimal distortion appears when the norm is positive.

ppCodeBooks	Pointer to pointer to the specified `IppsVQCodeBookState_32f` structures.
nCodeBooks	Number of codebooks.

Description

This function is declared in the `ippac.h` header file. The function computes *nCand* vectors for each codebook by the following formula

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

if *polbits*[*idiv*] is 1, the following distortion is also calculated

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

for `idiv=0` to *nDiv* -1, *icb* – number of codebook line.

The function subsequently restores vectors for all possible combinations of indexes across the specified number of codebooks then computes the distortion against the source vector. The function then returns the combination that provides for the minimal distortion. The following formula serves for computing of the distortion for the given fragmentation *idiv* with the specified quantization vectors *icb*[*i*], where *i* is within the range of [0,*nCodeBooks*-1].

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp] - pSrcDiv[ismp])^2$$

$$rec[ismp] = \frac{\sum_{i=0}^{nCodeBooks-1} pSignCand[idiv*nCand + icb[i]]*ppTable[pIndexCand[idiv*nCand + icb[i]]][ismp]}{nCodeBooks}$$

for idiv=0 to *nDiv* -1, *icb* – number of codebook line.

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCodeBook* initialized in the function VQCodeBookInitAlloc.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

NOTE. You can use this function instead of calling the functions VQPreliminarySelect and VQMainSelect to save time and memory cost.

VQReconstruction

Reconstructs vectors from indexes.

Syntax

```
IppStatus ippsVQReconstruction_32f(const Ipp32s** ppIndx, const Ipp32s**
    ppSign, const Ipp32s* pLengths, int nDiv, Ipp32f* pDst,
    IppsVQCodeBookState_32f** ppCodeBooks, int nCodeBooks);
```

Parameters

ppIndx	Pointer to pointer to an array of input vectors of indexes for each codebook.
ppSign	Pointer to pointer to an array of input vectors of signs for each codebook containing either 1 or -1.
pLengths	Pointer to an array of lengths of partitions of output vector.
pDst	Pointer to the reconstructed vector of spectrum values.
nDiv	Number of partitions of output vector.
ppCodeBooks	Pointer to pointer to an array of pointers to specified `IppsVQCodeBookState_32f` structures.
nCodeBooks	Number of codebooks.

Description

This function is declared in the `ippac.h` header file. The function reconstructs a vector divided on several partitions. A partition is defined as set of indexes of vectors from the specified number of codebooks, that is, one index for one partition from each codebook. To reconstruct the values of the output vector, the function calculates the arithmetic mean of vector values from the specified number of codebooks. The arrays *pIndx* and *pSign* contain the number of vectors from the codebook and the sign.

$$pDSt\left[\sum_{i=0}^{idiv-1} pLength[i] + j\right] = \frac{1}{nCodeBooks}$$

$$\sum_{k=0}^{nCodeBooks-1} pSign[k][i] \cdot pCodeBooks[k] \rightarrow table[pIndx[k][i]][j]$$

for $i=0$ to $nDiv-1$ and for $j = 0$ to $pLength[i]-1$.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

NOTE. Length of pDst array should be equal to

$$\sum_{k=0}^{nCodeBooks-1} pLengths[k]$$

MP3 Audio Coding Functions

The ISO/IEC 11172-3 MPEG-1, Layer III (also referred to as “MP3”) audio coding algorithms are widely used to compress stereophonic and dual-channel music signals. Well-suited for both transmission and storage applications, the MP3 algorithm delivers high-fidelity audio playback quality with bitrates as low as one-tenth of the original. As a result, the MP3 algorithm has become the *de facto* standard compression methodology for portable and handheld storage media, as well as for transmission of high-fidelity compressed audio over the Internet. MP3 encoder/decoder is widely used in music storage and audio recording.

Macros and Constants

The MP3 macro and constant definitions are listed in Table 10-2.

Table 10-2 MP3 Macro and Constant Definitions

Global Macro Name	Definition	Notes
IPP_MP3_GRANULE_LEN	576	The number of samples in one granule
IPP_MP3_V_BUF_LEN	512	V data buffers length (32-bit words)

Table 10-2 MP3 Macro and Constant Definitions (continued)

Global Macro Name	Definition	Notes
IPP_MP3_SF_BUF_LEN	40	Scale factor buffer length (8-bit words)
IPP_MP3_SFB_TABLE_LONG_LEN	138	Scale factor band table for long block length (16-bit words)
IPP_MP3_SFB_TABLE_SHORT_LEN	84	Scale factor band table for short block length (16-bit words)
IPP_MP3_ID_MPEG2	0	MPEG-2 frame identifier
IPP_MP3_ID_MPEG1	1	MPEG-1 frame identifier

Data Structures

The MP3 coding API includes several data structures.

The structure `IppMP3FrameHeader` contains the complete set of header information associated with one frame.

The structure `IppMP3SideInfo` contains the complete set of side information associated with one granule of one channel.

The structure `IppMP3HuffmanTable` contains the complete set of information associated with Huffman table.

The structure `IppMP3PsychoacousticModelTwoAnalysis` contains the outputs generated by the Intel IPP implementation of ISO/IEC 11172-3 psychoacoustic analysis model 2, including estimates of the masked thresholds and perceptual entropy associated with the current frame. Masked thresholds are represented in terms of Mask-to-Signal Ratios (MSRs).

The structure `IppMP3PsychoacousticModelTwoState` contains the state information associated with the Intel IPP implementation of ISO/IEC 11172-3 psychoacoustic analysis model 2 to facilitate coherent block processing.

The structure `IppMP3BitReservoir` contains the state information associated with the quantization bit reservoir.

Frame Header

```
typedef struct {
    int id;                    /* ID: 1 - MPEG-1, 0 - MPEG-2 */
```

```
    int layer;                  /* layer index: 0x3 - Layer I
                                                 0x2 - Layer II
                                                 0x1 - Layer III */
    int protectionBit;          /* CRC flag: 0 - CRC on, 1 - CRC off */
    int bitRate;                /* bit rate index */
    int samplingFreq;           /* sampling frequency index */
    int paddingBit;             /* padding flag: 0 - no padding, 1 -
                                 padding */
    int privateBit;             /* private_bit, not used */
    int mode;                   /* mono/stereo selection */
    int modeExt;                /* extension to mode */
    int copyright;              /* copyright or not: 0 - no, 1 - yes */
    int originalCopy;           /* original or copied: 0 - copy,
                                 1 - original */
    int emphasis;             /* flag indicating the type of de-emphasis */
    int CRCWord;                /* CRC-check word */

} IppMP3FrameHeader;
```

Side Information

```
typedef struct {
     int  part23Len;            /* number of main_data bits */
     int  bigVals;       /* half the number of Huffman code words whose
                          maximum amplitudes may be greater than 1 */
     int  globGain;             /* quantizes step size information */
     int  sfCompress;          /* number of bits used for scale factors */
     int  winSwitch;            /* window switch flag */
     int  blockType;            /* block type flag */
     int  mixedBlock;           /* flag: 0 - non-mixed block, 1 - mixed
                                 block */
     int pTableSelect[3];  /* Huffman table index for the 3 rectangle in
                            <big_values> field */
```

```
    int  pSubBlkGain[3];     /* gain offset from the global gain for
                             one subblock */
    int  reg0Cnt;            /* the number of scale factor bands in the
                             first region of <big_values> less one */
    int  reg1Cnt;            /* the number of scale factor bands in the
                             second region of <big_values> less one */
    int  preFlag;            /* flag indicating high frequency boost */
    int  sfScale;            /* scale factor scaling */
    int  cntlTabSel;         /* Huffman table index for the <count1>
                             field of quadruples */
  } IppMP3SideInfo;
```

MP3 Huffman Table Structure

```
typedef struct {
  int tableSize;       /* number of rows in table  */
  int linbits;        /* variable used for encode if the magnitude of
                         encoded value is greater or equal to 15 */
  int maxBitsValue;    /* maximum bit length of codewords */
  Ipp16u *pHcod;       /* pointer to Huffman code table */
  Ipp8u *pSlen;        /* pointer to Huffman length table */
} IppMP3HuffmanTable;
```

MP3 Psychoacoustic Model Two Analysis

```
typedef struct{
Ipp32s pMSR[36];/* MSRs for one granule/channel.
              For long blocks, elements 0-20 represent the thresholds
              associated with the 21 SFBs. For short blocks, elements
              0,3,6,..,33, elements 1,4,...,34, and elements 2,5,...,35,
              respectively, represent the thresholds associated with the
              12 SFBs for each of the 3 consecutive short blocks in one
              granule/channel. That is, the block thresholds are
              interleaved such that the thresholds are grouped by SFB.*/
Ipp32s PE;          /* Estimated perceptual entropy, one
                       granule/channel */
} IppMP3PsychoacousticModelTwoAnalysis;
```

Psychoacoustic Model Two State

```
typedef struct {
     Ipp64s pPrevMaskedThesholdLong[2][63]; /* long block masked
                              threshold history buffer; contains masked
                              threshold estimates for the threshold
                              calculation partitions associated with the
                              two most recent long blocks */

     Ipp64s pPrevMaskedThesholdShort[IPP_MP3_PSY_BAND_SHORT_NUM]; /*
                              short block masked threshold history
                              buffer; contains masked threshold
                              estimates for the threshold calculation
                              partitions associated with the two most
                              recent short blocks */

     Ipp32sc pPrevFFT[2][FIRST_6_CW];/* FFT history buffer; contains
                                     real and imaginary FFT components
                                     associated with the two most recent
                                     long blocks */

     Ipp32s  pPrevFFTMag[2][FIRST_6_CW];/* FFT magnitude history buffer;
                                        contains FFT component magnitudes
                                        associated with the two most
                                         recent long blocks */
     int nextPerceptualEntropy; /* PE estimate for next granule; one
                              granule delay provided for synchronization
                              with analysis filterbank */

     int nextBlockType; /* Expected block type for next granule; either
                        long (normal), short, or stop. Depending upon
                        analysis results for the granule following the
                        next, a long block could change to a start block,
                        and a stop block could change to a short block.
                        This buffer provides one granule of delay for
                        synchronization with the analysis filterbank */

     Ipp32s pNextMSRLong[21]; /* long block MSR estimates for next
                        granule. One granule delay provided for
                        synchronization with analysis filterbank */
```

```
    Ipp32s pNextMSRShort[36]; /* short block MSR estimates for next
                                 granule. One granule delay provided for
                                synchronization with analysis filterbank */
} IppMP3PsychoacousticModelTwoState;
```

MP3 Bit Reservoir

```
typedef struct {
    int BitsRemaining;        /* bits currently remaining in the
                                 reservoir */
    int MaxBits;              /* maximum possible reservoir size, in
                                 bits, determined as follows:
                                  min(7680-avg_frame_len, 2^9*8),
                                where: avg_frame_len is the average frame
                                length (in bits), including padding bits
                                and excluding side information bits */
} IppMP3BitReservoir;
```

MP3 Codec Enumerated Types

The Intel® IPP MP3 encoder and decoder APIs define enumerated data types that facilitate the synchronization and data transfer between the encoder and decoder components. As shown in Table 10-3, the MP3 codec API includes several enumerated types that provide semantic interpretations for frequently used constants.

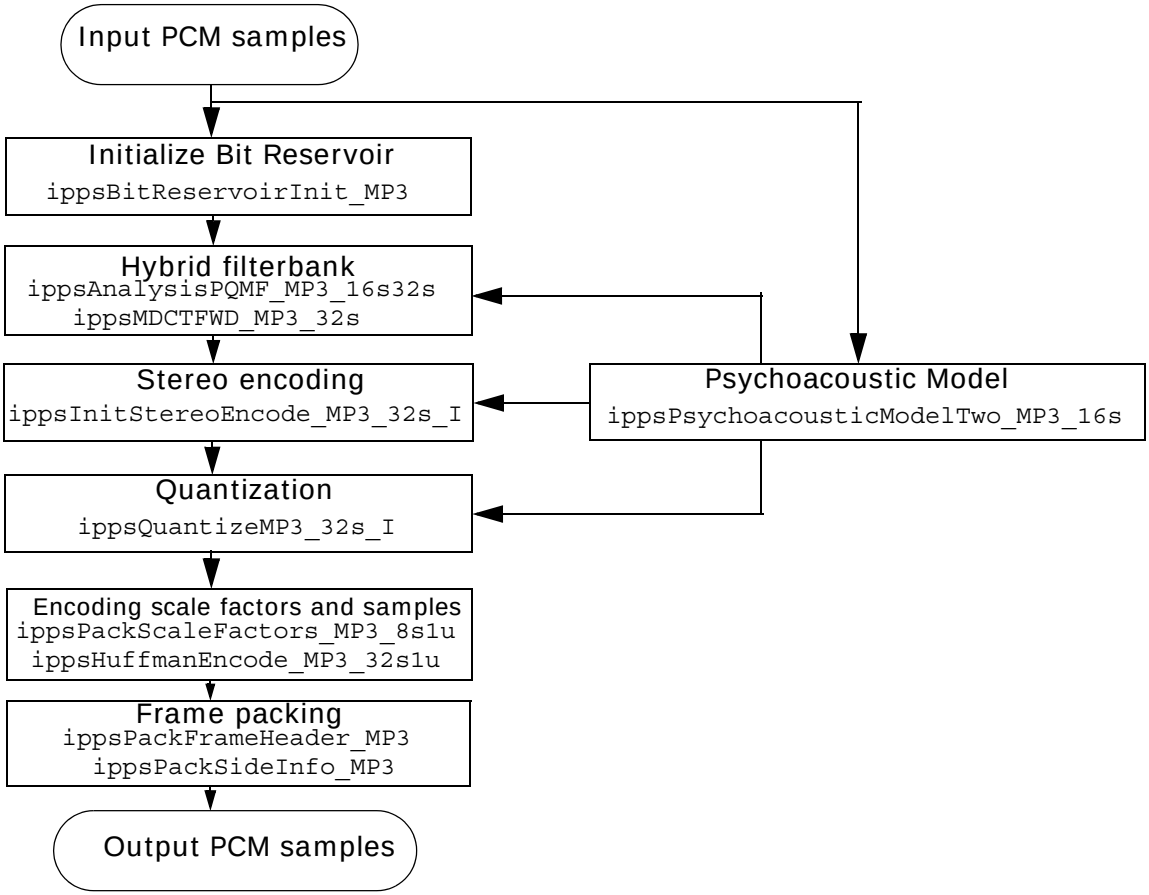
Table 10-3 MP3 Enumerated Data Types

Enumerated Type Name	Symbolic Values	Constant Value
IppMP3BitRate	ippMP3BitRateFree	0
	ippMP3BitRate32	1
	ippMP3BitRate40	2
	ippMP3BitRate48	3
	ippMP3BitRate56	4
	ippMP3BitRate64	5
	ippMP3BitRate80	6
	ippMP3BitRate96	7
	ippMP3BitRate112	8
	ippMP3BitRate128	9
	ippMP3BitRate160	10
	ippMP3BitRate192	11
	ippMP3BitRate224	12
	ippMP3BitRate256	13
	ippMP3BitRate320	14
IppMP3SampleRate	ippMP3SampleRate32000	2
	ippMP3SampleRate44100	0
	ippMP3SampleRate48000	1
IppMP3PcmMode	ippMP3NonInterleavedPCM	1
	ippMP3InterleavedPCM	2
IppMP3Emphasis	IppMP3EmphasisNone	0
	IppMP3Emphasis5015	1
	IppMP3EmphasisReserved	2
	IppMP3CCITTJ17	3

MP3 Audio Encoder

The MP3 encoder Application Programming Interface (API) provides a variety of capabilities, including bitstream packing functions and MP3 core encoding functions, See [ISO-11172. This chapter provides a reference guide to the Intel® Integrated Performance Primitives (Intel® IPP) MP3 audio encoder API. As shown in Figure 10-1, this API includes several functions as well as predefined macros and constants.

Figure 10-1 Intel® IPP MP3 Encoder API Flowchart



AnalysisPQMF_MP3

Implements stage 1 of MP3 hybrid analysis filterbank.

Syntax

```
IppStatus ippsAnalysisPQMF_MP3_16s32s(const Ipp16s* pSrcPcm,
    Ipp32s* pDstS, int pcmMode);
```

Parameters

pSrcPcm	Pointer to the start of the buffer containing the input PCM audio vector. The samples conform to the following guidelines: • must be in 16-bit, signed, little-endian, Q15 format • most recent 480 (512-32) samples should be contained in the vector *pSrcPcm*[*pcmMode**i], where *i* = 0,1,..,479 • samples associated with the current granule should be contained in the vector *pSrcPcm*[*pcmMode***j*], where *j* = 480,481,..,1055.
pcmMode	PCM mode flag. Communicates to PQMF filterbank the type of input PCM vector organization to expect: • *pcmMode* = 1 denotes non-interleaved PCM input samples • *pcmMode* = 2 denotes interleaved PCM input samples.
pDstS	Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples under the following index: *pDstXs*[32**i* + *sb*], where *i* = 0,1,…,17 is time series index *sb* = 0,1,…,31 is the subband index.

Description

The function is declared in the `ippac.h` file. This function implements the first stage of MP3 hybrid analysis filterbank. The function applies the critically sampled block PQMF analysis bank characterized by the 512-sample prototype window to a PCM input audio vector.

Call the `ippsAnalysisPQMF_MP3_16s32s` function 18 times per granule on each channel, that is, 36 times per channel on each frame.

NOTE. All coefficients are represented using the Q7.24 format

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcPcm* or *pDstXs* is NULL.
ippStsErr	Indicates an error when *pcmMode* exceeds [1,2].

AnalysisFilterInit_PQMF_MP3

Initializes PQMF MP3 analysis specification structure.

Syntax

```
IppStatus ippsAnalysisFilterInit_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3
    **ppFilterSpec, Ipp8u *pMemSpec);
```

Parameters

ppFilterSpec	Pointer to the pointer to the IppsFilterSpec_PQMF_MP3 specification structure. Only the main function uses this structure.
pMemSpec	Pointer to the area for the IppsFilterSpec_PQMF_MP3 specification structure.

Description

The function is declared in the ippac.h file. This function creates and initializes a polyphase quadrature mirror MP3 analysis specification structure.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.

AnalysisFilterInitAlloc_PQMF_MP3

Allocates memory for PQMF MP3 analysis specification structure and initializes it.

Syntax

```
IppStatus
    ippsAnalysisFilterInitAlloc_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3
    **ppFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the pointer to the `IppsFilterSpec_PQMF_MP3` specification structure. Only the main function uses this structure.

Description

The function is declared in the `ippac.h` file. This function allocates memory for polyphase quadrature mirror MP3 analysis specification structure, creates, and initializes this structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *ppFilterSpec* is `NULL`.

AnalysisFilterGetSize_PQMF_MP3

Returns size of PQMF MP3 analysis specification structure.

Syntax

```
IppStatus ippsAnalysisFilterGetSize_PQMF_MP3_32f(int *pSizeSpec);
```

Parameters

pSizeSpec	Address of the `IppsFilterSpec_PQMF_MP3` specification structure size value in bytes.

Description

The function is declared in the `ippac.h` file. This function gets size of the polyphase quadrature mirror MP3 analysis specification structure, creates, and initializes this structure and stores the result in *pSizeSpec*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSizeSpec* is `NULL`.

AnalysisFilterFree_PQMF_MP3

Frees memory allocated for PQMF MP3 analysis specification structure.

Syntax

```
IppStatus ippsAnalysisFilterFree_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3
    *pFilterSpec);
```

Parameters

pFilterSpec	Pointer to the `IppsFilterSpec_PQMF_MP3` specification structure.

Description

The function is declared in the `ippac.h` file. This function frees memory allocated for the polyphase quadrature mirror MP3 analysis specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFilterSpec* is `NULL`.
`ippStsContextMatchErr`	Indicates an error when *pFilterSpec* identifier is incorrect.

AnalysisFilter_PQMF_MP3

Tranforms PQMF MP3-processed subband signals into time domain samples.

Syntax

```
IppStatus ippsAnalysisFilter_PQMF_MP3_32f(const Ipp32f *pSrc, Ipp32f
    *pDst, const IppsFilterSpec_PQMF_MP3 *pFilterSpec, int mode);
```

Parameters

pSrc	Pointer to the input vector, holds time domain input samples.
pDst	Array of pointers, holds PQMF MP3-processed subband signals.
pFilterSpec	Pointer to the `IppsFilterSpec_PQMF_MP3` specification structure.
mode	Flag that indicates whether or not MP3 audio input channels should be interleaved: 1 - indicates no interleaving, 2 - indicates interleaving.

Description

The function is declared in the `ippac.h` file. This function performs the polyphase quadrature mirror filter bank, that is, a critically-sampled 32-channel PQMF analysis bank that generates 32-sample output block of IMDCT outputs for each 32 time-domain input sample. For each input block, the PQMF generates an output 32-sample block in the vector pointed to by *pDst*. If *mode* equals 2, the left and right channel input samples are interleaved, that is, `LRLRLR`, so that the left channel data is organized as follows: `pDst [2*i]`, `i` = 0 to 31. If *mode* equals 1, then the left and right channel inputs are not interleaved.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsBadArgErr`	Indicates no error. when *mode* exceeds [1,2].
`ippStsContextMatchErr`	Indicates an error when *pFilterSpec* identifier is incorrect.

MDCTFwd_MP3

Implements stage 2 of the MP3 hybrid analysis filterbank.

Syntax

```
IppStatus ippsMDCTFwd_MP3_32s(const Ipp32s* pSrc, Ipp32s* pDst, int blockType,
    int mixedBlock, IppMP3FrameHeader* pFrameHeader, Ipp32s* pOverlapBuf);
```

Parameters

pSrc	Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples that are indexed as follows: *pDstS*[32**i*+*sb*], where *i* = 0,1,…,17 is time series index, *sb* = 0,1,…,31 is the subband index. All coefficients are represented using the Q7.24 format.
blockType	Block type indicator: 0 stands for normal block 1 stands for start block 2 stands for short block 3 stands for stop block.
mixedBlock	Mixed block indicator: 0 stands for not mixed 1 stands for mixed.
pFrameHeader	Pointer to the `IppMP3FrameHeader` structure that contains the header associated with the current frame. Only MPEG-1 (*id* = 1) is supported.
pOverlapBuf	Pointer to the MDCT overlap buffer that contains a copy of the most recent 576-element block of PQMF bank outputs. Prior to processing a new audio stream with the analysis filterbank, all elements of the buffer *pOverlapBuf* should be initialized to the constant value 0.
pDst	Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank.

Description

The function is declared in the `ippac.h` file. This function implements the second stage of MP3 hybrid analysis filterbank by performing the following operations:

1. **Forward MDCT.** An appropriately arranged set of 12-point and/or 36-point forward Modified Discrete Cosine Transforms (MDCTs) is applied to the 18-sample spectral coefficient blocks generated on each of the 32 PQMF subbands during the first stage analysis.
2. **Aliasing reduction butterflies.** The butterflies specified in ISO/IEC 11172-3 are applied to the MDCT outputs in order to mitigate the aliasing artifacts introduced by cascading two critically sampled analysis filterbanks. Each of these introduces some non-negligible amount of interband aliasing.

The function `ippsMDCTFwd_MP3_32s` updates the 576-element MDCT overlap buffer *`pMDCTOverlap`*`[]`, the contents of which must be preserved between calls to facilitate coherent block processing. The function must be applied once per granule on each channel (that is, applied twice per channel on each frame).

NOTE. Input coefficients are represented in the Q7.24 format. Output coefficients are represented in the Q5.26 format.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the pointers *`pSrcXs`*, *`pDstXr`*, *`pFrameHeader`*, or *`pOverlapBuf`* is `NULL`.

PsychoacousticModelTwo_MP3

Implements ISO/IEC 11172-3 psychoacoustic model recommendation 2 to estimate masked threshold and perceptual entropy associated with a block of PCM audio input.

Syntax

```
IppStatus ippsPsychoacousticModelTwo_MP3_16s(const Ipp16s* pSrcPcm,
    IppMP3PsychoacousticModelTwoAnalysis* pDstPsyInfo, int* pDstIsSfbBound,
    IppMP3SideInfo* pDstSideInfo, IppMP3FrameHeader* pFrameHeader,
    IppMP3PsychoacousticModelTwoState* pFramePsyState, Ipp32s* pWorkBuffer,
    int pcmMode);
```

Parameters

pSrcPcm

Pointer to the start of the buffer containing the input PCM audio vector, the samples of which should conform to the following format specification:

- 16-bits per sample, signed, little-endian, Q15.
- *pSrcPcm* buffer should contain 1152 samples, that is, two granules of 576 samples each, if the parameter *pFrameHeader* `->` *mode* has the value 1 (mono), or 2304 samples, that is, two granules of 576 samples each, if the parameter *pFrameHeader* `->` *mode* has the value of 2 (stereo, dual mono).
- In the stereophonic case, the PCM samples associated with the left and right channels should be organized according to the *pcmMode* flag. Failure to satisfy any of the above PCM format and/or buffer requirements results in undefined model outputs.

pDstPsychoInfo

Pointer to the first element in a set of `PsychoacousticModelTwoAnalysis` structures. Each set member contains the MSR and PE estimates for one granule. The number of elements in the set is equal to the number of channels, with the outputs arranged as follows:
`(Analysis[0] = granule 1, channel 1), (…Analysis[1] = granule 1, channel 2), (…Analysis[2] = granule 2, channel 1), (…Analysis[3] = granule 2, channel 2).`

pDstIsSfbBound

If intensity coding has been enabled, *pDstIsSfbBound* points to the list of SFB lower bounds above which all spectral coefficients should be processed by the joint stereo intensity coding module. Since the intensity coding SFB lower bound is block-specific, the number of valid elements pointed to by *pDstIsSfbBound* varies depending upon the individual block types associated with each granule. In particular, the list of SFB bounds is indexed as follows:

pIsSfbBound`[3*`*gr*`]` for long block granules
pIsSfbBound`[3*`*gr* `+` *w*`]` for short block granules,

where *gr* is the granule index (0 indicates granule 1 and 1 indicates granule 2), and *w* is the block index (0 indicates block 1, 1 indicates block 2, 2 indicates block 3).

For example, given short-block analysis in granule 1 followed by long block analysis in granule 2, the list of SFB bounds would be generated in the following order:
pIsSfbBound`[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}.`

Only one SFB lower bound decision is generated for long block granules, whereas three are generated for short block granules. If both MS and intensity coding are enabled, then the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding has been enabled, then the SFB bound represents the lowest non-MS SFB.

pDstSideInfo Pointer to the updated set of `IppMP3SideInfo` structures associated with all granules and channels. The model updates the following fields in all set elements: *blockType*, *winSwitch*, and *mixedBlock*. The number of elements in the set is equal to 2 times the number of channels. Ordering of the set elements is the same as *pDstPsychoInfo*.

pFrameHeader Pointer to the updated `IppMP3FrameHeader` structure that contains the header associated with the current frame. The model updates the element *modeExt* to reflect the joint stereo coding mode decision. No other frame header fields are modified by this function.

pFramePsyState Pointer to the first element in a set of `IppMP3PsychoacousticModelTwoState` structures that contains the updated psychoacoustic model state information associated with both the current frame and next frame. The number of elements in the set is equal to the number of channels contained in the input audio. That is, a separate analysis is carried for each channel.

Prior to encoding a new audio stream, all elements of the psychoacoustic model state structure *pPsychoacousticModelState* should be initialized to contain the value 0.

In the signal processing domain, this could be accomplished using the function `ippsZero_16s` as follows:
`ippsZero_16s ((Ipp16s *) pPsychoacousticModelState,sizeof(IppMP3PsychoacousticModelTwoState)/sizeof(Ipp16s)).`

pWorkBuffer Pointer to the workspace buffer internally used by the psychoacoustic model for storage of intermediate results and other temporary data. The buffer length must be at least 25,200 bytes, that is, 6300 elements of type `Ipp32s`.

pcmMode PCM mode flag. Communicates the psychoacoustic model which type of PCM vector organization to expect:

- *pcmMode* = `1` denotes non-interleaved PCM input samples, that is, *pSrcPcm*`[0..1151]` contains the input samples associated with the left channel, and *pSrcPcm*`[1152..2303]` contains the input samples associated with the right channel.

- *pcmMode* = 2 denotes interleaved PCM input samples, that is, *pSrcPcm*[2*i] and *pSrcPcm*[2*i+1] contain the samples associated with the left and right channels, respectively, where *i* = 0,1,…,1151.

You can also use appropriately typecast elements ippMP3NonInterleavedPCM and ippMP3InterleavedPCM of the enumerated type IppMP3PcmMode as an alternative to the constants 1 and 2 for *pcmMode*.

Description

The function is declared in the ippac.h file. This function implements the ISO/IEC 11172-3 psychoacoustic model recommendation 2 to estimate the masked threshold and perceptual entropy associated with a block of PCM audio input. Quantization process uses model outputs to estimate a perceptually optimal bit allocation for the spectral coefficients generated by the analysis filterbank. The psychoacoustic model also controls stereophonic MS/intensity mode selection and processing as well as analysis filterbank block size switching. Given one frame of PCM input audio of 1152 samples per channel, that is, two granules of 576 samples each, the psychoacoustic model generates the following outputs:

1. **Estimated SFB (scale factor band) Mask-to-Signal ratios (MSRs)**. The model generates a vector of estimated MSRs for the 21 SFBs in long block mode and 12 SFBs for each of three consecutive blocks in short block mode.

   The MSR is derived from the masked threshold, which quantifies the simultaneous masking power associated with one granule/channel (576 samples) of input audio. Given the properties of the audio stimulus presented to the listener, this threshold essentially quantifies the granule-instantaneous modified threshold of hearing. Ideally, the threshold estimate should provide a frequency-dependent intensity (dB SPL) profile beneath which an average listener cannot perceive quantization noise or, for that matter, any other spectral energy.

   To estimate the masked threshold from a block of input audio, the function ippsPsych_MP3_16s implements the procedure recommended in Annex D.2 of ISO/IEC 11172-3.

   First, the output of a classical FFT-based spectral analysis is grouped into threshold calculation partitions that are organized to achieve analysis with sub-critical bandwidth resolution. On each threshold calculation partition, the model employs a weighted estimate of tone-like or noise-like signal behavior determined by an assessment of a spectral unpredictability across time to estimate masking power in each partition.

   Second, a spreading function is applied to model the spectral selectivity of the auditory system.

Finally, the estimated threshold is compared against the absolute threshold of hearing in quiet and the maximum of the two is assigned to the threshold calculation partition. Ultimately, in order to match its output to the bit allocation scheme of the quantization module, the model converts from the threshold calculation partition scale to a scale factor band (SFB) scale. One set of 21 SFB thresholds is generated for long blocks (576 samples), or three consecutive blocks of 12 SFB thresholds are generated for short blocks (192 samples).

To facilitate efficient quantization, the SFB thresholds are inverted and normalized by the signal energy and returned in a vector of SFB Mask-to-Signal ratios (MSRs). The estimated MSRs are returned in the `PsychoacousticModelTwoAnalysis` structure.

2. **Estimated perceptual entropy**. The model generates a perceptual entropy (PE) estimate for each granule. The PE quantifies the minimum number of bits required to represent the PCM samples of the granule with “perceptual transparency”. That is, without audible loss of quality for an average listener in comparison to the original, uncoded version.

   The estimated PE is derived from the masked threshold, in combination with classical assumptions about the minimum number of bits required to achieve a particular signal-to-noise ratio (SNR) target in each SFB, incremental per bit SNR improvement = +6 dB, where the minimum required SNR and hence minimum required number of bits for each SFB is derived from the signal-to-mask ratio (SMR).

   Perceptual entropy is used to control analysis filterbank block size switching, since sudden large PE increases are often associated with transient audio events that are prone to pre-echo distortion. The PE estimate is returned in the `PsychoacousticModelTwoAnalysis` structure.

3. **Analysis filterbank block size decision**. Using perceptual entropy and other indicators, the model determines whether or not the current granule is susceptible to pre-echo distortion. You should prefer using the short block mode when pre-echoes are likely and use long blocks in all other cases.

   In order to ensure selection of the appropriate block type, the decision incorporates single block look ahead switching logic. For example, if the current block type is long and the next block type is short, the current block type is changed from block type “long/normal” to block type “long/start” in order to guarantee seamless block processing upon mode switch. Similarly, if the current block type has been designated as “long/stop” and the next block type is determined to be “short”, the block switching logic changes the current block from “long/stop” back to “short” in order to avoid unnecessary mode switching. The block type decision is returned in the frame/granule `IppMP3SideInfo` structure.

4. **Joint stereophonic processing mode decision**. For 2-channel audio sources, the model evaluates interchannel correlations and other indicators in order to generate joint stereo LR/MS and/or intensity processing mode decisions. The joint stereo mode decision is returned in the `modeExt` field of the `IppMP3FrameHeader` structure.
5. **Intensity stereo coding SFB bound decision**. If intensity coding has been activated (see the preceding item joint stereophonic processing mode decision), the psychoacoustic model determines an appropriate lower SFB bound above which all spectral coefficients should be encoded using intensity mode stereophonic processing.

The psychoacoustic model performs analysis on a frame basis (1152 samples per channel), including two granules and up to two channels for either stereophonic or dual mono inputs. Valid lengths for both input and output vectors depend upon which mono or stereo channel modes have been enabled.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *pSrcPcm*, *pDstPsyInfo*, *pDstSideInfo*, *pDstIsSfbBound*, *pFrameHeader*, *pDstPsyState*, or *pWorkBuffer* is `NULL`.

JointStereoEncode_MP3

Transforms independent left and right channel spectral coefficient vectors into combined mid/side and/or intensity mode coefficient vectors suitable for quantization.

Syntax

```
IppStatus ippsJointStereoEncode_MP3_32s_I(Ipp32s* pSrcDstXrL, Ipp32s*
    pSrcDstXrR, Ipp8s* pDstScaleFactorR, IppMP3FrameHeader* pFrameHeader,
    IppMP3SideInfo* pSideInfo, int* pIsSfbBound);
```

Parameters

pSrcDstXrL	Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank for the left channel of input audio. All coefficients are represented using the Q5.26 format.

pSrcDstXrR	Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank for the right channel of input audio. All coefficients are represented using the Q5.26 format.
pFrameHeader	Pointer to the `IppMP3FrameHeader` structure that contains the header information associated with the current frame. Upon function entry, the structure fields `samplingFreq`, `id`, `mode`, and `modeExt` should contain, respectively, the sample rate associated with the current input audio, the algorithm *id* (MPEG-1 or MPEG-2), and the joint stereo coding commands generated by the psychoacoustic model. All other *pFrameHeader* fields are ignored. Only MPEG-1 (`id = 1`) is supported.
pSideInfo	Pointer to the pair of `IppMP3SideInfo` structures associated with the channel pair to be jointly encoded. The number of elements in the set is 2, and ordering of the set elements is as follows: *pSideInfo*`[0]` describes channel 1, and *pSideInfo*`[1]` describes channel 2. Upon the function entry, the `blockType` side information fields for both channels should reflect the analysis modes (short or long block) selected by the psychoacoustic model on each channel. All other fields in the *pSideInfo*`[0]` and *pSideInfo*`[1]` structures are ignored.
pIsSfbBound	Pointer to the list of intensity coding SFB lower bounds for both channels of the current granule above which all L/R channel spectral coefficients are combined into an intensity-coded representation. The number of elements depends on the block type associated with the current granule. For short blocks, the SFB bounds are represented in the following order: *pIsSfbBound*`[0]` describes block 1, *pIsSfbBound*`[1]` describes block 2, and *pIsSfbBound*`[2]` describes block 3. For long blocks, only a single SFB lower bound decision is required and is represented in *pIsSfbBound*`[0]`. If both MS and intensity coding are enabled, the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding is enabled, the SFB bound represents the lowest non-MS SFB.
pDstScaleFactorR	Pointer to the vector of scalefactors associated with one granule of the right/S channel. If intensity coding has been enabled by the psychoacoustic model above a certain SFB lower bound, as indicated by the frame header and the vector pointed to by `pIsSfbBound`, the function `StereoEncode_MP3_32s_I` updates with the appropriate scalefactors those elements of *pDstScaleFactorR*`[]` that are associated with intensity coded scale factor bands. Other SFB entries in the scale factor vector remain unmodified. The length of the vector referenced by

pDstScaleFactorR varies as a function of block size. The vector contains 21 elements for long block granules, or 36 elements for short block granules.

Description

The function is declared in the `ippac.h` file. This function transforms the independent left and right channel spectral coefficient vectors into combined mid/side (MS) and/or intensity (IS) mode coefficient vectors suitable for quantization. If MS coding is enabled (*pFrameHeader* -> *modeExt* & 0x10 == 1), the left and right channels are converted to Mid and Side channels as follows:

$$M = \frac{L+R}{\sqrt{2}} \quad S = \frac{L-R}{\sqrt{2}}$$

This function is called on dual granule basis. You should call it for every granule/2 channels.

If intensity coding is enabled (*pFrameHeader* -> *modeExt* & 0x01 = 1), the left channel carries the intensity data for SFBs above the SFB intensity lower bound and the right channel above the SFB lower bound is cleared. All coefficients are set to 0.

$$L = L + R, R = 0$$

In order to facilitate energy-proportional recovery of the left and right spectral coefficients at the decoder, an intensity energy scale factor, *is_pos*, is transmitted in place of the right channel scale factor since the right channel spectral coefficients above the SFB bound are eliminated. The energy normalization constant is derived from the L/R SFB energy ratios and then transformed to improve its quantization properties. That is,

$$is_pos = n\,\mathrm{int}\left(\frac{12}{\pi}\arctan\left(\sqrt{\frac{L_energy}{R_energy}}\right)\right)$$

where *L_energy* and *R_energy* are, respectively, the SFB energies associated with the spectral coefficients of the left and right channels. At the decoder, the *is_pos* scale factor is used to apportion jointly coded signal energy between the left and right channels in a manner consistent with the distribution of signal energy prior to joint coding. The *is_pos* intensity scalefactors are returned in the vector *pDstScalefactorR*.

On each granule, joint stereo coding is applied once per channel pair (576 samples per granule on each channel). The function `JointStereoEncode_MP3_32s_I` must therefore be called once per granule, or twice per frame.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the pointers *pSrcDstXrL*, *pSrcDstXrR*, *pDstScaleFactorR*, *pFrameHeader*, *pSideInfo*, or *pIsSfbBound* is NULL.
ippStsMP3SideInfoErr	Indicates an error if pSideInfo[0].blockType != pSideInfo[1].blockType when IS or MS is used.

Quantize_MP3

Quantizes spectral coefficients generated by analysis filterbank.

Syntax

```
IppStatus ippsQuantize_MP3_32s_I(Ipp32s* pSrcDstXrIx, Ipp8s* pDstScalefactor,
    int* pDstScfsi, int* pDstCount1Len, int* pDstHufSize, IppMP3FrameHeader*
    pFrameHeader, IppMP3SideInfo* pSideInfo,
    IppMP3PsychoacousticModelTwoAnalysis* pPsychoInfo,
    IppMP3PsychoacousticModelTwoState* pFramePsyState, IppMP3BitReservoir*
    pResv, int meanBits, int* pIsSfbBound, Ipp32s* pWorkBuffer);
```

Parameters

pSrcDstXrIx

Pointer to the set of unquantized spectral coefficient vectors generated by the analysis filterbank and optionally processed by the joint stereo coding module for one frame. The set of unquantized coefficients should be indexed as follows:

- *pSrcDstXrIx*[*gr**1152 + *ch**576 + *i*] for stereophonic and dual-mono input sources
- *pSrcDstXrIx*[*gr**576 + *i*] for monaural, that is, input sources,

where

- *i* = 0,1,...,575 is the spectral coefficient index
- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

Depending on which type of joint coding has been applied, if any, the coefficients for each channel could be associated with L/R, M/S, and/or intensity representations of the input audio. All coefficients should be represented using the Q5.26 format.

pDstScaleFactor Pointer to the output set of scalefactors generated during the quantization process. These scalefactors determine the quantizer granularity. Scale factor vector lengths depend on the block mode associated with each granule. The order of the elements is:

1. (granule 1, channel 1)
2. (granule 1, channel 2)
3. (granule 2, channel 1)
4. (granule 2, channel 2).

Given this general organization, the side information for each granule/channel in conjunction with the flags contained in the vector *pDstScfsi* can be used to determine the precise scale factor vector indices and lengths.

pDstScfsi Pointer to the output vector of scale factor selection information. This vector contains a set of binary flags that indicate whether or not scalefactors are shared across granules of a frame within predefined scale factor selection groups. For example, bands 0,1,2,3,4,5 form one group; bands 6,7,8,9,10 form a second group, as defined in ISO/IEC 11172-3. The vector is indexed as follows: *pDstScfsi*[*ch*] [*scfsi_band*], where

- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.
- *scfsi_band* is the scale factor selection group number:

  group 0 includes SFBs 0-5,
  group 1 includes SFBs 6-10,
  group 2 includes SFBs 11-15,
  group 3 includes SFBs 16-20.

pDstCount1Len Pointer to an output vector of *count1* region length specifiers. For the purposes of Huffman coding spectral coefficients of a higher frequency than the *bigvals* region, the *count1* parameter indicates the size of the region in which spectral samples can be combined into quadruples for which all elements are of magnitude less than or equal to 1. The vector contains 2**nchan*, elements and is indexed as follows: *pDstCount1Len*[*gr*nchan + ch*], where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- *nchan* is the number of channels.
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

pDstHufSize

Pointer to an output vector of Huffman coding bit allocation specifiers. For each granule/channel, the specifiers indicate the total number of Huffman bits required to represent the quantized spectral coefficients in the *bigvals* and *count1* regions.

Whenever necessary, each *HufSize* bit count is augmented to include the number of bits required to manage the bit reservoir. For frames in which the reservoir has reached the maximum capacity, the quantizer expends the surplus bits by padding with additional bits the Huffman representation of the spectral samples. The *HufSize* result returned by the quantizer reflects these padding requirements. That is, *HufSize*[i] is equal to the sum of the number of bits required for Huffman symbols and the number of padding bits. The vector contains 2**nchan*, elements and is indexed as follows: *pDstHufSize*[gr**nchan*+*ch*], where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- *nchan* is the number of channels.
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

pFrameHeader

Pointer to the IppMP3FrameHeader structure that contains the header information associated with the current frame. Upon the function entry, the structure fields samplingFreq, id, mode, and modeExt should contain, respectively, the sample rate associated with the current input audio, the algorithm id, that is, MPEG-1 or MPEG-2, and the joint stereo coding commands generated by the psychoacoustic model. All other **pFrameHeader* fields are ignored. Only MPEG-1 (*id* = 1) is supported.

pSideInfo

Pointer to the set of IppMP3SideInfo structures associated with all granules and channels. The set should contain 2**nchan*, elements and should be indexed as follows: *pSideInfo*[*gr***nchan* + *ch*], where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- *nchan* is the number of channels.
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

Upon the function entry, in all set elements the structure fields *blockType*, *mixedBlock*, and *winSwitch* should contain, respectively, the block type indicator (*start*, *short*, or *stop*), filter bank mixed block analysis mode specifier, and window switching flags (*normal* or *blockType*) associated with the current

input audio. All other **pSideInfo* fields are ignored upon the function entry and updated upon function exit, as described below under the description of output arguments.

pPsychoInfo Pointer to the first element in a set of `PsychoacousticModelTwoAnalysis` structures associated with the current frame. Each set member contains the MSR and PE estimates for one channel of one granule. The set should contain `2*nchan`, elements and is indexed as: `pPsychoaInfo[gr*nchan+ch]`, where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2
- *nchan* is the number of channels
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

pFramePsyState Pointer to the first element in a set of `IppMP3PsychoacousticModelTwoState` structures that contains the psychoacoustic model state information associated with both the current frame and next frame. The number of elements in the set is equal to the number of channels contained in the input audio. That is, a separate analysis is carried for each channel. The quantizer uses the frame type look ahead information *nextBlockType* to manage the bit reservoir. All other structure elements are ignored by the quantizer.

pResv Pointer to the `IppMP3BitReservoir` structure that contains the bit reservoir state information. Upon the function entry, all structure fields should contain valid data.

meanBits The number of bits allocated on an average basis for each frame of spectral coefficients and scalefactors given the target bit rate (kilobits per second) specified in the frame header. This number excludes the bits allocated for the frame header and side information. The quantizer uses *meanBits* as a target allocation for the current frame. Given perceptual bit allocation requirements greater than this target, the quantizer makes use of the surplus bits held in the bit reservoir to satisfy frame-instantaneous demands. Similarly, given perceptual bit allocation requirements below this target, the quantizer will store surplus bits in the bit reservoir for use by future frames.

pIsSfbBound Pointer to the list of SFB lower bounds above which all L/R channel spectral coefficients have been combined into an intensity-coded representation. The number of valid elements pointed to by `pIsSfbBound` depends upon the block types associated with the granules of the current frame.

In particular, the list of SFB bounds pointed to by `pIsSfbBound` is indexed as follows: *pIsSfbBound*[3*gr] for long block granules, and *pIsSfbBound*[3*gr + w] for short block granules, where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2
- *w* is the block index. 0 stands for block 1, 1 stands for block 2, 2 stands for block 3.

For example, given short-block analysis in granule 1 followed by long block analysis in granule 2, the list of SFB bounds would be expected in the following order:
pIsSfbBound[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}.

If a granule is configured for long block analysis, then only a single SFB lower bound decision is expected, whereas three are expected for short block granules. If both MS and intensity coding have been enabled, then the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding has been enabled, then the SFB bound represents the lowest non-MS SFB.

pWorkBuffer Pointer to the workspace buffer internally used by the quantizer for storage of intermediate results and other temporary data. The buffer length should be at least 2880 bytes, that is, 720 words of 32-bits each.

Description

The function is declared in the `ippac.h` file. This function quantizes the spectral coefficients generated by the analysis filterbank such that the resulting distortion, that is, quantization noise, is shaped to match a profile derived from the masked thresholds estimated by the psychoacoustic model.

Along with meeting perceptual distortion criteria, the quantizer simultaneously adjusts the overall bit allocation to achieve a fixed bit rate target. In accordance with the ISO/IEC 11172-3 recommendation, a bit reservoir is maintained in order to meet instantaneous peak rate demands without violating on an average basis the fixed rate constraint. Surplus bits are deposited in the reservoir during frames with lower than average perceptual bit rate requirements. Supplementary bits are withdrawn from the reservoir to satisfy frames with higher-than-average perceptual bit allocation requirements.

The quantizer manages the reservoir and overall bit allocation such that a constant average rate constraint is satisfied. The quantizer operates on a complete frame of data, that is, two granules and either one or two channels, and it should therefore be called once per frame.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *pSrcDstXrIx*, *pDstScaleFactor*, *pDstScfsi*, *pDstCount1Len*, *pDstHufSize*, *pFrameHeader*, *pSideInfo*, *pPsychInfo*, *pFramePsyState*, *pResv*, *pIsSfbBound*, or *pWorkBuffer* is NULL.
ippStsMP3SideInfo	Indicates an error when *pSideInfo->winSwitch* and *pSideInfo->mixedBlock* are both defined.
ippStsMP3FrameHeaderErr	Indicates an error when *pFrameHeader->samplingFreq* or *pFrameHeader->id* is not correct.

PackScaleFactors_MP3

Applies noiseless coding to scalefactors and packs output into bitstream buffer.

Syntax

```
IppStatus ippsPackScaleFactors_MP3_8s1u(const Ipp8s* pSrcScalefactor, Ipp8u**
    ppBitStream, int* pOffset, IppMP3FrameHeader* pFrameHeader, IppMP3SideInfo*
    pSideInfo, int* pScfsi, int granule, int channel);
```

Parameters

pSrcScaleFactor	Pointer to a vector of scalefactors generated during the quantization process for one channel of one granule. Scale factor vector lengths depend on the block mode. Short block granule scale factor vectors contain 36 elements, or 12 elements for each subblock. Long block granule scale factor vectors contain 21 elements. Thus short block scale factor vectors are indexed as follows: *pSrcScaleFactor*[*sb**12+*sfb*], where

- *sb* is the subblock index. 0 stands for subblock 1, 1 stands for subblock 2, 2 stands for subblock 3.
- *sfb* is the scale factor band index (0-11).

Long block scale factor vectors are indexed as follows: *pSrcScaleFactor*[*sfb*], where *sfb* is the scale factor band index (0-20).
The associated side information for an individual granule/channel can be used to select the appropriate indexing scheme.

ppBitStream Updated bitstream byte pointer. This parameter points to the first available bitstream buffer byte immediately following the bits generated by the scale factor Huffman encoder and sequentially written into the stream buffer. The scale factor bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3.

pOffset Updated bitstream bit pointer. The *pOffset* parameter indexes the next available bit in the next available byte referenced by the updated bitstream buffer byte pointer *ppBitStream*. This parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit.

pFrameHeader Pointer to the IppMP3FrameHeader structure for this frame. Upon the function entry, the structure fields *id* and *modeExt* should contain, respectively, the algorithm *id* (MPEG-1 or MPEG-2) and the joint stereo coding commands generated by the psychoacoustic model. All other *pFrameHeader* fields are ignored. Only MPEG-1 (*id* = 1) is supported.

pSideInfo Pointer to the IppMP3SideInfo structure for the current granule and channel. Upon function entry, the structure fields *blockType*, *mixedBlock*, and *sfCompress* should contain, respectively, the block type indicator *start*, *short*, or *stop*, filter bank mixed block analysis mode specifier, and scale factor bit allocation. All other **pSideInfo* fields are ignored by the scale factor encoder.

pScfsi Pointer to the scale factor selection information table that contains the set of binary flags that indicate whether scalefactors are shared across granules of a frame within the predefined scale factor selection groups.

For example, bands 0,1,2,3,4,5 form one group and bands 6,7,8,9,10 form a second group (as defined in ISO/IEC 11172-3). The vector is indexed as follows: *pScfsi*[*ch*][*scfsi_band*], where

- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.
- *scfsi_band* is the scale factor selection group number:

	group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, group 3 includes SFBs 16-20.
granule	Index of the current granule. 0 stands for granule 1, 1 stands for granule 2.
channel	Index of the current channel. 0 stands for channel 1, 1 stands for channel 2.

Description

The function is declared in the `ippac.h` file. This function applies noiseless coding to the scalefactors and then packs the output into the bitstream buffer. This function operates on one channel of one granule at a time. Therefore, you should call this function once for each channel of each granule.

The resulting bitstream is fully compliant with the syntax specified in ISO/IEC 11172-3.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsMP3SideInfoErr`	Indicates an error when *`pFrameHeader->id`* `== IPP_MP3_ID_MPEG1` and *`pSideInfo->sfCompress`* exceeds `[0..15]`; *`pFrameHeader->id`* `== IPP_MP3_ID_MPEG2` and *`pSideInfo->sfCompress`* exceeds `[0..511]`.
`ippStsMP3FrameHeaderErr`	Indicates an error when *`pFrameHeader->id`* `== IPP_MP3_ID_MPEG2` and *`pFrameHeader->modeExt`* exceeds `[0..3]`.

HuffmanEncode_MP3

Applies lossless Huffman encoding to quantized samples and packs output into bitstream buffer.

Syntax

```
IppStatus ippsHuffmanEncode_MP3_32s1u(Ipp32s* pSrcIx, Ipp8u** ppBitStream, int*
    pOffset, IppMP3FrameHeader* pFrameHeader, IppMP3SideInfo* pSideInfo, int
    count1Len, int hufSize);
```

Parameters

pSrcIx	Pointer to the quantized samples of a granule. The buffer length is 576. Depending on which type of joint coding has been applied, if any, the coefficient vector might be associated with either the L, R, M, S, and/or intensity channel of the quantized spectral data.
ppBitStream	Updated bitstream byte pointer. The parameter `*ppBitStream` points to the first available bitstream buffer byte immediately following the bits generated by the spectral coefficient Huffman encoder and sequentially written into the stream buffer. The Huffman symbol bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3.
pOffset	Updated bitstream bit pointer. The *pOffset* parameter indexes the next available bit in the next available byte referenced by the updated bitstream buffer byte pointer *ppBitStream*. This parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit.
pFrameHeader	Pointer to the `IppMP3FrameHeader` structure for this frame. The Huffman encoder uses the frame header id field in connection with the side information (as described below) to compute the Huffman table region boundaries for the big value spectral region. The Huffman encoder ignores all other frame header fields.Only MPEG-1 (*id* = 1) is supported.
pSideInfo	Pointer to the `IppMP3SideInfo` structure for the current granule and channel. The structure elements `bigVals`, `pTableSelect[0]-[2]`, `reg0Cnt`, and `reg1Cnt` are used to control coding of spectral coefficients in the big value region. The structure element `cnt1TabSel` is used to select the appropriate Huffman table for the (-1,0,+1)-valued 4-tuples in the *count1* region.

For detailed descriptions of all side information elements, see the structure definition header file.

count1Len The count1 region length specifier. Indicates the number of spectral samples for the current granule/channel above the big value region that can be combined into 4-tuples in which all elements are of magnitude less than or equal to 1.

hufSize Huffman coding bit allocation specifier. Indicates the total number of bits that are required to represent the Huffman-encoded quantized spectral coefficients for the current granule/channel in both the *bigvals* and *count1* regions.

Whenever necessary, this bit count should be augmented to include the number of bits required to manage the bit reservoir. For frames in which the reservoir has reached maximum capacity, the surplus bits are expended by padding with additional bits the Huffman representation of the spectral samples.

The *HufSize* result returned by the function Quantize_MP3_32s_I reflects these padding requirements. That is, *HufSize*[*i*] is equal to the total of the number of bits required for Huffman symbols and the number of padding bits.

Description

The function is declared in the ippac.h file. This function applies lossless Huffman encoding to the quantized samples and packs the output into the bitstream buffer.

The function encodes one granule at a time, and therefore must be called once for each granule of each channel.

The resulting bitstream is fully compliant with ISO/IEC 11172-3.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsBadArgErr	Indicates an error when *pOffset* exceeds [0,7].
ippStsMP3SideInfoErr	Indicates an error when *pSideInfo*->bigVals*2>IPP_MP3_GRANULE_LEN (pSideInfo->reg0Cnt + pSideInfo->reg1Cnt + 2) >= 23, *pSideInfo->cnt1TabSel* exceeds [0,1], *pSideInfo->pTableSelect*[i] exceeds [0..31].

ippStsMP3FrameHeader	Indicates an error when *pFrameHeader*->id != IPP_MP3_ID_MPEG1 *pFrameHeader*->layer != 1 *pFrameHeader*->*samplingFreq* exceeds [0..2].

PackFrameHeader_MP3

Packs the content of the frame header into the bitstream.

```
IppStatus ippsPackFrameHeader_MP3 (IppMP3FrameHeader* pSrcFrameHeader, Ipp8u**
    ppBitStream);
```

Parameters

pSrcFrameHeader	Pointer to the IppMP3FrameHeader structure. This structure contains all the header information associated with the current frame. All structure fields must contain valid data upon function entry.
ppBitStream	Updated bitstream byte pointer. The parameter *ppBitStream* points to the first available bitstream buffer byte immediately following the packed frame header bits. The frame header bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3.

Description

The function is declared in the ippac.h file. This function packs the content of the frame header into the bitstream.

The resulting bitstream is fully compliant with the syntax specified in ISO/IEC 11172-3.

The function should be called once per frame.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcFrameHeader* or *ppBitStream* is NULL.

PackSideInfo_MP3

Packs the side information into the bitstream buffer.

Syntax

```
IppStatus ippsPackSideInfo_MP3(IppMP3SideInfo* pSrcSideInfo, Ipp8u** ppBitStream,
    int mainDataBegin, int privateBits, int* pSrcScfsi, IppMP3FrameHeader*
    pFrameHeader);
```

Parameters

pSrcSideInfo	Pointer to the `IppMP3SideInfo` structures. This should contain twice the channel number of elements. The order is the following: • granule 1, channel 1 • granule 1, channel 2 • granule 2, channel 1 • granule 2, channel 2. All fields of all set elements should contain valid data upon the function entry.
ppBitStream	Updated bitstream byte pointer. The parameter `*`*ppBitStream* points to the first available bitstream buffer byte immediately following the packed side information bits. The frame header bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3:1993.
mainDataBegin	Negative bitstream offset, in bytes. The parameter value is typically the number of bytes remaining in the bit reservoir before the start of quantization for the current frame. When computing *mainDataBegin*, you should exclude the header and side information bytes. The side information formatter packs the 9-bit value of *mainDataBegin* into the `main_data_begin` field of the output bitstream.
privateBits	Depending on the number of channels, the function extracts the appropriate number of least significant bits from the parameter *privateBits* and packs them into the `private_bits` field of the output bitstream. The ISO/IEC 11172-3 bitstream syntax reserves a channel-dependent number of application-specific (private) bits in the layer III bitstream audio data section immediately following

	the parameter `main_data_begin`. See ISO/IEC 11172-3:1993. For dual- and single-channel streams, respectively, three and five bits are reserved.
pSrcScfsi	Pointer to the scale factor selection information table. This vector contains a set of binary flags that indicate whether scalefactors are shared across granules of a frame within predefined scale factor selection groups. For example, bands 0,1,2,3,4,5 form one group and bands 6,7,8,9,10 form the second group, as defined in ISO/IEC 11172-3 [2]. The vector is indexed as *pDstScfsi*[*ch*][*scfsi_band*], where • *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2. • *scfsi_band* is the scale factor selection group number: group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, group 3 includes SFBs 16-20.
pFrameHeader	Pointer to the `IppMP3FrameHeader` structure. Only MPEG-1 (*id* = 1) is supported. Upon the function entry, the structure fields `id`, `mode`, and `layer` should contain, respectively, the algorithm `id` (MPEG-1 or MPEG-2), the mono or stereo mode, and the MPEG layer specifier. All other *pFrameHeader* fields are ignored.

Description

The function is declared in the `ippac.h` file. This function packs the side information into the bitstream buffer.

The resulting bitstream is fully compliant with the syntax specified in ISO/IEC 11172-3.

This function should be called once per frame.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrcSideInfo*, *ppBitStream*, *pSrcScfsi*, or *pFrameHeader* is `NULL`.
`ippStsMP3FrameHeaderErr`	Indicates an error when *pFrameHeader*->id != IPP_MP3_ID_MPEG1 *pFrameHeader*->id != IPP_MP3_ID_MPEG2 *pFrameHeader*->*layer* != 1 *pFrameHeader*->*mode* exceeds [0..3].

BitReservoirInit_MP3

Initializes all elements of the bit reservoir state structure.

Syntax

```
IppStatus ippsBitReservoirInit_MP3(IppMP3BitReservoir* pDstBitResv,
    IppMP3FrameHeader* pFrameHeader);
```

Parameters

pFrameHeader	Pointer to the `IppMP3FrameHeader` structure that contains the header information associated with the current frame. The frame header fields `bitRate` and `id`, bit rate index and algorithm identification, respectively, must contain valid data prior to calling the function `BitReservoirInit_MP3` since both are used to generate the bit reservoir initialization parameters. All other frame header parameters are ignored by the bit reservoir initialization function. Only MPEG-1 (*id* = 1) is supported.
pDstBitResv	Pointer to the initialized `IppMP3BitReservoir` state structure. The structure element `BitsRemaining` is initialized as 0. The structure element `MaxBits` is initialized to reflect the maximum number of bits that can be contained in the reservoir at the start of any given frame. The appropriate value of `MaxBits` is directly determined by the selected algorithm (MPEG-1 or MPEG-2) and the stream bit rate indicated by the rate index parameter *pFrameHeader*.bitRate.

Description

The function is declared in the `ippac.h` file. This function initializes all elements of the bit reservoir state structure based on the coding algorithm (MPEG-1 or MPEG-2) and the average per frame bit allocation specified in the frame header.

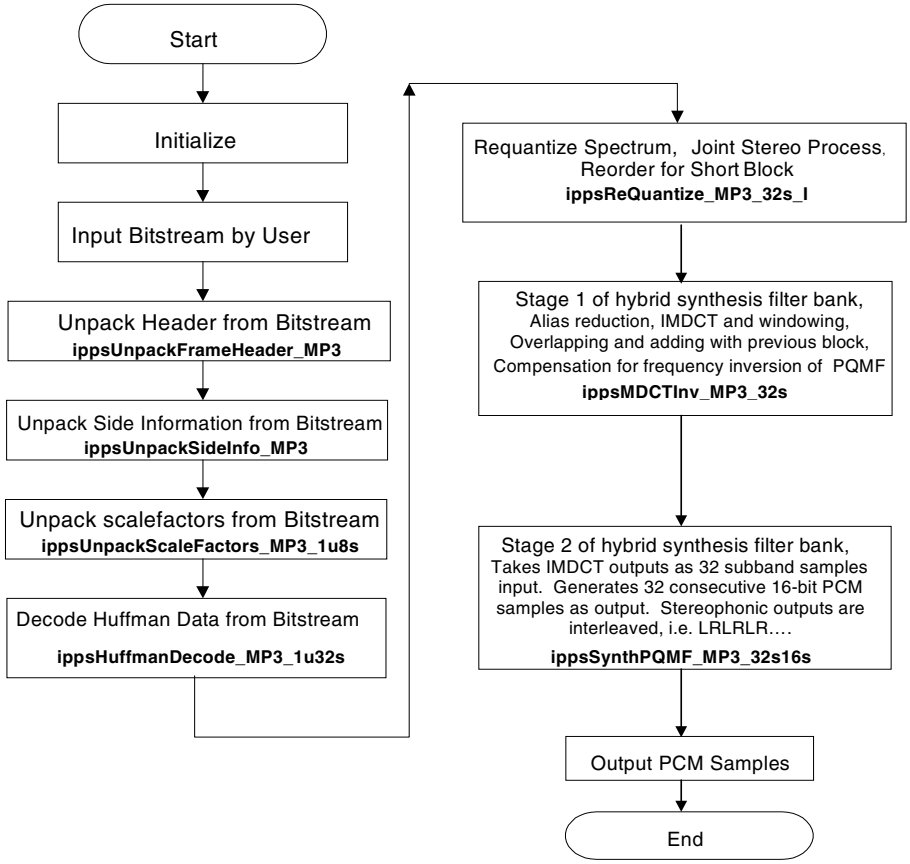
Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pDstBitResv* or *pFrameHeader* is `NULL`.
`ippStsMP3FrameHeaderErr`	Indicates an error when *pFrameHeader->id* != `IPP_MP3_ID_MPEG1`.

MP3 Audio Decoder

This section describes Intel IPP functions for assembling audio decoders compliant with the layer III portions of the ISO/IEC 11172-3 MPEG-1 and ISO/IEC 13818-3 MPEG-2 audio specifications (see [ISO]) and provides a reference guide to the Intel IPP MP3 Application Programming Interface (API). As shown in Figure 10-2, the MP3 API consists of seven functions, two data structures, and several pre-defined constants and macros.

Figure 10-2 MP3 Decoder API



UnpackFrameHeader_MP3

Unpacks audio frame header.

Syntax

```
IppStatus ippsUnpackFrameHeader_MP3(Ipp8u** ppBitStream, IppMP3FrameHeader*
    pFrameHeader);
```

Parameters

ppBitStream	Pointer to the pointer to the first byte of the MP3 frame header (**ppBitstream* will be updated in the function).
pFrameHeader	Pointer to the MP3 frame header structure.

Description

The function `ippsUnpackFrameHeader_MP3` is declared in the `ippac.h` file. This function unpacks the audio frame header. If cyclic redundancy check (CRC) is enabled, this function also unpacks the CRC word.

Before calling `ippsUnpackFrameHeader_MP3`, you should locate the bit stream synchronization word and ensure that **ppBitStream* points to the first byte of the 32-bit frame header.

If CRC is enabled, the assumption is that the 16-bit CRC word is adjacent to the 32-bit frame header, as defined in the MP3 standard. Before returning to the caller, the function updates the pointer **ppBitStream* so that it references the next byte after the frame header or the CRC word.

The first byte of the 16-bit CRC word is stored in *pFrameHeader*`->CRCWord(15:8)`, and the second byte is stored in *pFrameHeader*`->CRCWord(7:0)`.

The function does not detect corrupted frame headers.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *ppBitStreamp* or *FrameHeader* is `NULL`.

UnpackSideInfo_MP3

Unpacks side information from input bitstream for use during decoding of associated frame.

Syntax

```
IppStatus ippsUnpackSideInfo_MP3(Ipp8u** ppBitStream, IppMP3SideInfo*
    pDstSideInfo, int* pDstMainDataBegin, int* pDstPrivateBits, int* pDstScfsi,
    IppMP3FrameHeader* pFrameHeader);
```

Parameters

ppBitStream	Pointer to the pointer to the first byte of the side information associated with the current frame in the bit stream buffer. The function updates this parameter.
pFrameHeader	Pointer to the structure that contains the unpacked MP3 frame header. The header structure provides format information about the input bitstream. Both single- and dual-channel MPEG-1 and MPEG-2 modes are supported.
pDstSideInfo	Pointer to the MP3 side information structure. The structure contains side information that applies to all granules and all channels for the current frame. One or more of the structures are placed contiguously in the buffer pointed to by *pDstSideInfo* in the following order: {granule 0 (channel 0, channel 1), granule 1 (channel 0, channel 1)}.
pDstMainDataBegin	Pointer to the `main_data_begin` field.
pDstPrivateBits	Pointer to the `private bits` field.
pDstScfsi	Pointer to the scale factor selection information associated with the current frame. The data is organized contiguously in the buffer pointed to by *pDstScfsi* in the following order: {channel 0 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3), channel 1 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3)}.

Description

The function `ippsUnpackSideInfo_MP3` is declared in the `ippac.h` file. This function unpacks the side information from the input bitstream. Before calling the function `ippsUnpackSideInfo_MP3`, make sure that the pointer `*`*ppBitStream* points to the first

byte of the bit stream that contains the side information associated with the current frame. Before returning to the caller, the function updates the pointer *ppBitStream so that it references the next byte after the side information.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pDstSideInfo*, *pDstMainDataBegin*, *pDstPrivateBits*, *pDstScfsi*, *pFrameHeader*, or *ppBitStream* is NULL.
ippStsMP3FrameHeaderErr	Indicates an error when some elements in the MP3 frame header structure are invalid: *pFrameHeader*->id != IPP_MP3_ID_MPEG1 *pFrameHeader*->id != IPP_MP3_ID_MPEG2 *pFrameHeader*->layer != 1 *pFrameHeader*->mode exceeds [0..3].
ippStsMP3SideInfoErr	Indicates an error when the value of *block_type* is zero when *window_switching_flag* is set.

UnpackScaleFactors_MP3

Unpacks scalefactors.

Syntax

```
IppStatus ippsUnpackScaleFactors_MP3_1u8s(Ipp8u** ppBitStream, int* pOffset,
    Ipp8s* pDstScaleFactor, IppMP3SideInfo* pSideInfo, int* pScfsi,
    IppMP3FrameHeader* pFrameHeader, int granule, int channel);
```

Parameters

ppBitStream	Pointer to the pointer to the first bitstream buffer byte associated with the scale factors for the current frame, granule, and channel. The function updates this parameter.
pOffSet	Pointer to the next bit in the byte referenced by *ppBitStream. Valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit. The function updates this parameter.
pSideInfo	Pointer to the MP3 side information structure associated with the current granule and channel.

pScfsi	Pointer to scale factor selection information for the current channel.
channel	Channel index. Can take the values of either 0 or 1.
granule	Granule index. Can take the values of either 0 or 1.
pFrameHeader	Pointer to MP3 frame header structure for the current frame.
pDstScaleFactor	Pointer to the scalefactor vector for long and/or short blocks.

Description

The function `ippsUnpackScaleFactors_MP3` is declared in the `ippac.h` file. This function unpacks short and/or long block scalefactors for one granule of one channel and places the results in the vector *pDstScaleFactor*. Before returning to the caller, the function updates **ppBitStream* and **pOffset* so that they point to the next available bit in the input bitstream.

NOTE. MPEG-2 intensity mode: if the intensity position is equal to the maximum value, or illegal position, the illegal position sets to negative. Consequently, in the requantization module, negative positions indicate illegal positions. Scalefactors that are not treated as intensity positions must be set to positive before using them.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the pointers *ppBitStream*, *pOffset*, *pDstScaleFactor*, *pSideInfo*, *pScfsi*, *ppBitStream*, or *pFrameHeader* is `NULL`.
`ippStsBadArgErr`	Indicates an error when *pOffset* `> 7` *pOffset* `< 0` granule and/or channel exceeds `[0, 1]`.
`ippStsMP3FrameHeaderErr`	Indicates an error when *pFrameHeader*`->id != IPP_MP3_ID_MPEG1` *pFrameHeader*`->id != IPP_MP3_ID_MPEG2` *pFrameHeader*`->`*mode* exceeds `[0..3]`.
`ippStsMP3SideInfoErr`	Indicates an error when *pSideInfo*`->`*blockType* exceeds `[0, 3]` *pSideInfo*`->`*mixedBlock* exceeds `[0, 1]`

pSideInfo->sfCompress exceeds [0, 15] or *pScfsi*[0..3] exceeds [0, 1] when *pFrameHeader* indicates the bitstream is MPEG-1

pSideInfo->sfCompress exceeds [0, 511] when *pFrameHeader* indicates the bitstream is MPEG-2.

HuffmanDecode_MP3
HuffmanDecodeSfb_MP3
HuffmanDecodeSfbMbp_MP3

Decodes Huffman data.

Syntax

```
IppStatus ippsHuffmanDecode_MP3_1u32s(Ipp8u** ppBitStream, int* pOffset,
    Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,
    IppMP3FrameHeader* pFrameHeader, int hufSize);
```

```
IppStatus ippsHuffmanDecodeSfb_MP3_1u32s(Ipp8u** ppBitStream, int* pOffset,
    Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,
    IppMP3FrameHeader* pFrameHeader, int hufSize,
    IppMP3ScaleFactorBandTableLong* pSfbTableLong);
```

```
IppStatus, ippsHuffmanDecodeSfbMbp_MP3_1u32s(Ipp8u** ppBitStream, int* pOffset,
    Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,
    IppMP3FrameHeader* pFrameHeader, int hufSize,
    IppMP3ScaleFactorBandTableLong* pSfbTableLong,
    IppMP3ScaleFactorBandTableShort pSfbTableShort,
    IppMP3MixedBlockPartitionTable pMbpTable);
```

Parameters

ppBitStream	Pointer to the pointer to the first bit stream byte that contains the Huffman code words associated with the current granule and channel. The function updates this parameter.
pOffset	Pointer to the starting bit position in the bit stream byte pointed to by **ppBitStream*. The parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit, and 7 corresponds to the least significant bit. The function updates this parameter.

pSideInfo	Pointer to MP3 structure containing side information associated with the current granule and channel.
pFrameHeader	Pointer to MP3 structure containing the header associated with the current frame.
pDstIs	Pointer to the vector of decoded Huffman symbols used to compute the quantized values of the 576 spectral coefficients associated with the current granule and channel.
pDstNonZeroBound	Pointer to the spectral region above which all coefficients are set to zero.
hufSize	The number of Huffman code bits associated with the current granule and channel.
pSfbTableLong	Pointer to the scale factor bands table for a long block.
pSfbTableShort	Pointer to scale factor band table for short block. You can use the default table from MPEG-1, MPEG-2 standards.
pMbpTable	Pointer to the mixed block partition table.

Description

The functions `ippsHuffmanDecode_MP3`, `ippsHuffmanDecodeSfb_MP3`, and `ippsHuffmanDecodeSfbMbp_MP3_1u32s` are declared in the `ippac.h` file. These functions decode Huffman symbols for the 576 spectral coefficients associated with one granule of one channel. Before returning to the caller, the function updates **ppBitStream* so that it points to the byte in the bitstream that contains the first new bit following the decoded block of Huffman codes.

The function `ippsHuffmanDecodeSfb_MP3` makes the use of a predefined scale factor bands table possible when working with a long block. Alternatively, you can use the default table from MPEG-1, or MPEG-2 standards.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, *pDstIs*, *pDstNonZeroBound*, *pSideInfo*, *pFrameHeader*, or *ppBitStream* is `NULL` or when *pOffset* `< 0` or *pOffset* `> 7`.
`ippStsMP3FrameHeaderErr`	Indicates an error when some elements in the MP3 frame header structure are invalid.

ippStsMP3SideInfoErr	Indicates an error when some elements in the MP3 side information structure are invalid or when *hufSize* < 0 or *hufSize* > *pSideInfo->part23Len*.
ippStsErr	Indicates unknown error.

ReQuantize_MP3
ReQuantizeSfb_MP3

Requantizes the decoded Huffman symbols.

Syntax

```
IppStatus ippsReQuantize_MP3_32s_I(Ipp32s* pSrcDstIsXr, int* pNonZeroBound,
    Ipp8s* pScaleFactor, IppMP3SideInfo* pSideInfo, IppMP3FrameHeader*
    pFrameHeader, Ipp32s* pBuffer);
IppStatus ippsReQuantizeSfb_MP3_32s_I(Ipp32s* pSrcDstIsXr, int* pNonZeroBound,
    Ipp8s* pScaleFactor, IppMP3SideInfo* pSideInfo, IppMP3FrameHeader*
    pFrameHeader, Ipp32s* pBuffer, IppMP3ScaleFactorBandTableLong pSfbTableLong,
    IppMP3ScaleFactorBandTableShort pSfbTableShort);
```

Parameters

pSrcDstIsXr	Pointer to the vector of the decoded Huffman symbols. For stereo and dual channel modes, right channel data begins at the address &(*pSrcDstIsXr*[576]). The function updates this vector.
pNonZeroBound	Pointer to the spectral bound above which all coefficients are set to zero. For stereo and dual channel modes, the left channel bound is pNonZeroBound [0], and the right channel bound is pNonZeroBound [1].
pScaleFactor	Pointer to the scalefactor buffer. For stereo and dual channel modes, the right channel scalefactors begin at & (*pScaleFactor* [IPP_MP3_SF_BUF_LEN]).
pSideInfo	Pointer to the side information for the current granule.
pFrameHeader	Pointer to the frame header for the current frame.
pBuffer	Pointer to the workspace buffer. The buffer length must be 576 samples.
pSfbTableLong	Pointer to the scale factor bands table for a long block.
pSfbTableShort	Pointer to the scale factor bands table for a short block.

Description

The functions `ippsReQuantize_MP3` and `ippsReQuantizeSfb_MP3` are declared in the `ippac.h` file. These functions requantize the decoded Huffman symbols. Spectral samples for the synthesis filter bank are derived from the decoded symbols using the requantization equations given in the ISO standard.

If necessary, you can apply stereophonic mid/side (M/S) and/or intensity decoding. The function returns requantized spectral samples in the vector *pSrcDstIsXr*.

For short blocks you can perform reordering operation. You should preallocate a workspace buffer pointed to by *pBuffer* prior to calling the requantization function. For short blocks, the value pointed to by *pNonZeroBound* is recalculated according to the reordered sequence.

You can use predefined scale factor bands table for a long or short block with the function `ippsReQuantizeSfb_MP3`. Alternatively, you can use the default table from MPEG-1, or MPEG-2 standards.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *pSrcDstIsXr*, *pNonZeroBound*, *pScaleFactor*, *pSideInfo*, *pFrameHeader*, or *pBuffer* is `NULL`.
`ippStsBadArgErr`	Indicates an error when *pNonZeroBound* exceeds `[0, 576]`.
`ippStsMP3FrameHeaderErr`	Indicates an error when *pFrameHeader*`->id != IPP_MP3_ID_MPEG1` *pFrameHeader*`->id != IPP_MP3_ID_MPEG2` *pFrameHeader*`->`*samplingFreq* exceeds `[0, 2]`, *pFrameHeader*`->`*mode* exceeds `[0, 3]`, *pFrameHeader*`->`*modeExt* exceeds `[0, 3]`.
`ippStsMP3SideInfoErr`	Indicates an error when the bitstream is in the stereo mode, but the block type of left is different from that of right, *pSideInfo*`[ch].blockType` exceeds `[0, 3]`, *pSideInfo*`[ch].mixedBlock` exceeds `[0, 1]`, *pSideInfo*`[ch].globGain` exceeds `[0, 255]`, *pSideInfo*`[ch].sfScale` exceeds `[0, 1]`, *pSideInfo*`[ch].preFlag` exceeds `[0, 1]`, *pSideInfo*`[ch].pSubBlkGain[w]` exceeds `[0, 7]`, where `ch` is within the range of 0 to 1, and *w* is within the range of 0 to 2.

ippStsErr	Indicates an unknown error.

MDCTInv_MP3

Performs the first stage of hybrid synthesis filter bank.

Syntax

```
IppStatus ippsMDCTInv_MP3_32s(Ipp32s* pSrcXr, Ipp32s* pDstY,  Ipp32s*
    pSrcDstOverlapAdd, int nonZeroBound,  int* pPrevNumOfImdct, int blockType,
    int mixedBlock);
```

Parameters

pSrcXr	Pointer to the vector of requantized spectral samples for the current channel and granule, represented in Q5.26 format.
pDstY	Pointer to the vector of IMDCT outputs in Q7.24 format for input to PQMF bank.
pSrcDstOverlapAdd	Pointer to the overlap-add buffer. Contains the overlapped portion of the previous granule IMDCT output in Q7.24 format. The function updates this buffer.
nonZeroBound	Limiting bound for spectral coefficients. All spectral coefficients exceeding this boundary become zero for the current granule and channel.
pPrevNumOfImdct	Pointer to the number of IMDCTs computed for the current channel of the previous granule. The function updates this parameter so that it references the number of IMDCTs for the current granule.
blockType	Block type indicator.
mixedBlock	Mixed block indicator.

Description

The function `ippsMDCTInv_MP3` is declared in the `ippac.h` file. This function performs the first stage of the hybrid synthesis filter bank. The following operations are performed:

- alias reduction
- inverse modified discrete cosine transform (IMDCT) according to block size specifiers and mixed block modes
- overlap add of IMDCT outputs

- frequency inversion prior to pseudo-quadrature mirror synthesis filter (PQMF) bank.

Since IMDCT is a lapped transform, you should preallocate a buffer referenced by *pSrcDstOverlapAdd* to maintain the IMDCT overlap-add state.

The buffer must contain 576 elements. Prior to the first call to the synthesis filter bank, all elements of the overlap-add buffer should be set to zero. In between all subsequent calls, the MP3 application must preserve the contents of the overlap-add buffer.

Upon entry to `ippsMDCTInv_MP3_32s`, the overlap-add buffer should contain the IMDCT output generated by operating on the previous granule. Upon exit from `ippsMDCTInv_MP3_32s`, the overlap-add buffer contains the overlapped portion of the output generated by operating on the current granule.

Upon return from the function, the IMDCT sub-band output samples are organized as follows:

pDstY[*j**32+subband], for *j* = 0 to 17, subband = 0 to 31.

Q5.26 designates that 5 bits before and 26 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q7.24 designates that 7 bits before and 24 bits after fixed point position are used to present a 32-bit value in the fixed point format.

NOTE. Note that the pointers *pSrcXr* and *pDstY* must reference different buffers.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsErr`	Indicates an error when one or more of the following input data errors are detected: either *blockType* exceeds `[0,3]`, or *mixedBlock* exceeds `[0,1]`, or *nonZeroBound* exceeds `[0,576]`, or **pPrevNumOfImdct* exceeds `[0,32]`.

SynthPQMF_MP3

Performs the second stage of hybrid synthesis filter bank.

Syntax

```
IppStatus ippsSynthPQMF_MP3_32s16s(Ipp32s* pSrcY, Ipp16s* pDstAudioOut, Ipp32s*
    pVBuffer, int* pVPosition, int mode);
```

Parameters

pSrcY	Pointer to the block of 32 IMDCT sub-band input samples in Q7.24 format.
pDstAudioOut	Pointer to the block of 32 reconstructed PCM output samples in 16-bit signed little-endian format. Left and right channels are interleaved according to the *mode* flag.
pVBuffer	Pointer to the input workspace buffer containing Q7.24 data. The function updates this parameter.
pVPosition	Pointer to the internal workspace index. The function updates this parameter.
mode	Flag that indicates whether or not PCM audio output channels should be interleaved. 1 indicates not interleaved, 2 indicates interleaved.

Description

The function `ippsSynthPQMF_MP3` is declared in the `ippac.h` file. This function performs the second stage of the hybrid synthesis filter bank, that is, a critically-sampled 32-channel PQMF synthesis bank that generates 32 time-domain output samples for each 32-sample input block of IMDCT outputs.

For each input block, the PQMF generates an output sequence of 16-bit signed little-endian PCM samples in the vector pointed to by *pDstAudioOut*.

If *mode* equals to 2, the left and right channel output samples are interleaved, that is, LRLRLR, so that the left channel data is organized as follows:

pDstAudioOut [2*i], *i* = 0 to 31.

If *mode* equals 1, then the left and right channel outputs are not interleaved.

Since PQMF bank contains memory, the MP3 application must maintain two state variables in between calls to the function.

The application must preallocate a workspace buffer of size 512x (Number of Channels) for the PQMF computation. This buffer is referenced by the pointer *pVBuffer* and its elements should be initialized to zero prior to the first call. During subsequent calls, the *pVBuffer* input for the current call should contain the *pVbuffer* output generated by the previous call.

In addition to *pVBuffer*, the MP3 application must also initialize to zero and thereafter preserve the value of the state variable *pVPosition*. The MP3 application should modify the values contained in *pVBuffer* or *pVPosition* only during decoder reset, and the reset values should always be zero.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when least one of the pointers *pSrcY*, *pDstAudioOut*, *pVBuffer*, or *pVPosition* is NULL.
ippStsBadArgErr	Indicates an error when at least one of the specified pointers is NULL, or the value of *mode* is less than 1 or more than 2, or when the value of *pVPosition* exceeds [0, 15].
ippStsErr	Indicates an unknown error.

SynthesisFilterInit_PQMF_MP3

Initializes PQMF MP3 synthesis specification structure.

Syntax

```
IppStatus ippsSynthesisFilterInit_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3
    **ppFilterSpec, Ipp8u *pMemSpec);
```

Parameters

ppFilterSpec	Pointer to the pointer to IppsFilterSpec_PQMF_MP3 specification structure. Only the main function uses this structure.
pMemSpec	Pointer to the area for IppsFilterSpec_PQMF_MP3 specification structure.

Description

The function `ippsSynthesisFilterInit_PQMF_MP3_32f` is declared in the `ippac.h` file. This function creates and initializes a polyphase quadrature mirror MP3 synthesis specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.

SynthesisFilterInitAlloc_PQMF_MP3

Allocates memory for PQMF MP3 synthesis specification structure and initializes it.

Syntax

```
IppStatus ippsSynthesisFilterInitAlloc_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3
    **ppFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the pointer to `IppsFilterSpec_PQMF_MP3` specification structure. Only the main function uses this structure.

Description

The function `ippsSynthesisFilterInitAlloc_PQMF_MP3_32f` is declared in the `ippac.h` file. This function allocates memory for polyphase quadrature mirror MP3 synthesis specification structure, creates, and initializes this structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.

SynthesisFilterGetSize_PQMF_MP3

Returns size of PQMF MP3 synthesis specification structure.

Syntax

```
IppStatus ippsSynthesisFilterGetSize_PQMF_MP3_32f(int *pSizeSpec);
```

Parameters

pSizeSpec	Address of the IppsFilterSpec_PQMF_MP3 specification structure size value in bytes.

Description

The function ippsSynthesisFilterGetSize_PQMF_MP3_32f is declared in the ippac.h file. This function gets size of the polyphase quadrature mirror MP3 synthesis specification structure and stores the result in *pSizeSpec*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSizeSpec* is NULL.

SynthesisFilterFree_PQMF_MP3

Frees memory allocated for PQMF MP3 synthesis specification structure.

Syntax

```
IppStatus ippsSynthesisFilterFree_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3
    *pFilterSpec);
```

Parameters

pFilterSpec	Pointer to IppsFilterSpec_PQMF_MP3 specification structure.

Description

The function `ippsSynthesisFilterFree_PQMF_MP3_32f` is declared in the `ippac.h` file. This function frees memory allocated for the polyphase quadrature mirror MP3 synthesis specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the *`pFilterSpec`* identifier is incorrect.

SynthesisFilter_PQMF_MP3

Transforms PQMF MP3-processed subband signals into time domain samples.

Syntax

```
IppStatus ippsSynthesisFilter_PQMF_MP3_32f(const Ipp32f *pSrc, Ipp32f *pDst,
    const IppsFilterSpec_PQMF_MP3 *pFilterSpec, int mode);
```

Parameters

`pSrc`	Array of pointers, holds PQMF MP3-processed subband signals.
`pDst`	Pointer to the output vector, holds time domain output samples.
`pFilterSpec`	Pointer to `IppsFilterSpec_PQMF_MP3` specification structure.
`mode`	Flag that indicates whether or not MP3 audio output channels should be interleaved: 1 - indicates no interleaving, 2 - indicates interleaving.

Description

The function `ippsSynthesisFilter_PQMF_MP3_32f` is declared in the `ippac.h` file. This function performs the polyphase quadrature mirror filter bank, that is, a critically-sampled 32-channel PQMF synthesis bank that generates 32 time-domain output samples for each 32-sample input block of IMDCT outputs. For each input block, the PQMF generates an output sequence of PCM samples in the vector pointed to by *`pDst`*. If *`mode`* equals 2, the

left and right channel output samples are interleaved, that is, `LRLRLR`, so that the left channel data is organized as follows: `pDst [2*i]`, `i` = 0 to 31. If *mode* equals 1, then the left and right channel outputs are not interleaved.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsBadArgErr`	Indicates an error when *mode* exceeds [1,2].
`ippStsContextMatchErr`	Indicates an error when the *pFilterSpec* identifier is incorrect.

Advanced Audio Coding Functions

The ISO/IEC 13818-7 MPEG-2 AAC (Advanced Audio Coding) algorithm is an efficient coding method for surround signals like 5-channel signals (left, right, center, left surround, right surround).

MPEG-2 AAC supports up to 48 main audio channels with the sampling frequency between 8kHz and 96kHz. MPEG formal tests have shown that for 5-channel audio signals, AAC satisfies the ITU-R quality requirements and provides slightly better audio quality at 320 kilobits per second (kbps) than MPEG-2 BC (Backwards Compatibility) provides at 640 kbps.

Due to its high coding efficiency, AAC is a prime candidate for any digital broadcasting system and has been selected by the DRM (Digital Radio Mondiale) system. AAC also plays a major role for the delivery of high quality music via the Internet. Moreover, with some modifications, AAC is the only high quality audio coding scheme adopted within the MPEG-4 standard, the future “global multimedia language”.

Global Macros

Table 10-4 shows the definitions of global macros.

Table 10-4 Global Macro Definitions

Global Macro Name	Definition	Notes
IPP_AAC_FRAME_LEN	1024	size of data in one frame
IPP_AAC_SF_LEN	120	scale factor buffer length
IPP_AAC_GROUP_NUM_MAX	8	maximum group number for one frame
IPP_AAC_TNS_COEF_LEN	60	TNS coefficients buffer length
IPP_AAC_TNS_FILT_MAX	8	maximum TNS filter number for one frame
IPP_AAC_PRED_SFB_MAX	41	maximum prediction scale factor bands number for one frame
IPP_AAC_ELT_NUM	16	maximum number of elements for one program

Table 10-4 Global Macro Definitions (continued)

Global Macro Name	Definition	Notes
IPP_AAC_LFE_ELT_NUM	4	maximum Low Frequency Enhance elements number for one program
IPP_AAC_DATA_ELT_NUM	8	maximum data elements number for one program
IPP_AAC_COMMENTS_LEN	256	maximum length of the comment field in bytes
IPP_AAC_PULSE_NUM_MAX	5	maximum number of the pulse escapes

Data Types and Structures

This section describes the data types and structures of the Intel® IPP advanced audio coding.

ADIF Header

```
typedef struct {
     Ipp32u ADIFId;      /* 32-bit, "ADIF" ASCII code */
     int  copyIdPres;    /* copy id flag: 0 - off, 1 - on */
     int  originalCopy;  /* copyright bit:
                                 0 - no copyright on the coded bitstream,
                                 1 - copyright protected */
     int  home;          /* original/home bit:
                                 0 - the bitstream is a copy,
                                 1 - the bitstream is the original
     int  bitstreamType; /* bitstream flag:
                                 0 - constant rate bitstream,
                                 1 - variable rate bitstream */
     int  bitRate;       /* bit rate: if 0, unknown bit rate */
     int  numPrgCfgElt;        /* number of program configure elements */
```

```
    int  pADIFBufFullness[IPP_AAC_ELT_NUM];/* buffer fullness */
    Ipp8u pCopyId[9];          /* 72-bit copy id */
} IppAACADIFHeader;
```

ADTS Frame Header

```
typedef struct {
    /* ADTS fixed header */
    int id;                    /* ID 1*/
    int layer;                 /* layer index: 0x3 - Layer I
                                                0x2 - Layer II
                                                0x1 - Layer III */
    int protectionBit;         /* CRC flag 0: CRC on, 1: CRC off */
    int profile;               /* profile: 0:MP, 1:LC, 2:SSR */
    int samplingRateIndex;     /* sampling rate index */
    int privateBit;            /* private_bit, no use  */
    int chConfig;              /* channel configure */
    int  originalCopy;         /* copyright bit:
                               0 - no copyright on the coded bitstream,
                               1 - copyright protected */
    int  home;                 /* original/home bit:
                               0 - the bitstream is a copy,
                               1 - the bitstream is the original
    int emphasis;              /* not used by ISO/IEC 13818-7, but used
                               by 14496-3 */

    /* ADTS variable header */
    int cpRightIdBit;          /* copyright id bit */
    int cpRightIdStart;        /* copyright id start */
    int frameLen;              /* frame length in bytes */
    int ADTSBufFullness;       /* buffer fullness */
    int numRawBlock;           /* number of raw data blocks in the
                               frame */
    /* ADTS CRC error check, 16bits */
```

```
    int CRCWord;              /* CRC-check word */
} IppAACADTSFrameHeader;
```

Individual Channel Side Information

```
typedef struct {
    /* unpacked from the bitstream */
    int  icsReservedBit;      /* reserved bit */
    int  winSequence;         /* window sequence flag */
    int  winShape;            /* window shape flag:
                              0 - sine window,
                              1 - KBD window */
    int  maxSfb;              /* maximum effective scale factor bands */
    int  sfGrouping;          /* scale factor grouping information */
    int  predDataPres;        /* prediction data present flag for one
                              frame:
                              0 - prediction off,
                              1 - prediction on */
    int  predReset;           /* prediction reset flag:
                              0 - reset off,
                              1 - reset on */
    int  predResetGroupNum;   /* prediction reset group number */
    Ipp8u pPredUsed[IPP_AAC_PRED_SFB_MAX+3];/* prediction flag buffer
                              for each scale factor band:
                              0 - off,
                              1 - on buffer length 44 bytes, 4-byte
                              align */
    /* decoded from the above info */
    int  numWinGrp;           /* group number */
    int  pWinGrpLen[IPP_AAC_GROUP_NUM_MAX];/* buffer for number of
                                              windows in each group */
} IppAACIcsInfo;
```

AAC Scalable Main Element Header

```
typedef struct{
     int windowSequence; /* windows is short or long type */
     int windowShape;    /* what window is used for the right hand
                         part of this analysis window */
     int maxSfb;         /* number of scale factor band transmitted */
     int sfGrouping;     /* grouping of short spectral data */
     int numWinGrp;      /* window group number */
     int pWinGrpLen[IPP_AAC_GROUP_NUM_MAX]; /* length of every group */
     int msMode;         /* MS stereo flag:
                                0 - none,
                                1 - different for every sfb,
                                2 - all */
     Ipp8u (*ppMsMask)[IPP_AAC_SF_MAX]; /* if MS's used in one sfb, when
                                           msMode ==1 */
     IppAACTnsInfo pTnsInfo[IPP_AAC_CHAN_NUM];/* TNS structure for
                                                 channels */
     IppAACLtpInfo pLtpInfo[IPP_AAC_CHAN_NUM];/* LTP structure for
                                                 channels */
}IppAACMainHeader;
```

AAC Scalable Extension Element Header

```
typedef struct{
     int msMode;    /* 0 - non; 1 - part; 2 - all */
     int maxSfb;    /* number of scale factor band for extension layer */
     Ipp8u (*ppMsMask)[IPP_AAC_SF_MAX]; /* if ms is used */
     IppAACTnsInfo pTnsInfo[IPP_AAC_CHAN_NUM]; /* TNS structure for
                                                   Stereo */
     int pDiffControlLr[IPP_AAC_CHAN_NUM][IPP_AAC_SFB_MAX]; /* FSS
                                                 information for stereo */
}IppAACExtHeader;
```

AAC Program Config Element

```
typedef struct {
```

```
int eltInsTag;              /* element instanse tag */
int profile;                /* profile index:
                               0 - main,
                               1 - Low Complexity,
                               2 - Scaleable Sampling Rate,
                               3 - reserved */
int samplingRateIndex;      /* sampling rate index */
int numFrontElt;            /* number of front elements */
int numSideElt;             /* number of side elements */
int numBackElt;             /* number of back elements */
int numLfeElt;              /* number of LFE elements  */
int numDataElt;             /* number of data elements */
int numValidCcElt;          /* number of coupling channel elements */
int monoMixdownPres;         /* mono mixdown flag:
                               0 - off,
                               1 - on */
int monoMixdownEltNum;      /* number of mono mixdown elements */
int stereoMixdownPres;      /* stereo mixdown flag:
                               0 - off,
                               1 - on */
int stereoMixdownEltNum;    /* number of stereo mixdown elements */
int matrixMixdownIdxPres;   /* matrix mixdown flag:
                               0 - off,
                               1 - on  */
int matrixMixdownIdx;       /* identifier of the surround mixdown
                              coefficient */
int pseudoSurroundEnable;   /* pseudo surround:
                               0 - off,
                               1 - on */
int pFrontIsCpe[IPP_AAC_ELT_NUM];     /* channel pair flag for
                                         front elements */
int pFrontTagSel[IPP_AAC_ELT_NUM];    /* instance tag for front
                                         elements */
```

```
   int pSideIsCpe[IPP_AAC_ELT_NUM];      /* channel pair flag for
                                            side elements */
   int pSideTagSel[IPP_AAC_ELT_NUM];     /* instance tag for side
                                            elements */
   int pBackIsCpe[IPP_AAC_ELT_NUM];      /* channel pair flag for
                                            back elements */
   int pBackTagSel[IPP_AAC_ELT_NUM];     /* instance tag for back
                                            elements */
   int pLfeTagSel[IPP_AAC_LFE_ELT_NUM];  /* channel pair flag for
                                            LFE elements */
   int pDataTagSel[IPP_AAC_DATA_ELT_NUM];  /* instance tag for data
                                              elements */
   int pCceIsIndSw[IPP_AAC_ELT_NUM];     /* independent flag for
                                            coupling */
            /* channel elements */
   int pCceTagSel[IPP_AAC_ELT_NUM];      /* instance tag for coupling
                                            channel elements */
   int numComBytes;                      /* number of comment field
                                            bytes */
   Ipp8s pComFieldData[IPP_AAC_COMMENTS_LEN]; /* comment buffer field */
}IppAACPrgCfgElt;
```

Temporal Noise Shaping (TNS) Structure for One Layer

```
typedef struct{
     int tnsDataPresent; /* Indicator of TNS data present:
                            0 - not present,
                            1 - present */
     int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX];/*TNS number filter buffer*/
     int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX]; /* TNS coef resolution
                                                    flag */
     int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX];/* TNS filter length */
     int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX];/* TNS filter order */
     int pTnsDirection[IPP_AAC_TNS_FILT_MAX];/* TNS filter direction
```

```
                                                    flag */
     int pTnsCoefCompress[IPP_AAC_GROUP_NUM_MAX]; /* The most
                    significant bit of the coefficients of the noise
                   shaping filter in window w is omitted or not */
     Ipp8s pTnsFiltCoef[IPP_AAC_TNS_COEF_LEN]; /* Coefficients of one
                        noise shaping filter applied to window w */
}IppAACTnsInfo;
```

Long Term Prediction (LTP) Structure

```
typedef struct{
     int ltpDataPresent;     /* Indicator of LTP data present:
                                      0 - not present,
                                      1 - present */
     int ltpLag;               /* the optimal delay from 0 to 2047 */
     Ipp16s ltpCoef;           /* indicate the  LTP coefficient */
     int pLtpLongUsed[IPP_AAC_SF_MAX+1]; /* if long block use ltp */
     int pLtpShortUsed[IPP_AAC_WIN_MAX]; /* if short block use ltp */
     int pLtpShortLagPresent[IPP_AAC_WIN_MAX]; /* if short lag is
                                                    transmitted */
     int pLtpShortLag[IPP_AAC_WIN_MAX];  /* relative delay for short
                                             window */
}IppAACLtpInfo;
```

Channel Pair Element

```
typedef struct {
     int commonWin;                     /* common window flag:
                                        0 - off,
                                        1 - on */
     int msMaskPres;                    /* MS stereo mask present flag */
     Ipp8u pMsMask[IPP_AAC_GROUP_NUM_MAX][IPP_AAC_SF_MAX]; /* MS stereo
                                        flag buffer for each scalefactor
                                        band */
```

```
} IppAACChanPairElt;
```

Channel Information

```
typedef struct {
     int tag;                    /* channel tag */
     int id;                     /* element id */
     int samplingRateIndex;      /* sampling rate index */
     int predSfbMax;             /* maximum prediction scale factor bands*/
     int preWinShape;            /* previous block window shape */
     int winLen;                 /* 128 - short window, 1024 - others */
     int numWin;                 /* 1 - long block, 8 - short block */
     int numSwb;                 /* decided by sampling frequency and block
                                  type */

     /* unpacking from the bitstream */
     int globGain;       /* global gain */
     int pulseDataPres;  /* pulse data present flag: 0 - off, 1 - on */
     int tnsDataPres;    /* TNS data present flag: 0 - off, 1 - on */
     int gainContrDataPres;/* gain control data present flag: 0 - off,
                                                              1 - on */
        /* icsInfo pointer */
     IppAACIcsInfo *pIcsInfo; /* pointer to IppAACIcsInfo structure */

     /* channel pair pointer */
     IppAACChanPairElt *pChanPairElt;/* pointer to IppAACChanPairElt
                                        structure */

     /* section data */
     Ipp8u pSectCb[IPP_AAC_SF_LEN];/* section code book buffer */
     Ipp8u pSectEnd[IPP_AAC_SF_LEN];/* end of scalefactor offset
                                 in each section */
     int pMaxSect[IPP_AAC_GROUP_NUM_MAX];/* maximum section number
                                 for each group */
```

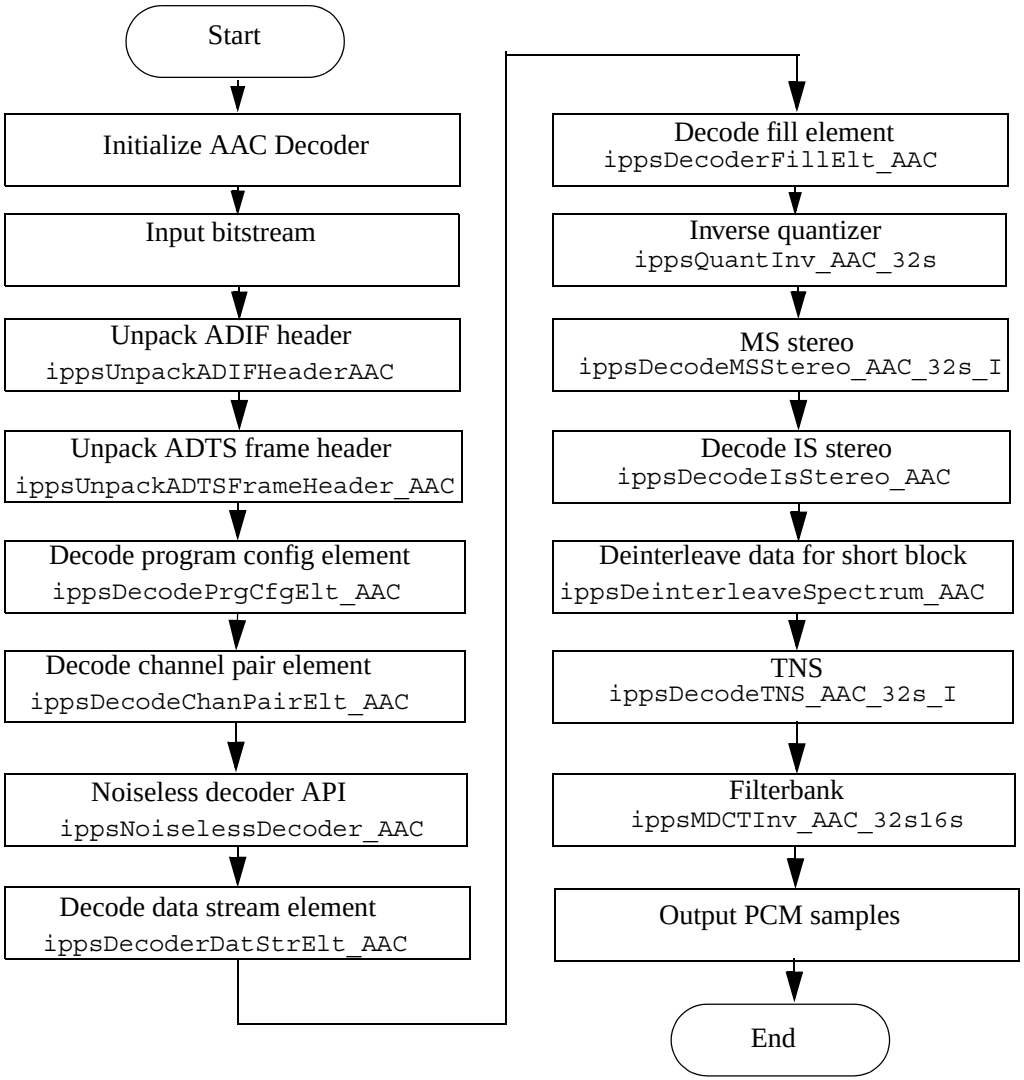
```
    int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX];/*TNS number filter
                                number buffer*/


    /* TNS data */
    int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX];/* TNS coefficients
                                resolution flag */
    int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX];/* TNS filter length */
    int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX];/* TNS filter order */
    int pTnsDirection[IPP_AAC_TNS_FILT_MAX];/* TNS filter direction
                                flag */
}IppAACChanInfo;
```

AAC Decoding Primitives

The AAC decoder API provides a variety of AAC LC decoder functions, including bitstream unpacking and AAC core decoding functions. This provides customers great flexibility in configuring the decoder system. See ISO/IEC 13818-7:1997.

Figure 10-3 shows AAC decoding flowchart.

Figure 10-3 AAC Decoding Flowchart



MPEG-2 AAC Primitives

In the following sections, the parameter *maxSfb* designates the number of scale factor bands transmitted per group and unpacked from the bitstream. The parameter *numSwb* stands for the number of scale factor window bands for the short block or the number of scale factor window bands for the long block. The values are computed according to the sampling rate and the block type.

See clause 8.3.1 of ISO/IEC 13818-7:1997.

UnpackADIFHeader_AAC

Gets the AAC ADIF format header.

Syntax

```
IppStatus ippsUnpackADIFHeader_AAC(Ipp8u** ppBitStream, IppAACADIFHeader*
    pADIFHeader, IppAACPrgCfgElt* pPrgCfgElt, int prgCfgEltMax);
```

Parameters

ppBitStream	Double pointer to the current byte before the ADIF header.
pADIFHeader	Pointer to the `IppACCADIFHeader` structure.
pPrgCfgElt	Pointer to the `IppAACPrgCfgElt` structure. There must be `prgCfgEltMax` elements in the buffer.
prgCfgEltMax	Maximum program configure element number. Must be within the range of `[1, 16]`.

Description

This function is declared in the `ippac.h` file. The function gets the AAC ADIF format header, including program configuration elements from the input bitstream. See Table 6.2, and 6.21. of ISO/IEC 13818-7:1997.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pADIFHeader*, *pPrgCfgElt*, or **ppBitStream* is `NULL`.

ippStsAacPrgNumErr	Indicates an error when the decoded *pADIFHeader->numPrgCfgElt > prgCfgEltMax*, or *prgCfgEltMax* is outside the range of [1,IPP_AAC_MAX_ELT_NUM].

NOTE. pADIFHeader->numPrgCfgElt is the number directly unpacked from bitstream plus 1.

UnpackADTSFrameHeader_AAC

Gets ADTS frame header from the input bitstream.

Syntax

```
IppStatus ippsUnpackADTSFrameHeader_AAC(Ipp8u** ppBitStream,
    IppAACADTSFrameHeader* pADTSFrameHeader);
```

Parameters

ppBitStream	Double pointer to the current byte after unpacking the ADTS frame header.
pADTSFrameHeader	Pointer to the IppAACADTSFrameHeader structure.

Description

The function gets ADTS frame header from the input bitstream. If the CRC word is applied, the first byte of the 16-bit CRC word is stored in pADTSFrameHeader->CRCWord[15:8] and the second byte is stored in pADTSFrameHeader->CRCWord[7:0]. It does not check whether the header is corrupt.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pADTSFrameHeader* or **ppBitStream* is NULL.

DecodePrgCfgElt_AAC

Gets program configuration element from the input bitstream.

Syntax

```
IppStatus ippsDecodePrgCfgElt_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACPrgCfgElt* pPrgCfgElt);
```

Parameters

ppBitStream	Double pointer to the current byte after decoding the program configuration element.
pOffset	Pointer to the bit position in the byte pointed to by `*`*ppBitStream*. Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
pPrgCfgElt	Pointer to `IppAACPrgCfgElt` structure.

Description

This function is declared in the `ippac.h` file. The function gets the program configuration element from the input bitstream. See clause 8.5 and Table 6.21 of ISO/IEC 13818-7.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, *pPrgCfgElt*, or `*`*ppBitStream* is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when *pOffset* is out of the range of `[0,7]`.

DecodeChanPairElt_AAC

Gets `channel_pair_element` *from the input bitstream.*

Syntax

```
IppStatus ippsDecodeChanPairElt_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACIcsInfo* pIcsInfo, IppAACChanPairElt* pChanPairElt, int predSfbMax);
```

Parameters

ppBitStream	Double pointer to the current byte after decoding the channel pair element.
pOffset	Pointer to the bit position in the byte pointed to by `*`*ppBitStream*. Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
pIcsInfo	Pointer to `IppAACIcsInfo` structure. If `pIcsInfo->predDataPres` = 0, set *pIcsInfo->predReset* = 0. Only the first *pIcsInfo->numWinGrp* elements in *pIcsInfo->pWinGrpLen* are meaningful. You should not change some members of the structure, as shown in Table 10-5.
pChanPairElt	Pointer to `IppAACChanPairElt` structure. You should not change some members of the structure, as shown in Table 10-5.
predSfbMax	Maximum prediction scale factor bands. For LC profile, set *predSfbMax* = 0.

Description

This function is declared in the `ippac.h` file. The function gets the channel pair element from the input bitstream. Individual channel stream is not included.

If *common_window* flag decoded from the input bitstream is 0, all members of *pIcsInfo* and *pChanPairElt* remain unchanged except for *pChanPairElt->commonWin*.

See clause 8.3 and Table 6.10, 6.11 of ISO/IEC 13818-7:1997.

Table 10-5 Unchanged Members of *pIcsInfo*

Unchanged Members	Conditions
sfGrouping	*pIcsInfo->winSequence* != 2
predResetGroupNum	*pIcsInfo->predDataPres* == 0 || *pIcsInfo->predReset* == 0
pPredUsed[sfb]	*pIcsInfo->predDataPres* == 0
pMsUsed[sfb]	*pChanPairElt*->msMaskPres != 1

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, **ppBitStream*, *pIcsInfo*, or *pChanPairElt* is NULL.
ippStsAacBitOffsetErr	Indicates an error when *pOffset* is out of the range of [0,7].
ippStsAacMaxSfbErr	Indicates an error when *pIcsInfo->maxSfb* decoded from bitstream is greater than IPP_AAC_MAX_SFB, the maximum scale factor band for all sampling frequencies.
ippStsAacPredSfbErr	Indicates an error when *predSfbMax* is out of the range of [0, IPP_AAC_PRED_SFB_MAX].

NoiselessDecoder_LC_AAC

Decodes all data for one channel.

Syntax

```
IppStatus ippsNoiselessDecoder_LC_AAC(Ipp8u** ppBitStream, int* pOffset, int
    commonWin, IppAACChanInfo* pChanInfo, Ipp16s* pDstScalefactor, Ipp32s*
    pDstQuantizedSpectralCoef, Ipp8u* pDstSfbCb, Ipp8s* pDstTnsFiltCoef);
```

Parameters

ppBitStream	Double pointer to the current byte.

pOffset	Pointer to the offset in one byte.
commonWin	Common window indicator.
pChanInfo	Pointer to the channel information. `IppAACChanInfo` structure. Denotes *pIcsInfo* as *pChanInfo*->`pIcsInfo` as shown in Table 10-6.
pDstScalefactor	Pointer to the scalefactor or intensity position buffer. Buffer length is more than or equal to 120. Only *maxSfb* elements are stored for each group. There is no space between sequence groups.
pDstQuantizedSpectralCoef	Pointer to the quantized spectral coefficients data. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length is more than or equal to 1024.
pDstSfbCb	Pointer to the scale factor band codebook. Buffer length must be more than or equal to 120. Store *maxSfb* elements for each group. There is no space between the sequence groups.
pDstTnsFiltCoef	Pointer to TNS coefficients. Buffer length must be more than or equal to 60. The store sequence is TNS order elements for each filter for each window. The elements are not changed if the corresponding TNS order is zero.

Description

This function is declared in the `ippac.h` file. The function decodes all data for one channel, including scale factor/intensity positions, spectral coefficients, TNS coefficients, and associated side information for LC profile.

You need to set *pChanInfo*->`pIcsInfo`, *pChanInfo->samplingRateIndex*, *pChanInfo*->`predSfbMax` to correct pointer/values before calling this function.

Table 10-6 Input/Output Members List of *pChanInfo*

Member	Output
Tag	Not used.
id	Not used.
preWinShape	Not used.
pChanPairElt	Not used.
samplingRateIndex	As input. Not changed.
predSfbMax	As input. Must be 0. Not changed.

Table 10-6 Input/Output Members List of *pChanInfo* (continued)

Member	Output
winLen	As output. Set to 128, if decoded, *pIcsInfo->winSequence* is short block. Otherwise set to 1024.
numWin	As output. Set to 8, if decoded pIcsInfo*->winSequence* is short block. Otherwise set to 1.
numSwb	As output. Set to the maximum number of scale factor window bands in each group according to *samplingRateIndex* and *pIcsInfo->winSequence*. See Table 8.4-8.1 of ISO/IEC 13818-7:1997.
globGain *pulseDataPres* *tnsDataPres* *gainContrDataPres*	As output. Unpacked from bitstream.
pMaxSect	As output. Pointer to the maximum of sections number in each group. Only *pIcsInfo->numWinGrp* elements in the buffer are meaningful.
pSectCb	As output. Pointer to the section codebook. Only *pMaxSect*[g] elements are stored for each group. There is no space between the sequence groups.
pTnsRegionLen	As output. Pointer to the length of the region in units of scale factor bands to which one filter is applied in each window.
pTnsFiltOrder	As output. Pointer to the order of the temporal noise shaping filter applied to each window.
pTnsDirection	As output. Pointer to the token that indicates whether the filter is applied in the upward or downward direction. 0 stands for upward and 1 for downward.
pIcsInfo	As input if *commonWin* == 1. As output if *commonWin* == 0. If *pIcsInfo->predDataPres* == 0, set *pIcsInfo->predReset* = 0. Only the first *pIcsInfo->numWinGrp* elements in *pIcsInfo->pWinGrpLen* are meaningful. Under specific conditions, some members of the structure must remain unchanged. See Table 10-5.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, *pChanInfo*, *pDstScalefactor*, *pDstQuantizedSpectralCoef*, *pDstSfbCb*, *pDstTnsFiltCoef*, *pChanInfo->pIcsInfo*, or **ppBitStream* is NULL.
ippStsAacBitOffsetErr	Indicates an error when *pOffset* is out of range [0,7].
ippStsAacComWinErr	Indicates an error when *commonWin* exceeds [0,1].
ippStsAacSmplRateIdxErr	Indicates an error when *pChanInfo->samplingRateIndex* exceeds [0,11].
ippStsAacPredSfbErr	Indicates an error when *pChanInfo->predSfbMax* is not equal to 0.
ippStsAacMaxSfbErr	Indicates an error when *pChanInfo->pIcsInfo->maxSfb > pChanInfo->numSwb*.
ippStsAacSectCbErr	Indicates an error when the codebook pointed to by *pChanInfo->pSectCb* is illegal or when (*pChanInfo->pSectCb*)==12, 13. If the current channel is not the right channel of the channel pair element, *pSectCb* = 14, 15 is also illegal.
ippStsAacPlsDataErr	Indicates an error when the *pChanInfo->pIcsInfo->winSequence* indicates a short sequence and *pChanInfo->pulsePres* indicates pulse data present. The start scale factor band for pulse data >= *pChanInfo->numSwb*, or pulse data position *offset* >= *winLen*.
ippStsAacGainCtrErr	Indicates an error when *pChanInfo->gainControlPres* is decoded as 1, which means that gain control data is present. Gain control data is not currently supported.
ippStsAacCoefValErr	Indicates an error when the quantized coefficients value pointed to by *pDstCoef* exceeds range [-8191, 8191].

DecodeDatStrElt_AAC

Gets data stream element from the input bitstream.

Syntax

```
IppStatus ippsDecodeDatStrElt_AAC(Ipp8u** ppBitStream, int* pOffset, int*
    pDataTag, int* pDataCnt, Ipp8u* pDstDataElt);
```

Parameters

ppBitStream	Double pointer to the current byte after the decode data stream element.
pOffset	Pointer to the bit position in the byte pointed to by **ppBitStream*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
pDataTag	Pointer to *element_instance_tag*. See Table 6.20 of ISO/IEC 13818-7:1997.
pDataCn	Pointer to the value of data length in bytes.
pDstDataElt	Pointer to the data stream buffer that contains the data stream extracted from the input bitstream. There are 512 elements in the buffer pointed to by *pDstDataElt*.

Description

This function is declared in the `ippac.h` file. The function gets data stream element from the input bitstream.

See clause 8.6 and Table 6.20 of ISO/IEC 13818-7:1997.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, **ppBitStream*, *pDataTag*, *pDataCnt*, or *pDstDataElt* is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when **pOffset* is out of range `[0,7]`.

DecodeFillElt_AAC

Gets the fill element from the input bitstream.

Syntax

```
IppStatus ippsDecodeFillElt_AAC(Ipp8u** ppBitStream, int* pOffset, int* pFillCnt,
    Ipp8u* pDstFillElt);
```

Parameters

ppBitStream	Pointer to the pointer to the current byte after the decode fill element.
pOffset	Pointer to the bit position in the byte pointed to by `*ppBitStream`. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
pFillCnt	Pointer to the value of the length of total fill data in bytes.
pDstFillElt	Pointer to the fill data buffer whose length must be equal to or greater than 270.

Description

This function is declared in the `ippac.h` file. The function gets the fill element from the input bitstream.

See clause 8.7 and Table 6.22 of ISO/IEC 13818-7:1997.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, *pFillCnt*, or *pDstFillElt* is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when `*pOffset` is out of the range of `[0,7]`.

QuantInv_AAC

Performs inverse quantization of Huffman symbols for current channel in-place.

Syntax

```
IppStatus ippsQuantInv_AAC_32s_I(Ipp32s* pSrcDstSpectralCoef, const Ipp16s*
    pScalefactor, int numWinGrp, const int* pWinGrpLen, int maxSfb, const Ipp8u*
    pSfbCb, int samplingRateIndex, int winLen);
```

Parameters

pSrcDstSpectralCoef	On input, pointer to the input quantized coefficients. For short block the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. On output, pointer to the destination inverse quantized coefficient in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. The maximum error of output *pSrcDstSpectralCoef*[*i*] is listed in Table 10-7.
pScalefactor	Pointer to the scalefactor buffer. Buffer length must be more than or equal to 120.
numWinGrp	Group number.
pWinGrpLen	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
maxSfb	Maximal scale factor bands number for the current block.
pSfbCb	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Only *maxSfb* elements for each group are meaningful. There are no spaces between the sequence groups.
samplingRateIndex	Sampling rate index. Valid within [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997.
winLen	Data number in one window.

Description

This function is declared in the `ippac.h` file. The function performs inverse quantization of Huffman symbols for the current channel as shown by the formula.

$$pSrcDst[i] = sign(pSrcDst[i])*(pSrcDst[i])^{\frac{4}{3}} *2^{\left(\frac{1}{2}(pScalefactor[sfb]-100)\right)}$$

See clause 10 of ISO/IEC 13818-7:1997.

Table 10-7 Computation Error List for *pSrcDstSpectralCoef*

Output	Conditions	
max(error (*pSrcDstSpectralCoef* [*i*]))	Input abs (*pSrcDstSpectralCoef* [*i*])	Output abs (*pSrcDstSpectralCoef*[*i*])
3	<= 128	< 2 ^ 29
3	129~8191	<= 2 ^ 25
7	129~8191	< 2 ^ 29

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacSmplRateIdxErr`	Indicates an error when *samplingRateIndex* exceeds `[0, 11]`.
`ippStsAacMaxSfbErr`	Indicates an error when *maxSfb* exceeds `[0,IPP_AAC_MAX_SFB]`.
`ippStsAacWinGrpErr`	Indicates an error when *numWinGrp* exceeds `[0, 8]` for long window or is not equal to 1 for short window.
`ippStsAacWinLenErr`	Indicates an error when *winLen* is not equal to 128 or 1024;
`ippStsAacCoefValErr`	Indicates an error when the quantized coefficients value pointed to by *pSrcDstSpectralCoef* exceeds `[-8191,8191]`.

DecodeMsStereo_AAC

Processes mid-side (MS) stereo for pair channels in-place.

Syntax

```
IppStatus ippsDecodeMsStereo_AAC_32s_I(Ipp32s* pSrcDstL, Ipp32s* pSrcDstR, int
    msMaskPres, const Ipp8u* pMsUsed, Ipp8u* pSfbCb, int numWinGrp, const int*
    pWinGrpLen, int maxSfb, int samplingRateIndex, int winLen);
```

Parameters

pSrcDstL	On input, pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. On output, pointer to left channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
pSrcDstR	On input, pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. On output, pointer to right channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
msMaskPres	MS stereo mask flag: 0 - MS off 1 - MS on 2 - MS all bands on.
pMsUsed	Pointer to the MS Stereo flag buffer. Buffer length must be more than or equal to 120.
pSfbCb	Pointer to the scale factor band codebook. If `invert_intensity (group, sfb) = -1`, and if `*pSfbCb` = `INTERITY_HCB`, let `*pSfbCb` = `INTERITY_HCB2`. If `*pSfbCb` = `INTERITY_HCB2`, let `*pSfbCb` = `INTERITY_HCB`.

	Buffer length must be more than or equal to 120. Store *maxSfb* elements for each group. There is no space between the sequence groups.
numWinGrp	Group number.
pWinGrpLen	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
maxSfb	Maximal scale factor bands number for the current block.
samplingRateIndex	Sampling rate index. Valid within `[0, 11]`. See Table 6.5 of ISO/IEC 13818-7:1997.
winLen	Data number in one window.

Description

This function is declared in the `ippac.h` file. The function performs mid-side (MS) stereo process for pair channels and at the same time runs the `invert_intensity(group, sfb)` function and stores the values in the *pSfbCb* buffer.

In the case when MS stereo flag is on, perform operation on *pSrcDstL*`[i]` and *pSrcDstR*`[i]` as described in the following formula:

$$pSrcDstL'[i] = pSrcDstL[i] + pSrcDstR[i]$$

$$pSrcDstR'[i] = pSrcDstL[i] - pSrcDstR[i].$$

See clause 12 of ISO/IEC 13818-7:1997.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacMaxSfbErr`	Indicates an error when the coefficient index calculated from *samplingFreqIndex* and *maxSfb* exceeds *winLen* in each window.
`ippStsAacSamplRateIdxErr`	Indicates an error when *pChanInfor->samplingRateIndex* exceeds `[0,11]`.
`ippStsAacWinGrpErr`	Indicates an error when *numWinGrp* exceeds `[0,8]` for long window or is not equal to 1 for short window.
`ippStsAacWinLenErr`	Indicates an error when *winLen* is not equal 128 or 1024.
`ippStsStereoMaskErr`	Indicates an error when the stereo mask flag is not equal 1 or 2.

DecodeIsStereo_AAC

Processes intensity stereo for pair channels.

Syntax

```
IppStatus ippsDecodeIsStereo_AAC_32s(const Ipp32s* pSrcL, Ipp32s* pDstR,
    const Ipp16s* pScalefactor, const Ipp8u* pSfbCb, int numWinGrp,
    const int* pWinGrpLen, int maxSfb, int samplingRateIndex, int winLen);
```

Parameters

pSrcL	Pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
pScalefactor	Pointer to the scalefactor buffer. Buffer length must be more than or equal to 120.
pSfbCb	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Store *maxSfb* elements for each group.There is no space between the sequence groups. Respective values of *pSfbCb*[sfb] equal to 1, -1, or 0 indicate the intensity stereo mode, that is, direct, inverse, or none.
numWinGrp	Group number.
pWinGrpLen	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
maxSfb	Maximal scalefactor bands number for the current block.
samplingRateIndex	Sampling rate index. Valid within [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997.
winLen	Data number in one window.
pDstR	Pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.

Description

This function is declared in the ippac.h file. The function processes intensity stereo for pair channels.

You should perform operation on *pSrcL*[*i*], *pDstR*[*i*] according to the following formula.

$$pDstR[i] = pSrc[i]*is_intensity(g, sfb)*2^{\left(-\frac{1}{4}pScalefactor[sfb]\right)}$$

NOTE. `invert_intensity(g, sfb)` is not used in the formula, because it is already decoded and stored in `pSfbCb[sfb]` in the MS stereo process primitive. Refer to clause 12 of ISO/IEC 13818-7:1997.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacMaxSfbErr`	Indicates an error when the coefficient index calculated from *samplingFreqIndex* and *maxSfb* exceeds *winLen* in each window.
`ippStsAacSamplRateIdxErr`	Indicates an error when *pChanInfor->samplingRateIndex* exceeds `[0,11]`.
`ippStsAacWinGrpErr`	Indicates an error when *numWinGrp* exceeds `[0,8]` for long window or is not equal to 1 for short window.
`ippStsAacWinLenErr`	Indicates an error when *winLen* is not equal to 128 or 1024.

DeinterleaveSpectrum_AAC

Deinterleaves the coefficients for short block.

Syntax

```
IppStatus ippsDeinterleaveSpectrum_AAC_32s(const Ipp32s* pSrc, Ipp32s* pDst,
    int numWinGrp, const int* pWinGrpLen, int maxSfb, int samplingRateIndex,
    int winLen);
```

Parameters

pSrc	Pointer to source coefficients buffer. The coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
numWinGrp	Group number.

pWinGrpLen	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
maxSfb	Maximal scalefactor bands number for the current block.
samplingRateIndex	Sampling rate index. Valid in `[0, 11]`. See Table 6.5 of ISO/IEC 13818-7:1997.
winLen	Data number in one window.
pDst	Pointer to the output of coefficients. Data sequence is ordered in *pDst*`[`*w*`*128+`*sfb*`*`*sfbWidth*`[`*sfb*`]+`*i*`]`, where *w* is window index, *sfb* is scale factor band index, *sfbWidth* is the scale factor band width table, *i* is the index within scale factor band. Buffer length must be more than or equal to 1024.

Description

This function is declared in the `ippac.h` file. The function deinterleaves the coefficients for short block.

See clause 8.3.5 of ISO/IEC 13818-7:1997.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacMaxSfbErr`	Indicates an error when the coefficient index calculated from *samplingFreqIndex* and *maxSfb* exceeds *winLen* in each window.
`ippStsAacSamplRateIdxErr`	Indicates an error when *pChanInfor->samplingRateIndex* exceeds `[0,11]`.
`ippStsAacWinGrpErr`	Indicates an error when *numWinGrp* exceeds `[0,8]`.
`ippStsAacWinLenErr`	Indicates an error when *winLen* is not equal to 128.

DecodeTNS_AAC

Decodes for Temporal Noise Shaping in-place.

Syntax

```
IppStatus ippsDecodeTNS_AAC_32s_I(Ipp32s* pSrcDstSpectralCoefs, const int*
    pTnsNumFilt, const int* pTnsRegionLen, const int* pTnsFiltOrder, const int*
    pTnsFiltCoefRes, const Ipp8s* pTnsFiltCoef, const int* pTnsDirection,
    int maxSfb, int profile, int samplingRateIndex, int winLen);
```

Parameters

pSrcDstSpectralCoefs On input, pointer to the input spectral coefficients to be filtered by the all-pole filters in Q13.18 format. There are 1024 elements in the buffer pointed to by *pSrcDstSpectralCoefs*.

On output, pointer to the output spectral coefficients after filtering by the all-pole filters in Q13.18 format.

See Table 10-8 for the computation error compared with the double precision data.

pTnsNumFilt Pointer to the number of noise shaping filters used for each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsNumFilt* which are arranged as follows:*pTnsNumFilt*[*w*]: the number of noise shaping filters used for window *w*, *w* = 0 to *numWin*-1.

pTnsRegionLen Pointer to the length of the region in units of scale factor bands to which one filter is applied in each window of the current frame.

There are 8 elements in the buffer pointed to by *pTnsRegionLen*, which are arranged as follows:

pTnsRegionLen[*i*]: the length of the region to which filter filt is applied in window *w*

$$i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt$$

w = 0 to *numWin*-1, *filt*=0 to *pTnsNumFilt*[*w*]-1.

pTnsFiltOrder Pointer to the order of one noise shaping filter applied to each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsFiltOrder*, which are arranged as follows:

pTnsFiltOrder[i]: the order of one noise shaping filter filt, which is applied to window *w*

$$i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt$$

w = 0 to *numWin*-1, *filt*=0 to *pTnsNumFilt*[*w*]-1.

pTnsFiltCoefRes Pointer to the resolution of 3 bits or 4 bits of the transmitted filter coefficients for each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsFiltCoefRes*, which are arranged as follows:

pTnsFiltCoefRes[w]: the resolution of the transmitted filter coefficients for window *w*, *w* = 0 to *numWin*-1.

pTnsFiltCoef Pointer to the coefficients of one noise shaping filter applied to each window of the current frame. There are 60 elements in the buffer pointed to by *pTnsFiltCoef*, which are arranged as follows:

pTnsFiltCoef[*i*], *pTnsFiltCoef*[*i*+1], …, *pTnsFiltCoef*[i+order-1]: the coefficients of one noise shaping filter *filt*, which is applied to window w.

The order is the same as that of the noise shaping filter *filt* as applied to window w, w = 0 to numWin-1, filt=0 to *pTnsNumFilt*[w]-1.

For example, *pTnsFiltCoef*[0], *pTnsFiltCoef*[1], …, *pTnsFiltCoef*[order0-1] are the coefficients of the noise shaping filter 0, which is applied to window 0, if present.

If so, *pTnsFiltCoef*[*order0*], *pTnsFiltCoef*[*order0*+1], …, *pTnsFiltCoef*[order0+order1-1] are the coefficients of the noise shaping filter 1 applied to window 0, if present, and so on.

order0 is the same as that of the noise shaping filter 0 applied to window 0, and order1 is the order of the noise shaping filter 1 applied to window 0.

After window 0 is processed, process window 1, then window 2 until all *numWin* windows are processed.

pTnsDirection Pointer to the token that indicates whether the filter is applied in the upward or downward direction.

0 stands for upward and 1 for downward.

There are 8 elements in the buffer pointed to by *pTnsDirection* which are arranged as follows:

pTnsDirection[i]: the token indicating whether the filter filt is applied in upward or downward direction to window *w*

$$i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt$$

w = 0 to $numWin$-1, $filt$=0 to $pTnsNumFilt[w]$-1.

maxSfb	Number of scale factor bands transmitted per window group of the current frame.
profile	Profile index from Table 7.1 in ISO/IEC 13818-7:1997.
samplingRateIndex	Index indicating the sampling rate of the current frame.
winLen	Data number in one window.

Description

This function is declared in the `ippac.h` file. The function performs decoding process for Temporal Noise Shaping (TNS) that controls the temporal shape of the quantization noise within each window of the transform.

The TNS decoding process proceeds separately for each window of the current frame by applying the all-pole filtering to selected regions of the spectral coefficients.

Table 10-8 Computation Error List for *pSrcDstSpectralCoefs*

MAX(error(*pSrcDstSpectralCoefs*[*i*]))	**Condition**
4095	8 == numWin
32767	1 == numWin

numWin is the number of windows in a window sequence of the current frame. *numWin* is equal to 8 if window sequence is `EIGHT_SHORT_SEQUENCE`, or to 1 for other window sequences.

numSwb is the total number of scale factor window bands for the actual window type, that is, long or short window of the current frame.

NOTE. This function supports LC profile only.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
IppStsTnsProfielErr	Indicates an error when *profile* != 1.
ippStsAacTnsNumFiltErr	Indicates an error when a data error occurs: for a short window sequence *pTnsNumFilt*[*w*] exceeds [0,1]; for a long window sequence, *pTnsNumFilt*[*w*] exceeds [0,3].
ippStsAacTnsLenErr	Indicates an error when *pTnsRegionLen* exceeds [0, *numSwb*].
ippStsAacTnsOrderErr	Indicates an error when a data error occurs: for a short window sequence, *pTnsFiltOrder* exceeds [0,7]; for a long window sequence, *pTnsFiltOrder* exceeds [0,12].
ippStsAacTnsCoefResErr	Indicates an error when *pTnsFiltCoefRes*[w] exceeds [3, 4].
ippStsAacTnsCoefErr	Indicates an error when **pTnsFiltCoef* exceeds [-8, 7].
ippStsAacTnsDirectErr	Indicates an error when **pTnsDirection* exceeds [0, 1].

NOTE. *numWin* is the number of windows in a window sequence of the current frame. *numWin* is 8 if window sequence is EIGHT_SHORT_SEQUENCE, or 1 for other window sequences.

NOTE. *numSwb* is the total number of scale factor window bands for the actual window type, that is, long or short window, of the current frame.

MDCTInv_AAC_32s16s

Maps time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples.

Syntax

```
IppStatus ippsMDCTInv_AAC_32s16s(Ipp32s* pSrcSpectralCoefs, Ipp16s*
    pDstPcmAudioOut, Ipp32s* pSrcDstOverlapAddBuf, int winSequence, int
    winShape, int prevWinShape, int pcmMode);
```

Parameters

pSrcSpectralCoefs	Pointer to the input time-frequency domain samples in Q13.18 format. There are 1024 elements in the buffer pointed to by *pSrcSpectralCoefs*.
pSrcDstOverlapAddBuf	Pointer to the overlap-add buffer that contains the second half of the previous block windowed sequence in Q13.18. There are 1024 elements in this buffer.
winSequence	Flag that indicates which window sequence is used for current block.
winShape	Flag that indicates which window function is selected for current block.
prevWinShape	Flag that indicates which window function is selected for previous block.
pcmMode	Flag that indicates whether the PCM audio output is interleaved, that is has the pattern LRLRLR… or not. 1 stands for not interleaved. 2 stands for interleaved
pDstPcmAudioOut	Pointer to the output 1024 reconstructed 16-bit signed little-endian PCM samples in Q15, interleaved, if needed. The maximum computation error for *pDstPcmAudioOut* is less than 1 for each vector element. The total quadratic error for the vector is less than 96.
pSrcDstOverlapAddBuf	Pointer to the overlap-add buffer which contains the second half of the current block windowed sequence in Q13.18. The maximum computation error for *pDstPcmAudioOut* is less than 4 for each vector element. The total quadratic error for the vector is less than 1536.

Description

This function is declared in the `ippac.h` file. This function maps the time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples as output for each channel.

This module consists of

- IMDCT transform
- windowing
- overlap-add operation.

In order to adapt the time/frequency resolution of the filterbank to the characteristics of the input signal, a block switching tool is also adopted. For each channel, 1024 time-frequency domain samples are transformed into the time domain via the IMDCT.

After applying the windowing operation, the first half of the windowed sequence is added to the second half of the previous block windowed sequence to reconstruct 1024 output samples for each channel. Output can be interleaved according to *pcmMode*.

If *pcmMode* equals to 2, the output is in the sequence *pDstPcmAudioOut*`[2*i], i=0` to 1023, that is, 1024 output samples are stored in the sequence: *pDstPcmAudioOut*`[0]`, *pDstPcmAudioOut*`[2]`, *pDstPcmAudioOut*`[4]`,..., *pDstPcmAudioOut*`[2046]`.

If *pcmMode* equals 1, the output is in the sequence *pDstPcmAudioOut*`[i], i=0` to 1023.

You should also preallocate an input-output buffer pointed to by *pSrcDstOverlapAddBuf* for the overlap-add operation.

Reset this buffer to zero before the first call and then use the output of the current call as the input of the next call for the same channel.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacWinSeqErr`	Indicates an error when *winSequence* exceeds `[0,3]`.
`ippStsAacWinShapeErr`	Indicates an error when *winShape* or *prevWinShape* exceeds `[0,1]`.
`ippStsAacPcmModeErr`	Indicates an error when *pcmMode* exceeds `[1,2]`.

MDCTInv_AAC_32s_I

Computes inverse modified discrete cosine transform (MDCT), windowing and overlapping of signals.

Syntax

```
IppStatus ippsMDCTInv_AAC_32s_I(Ipp32s* pSrcDst, Ipp32s *pSrcDstOverlapBuf, int
    winSequence, int winShape, int prevWinShape, int len);
```

Parameters

pSrcDst	Pointer to the source and destination vector for the in-place operation.
pSrcDstOverlapBuf	Pointer to overlap-add buffer, contains the output of the 2nd half of the previous frame windowed sequence.
winSequence	Window sequence indicator.
winShape	Window shape indicator.
prevWinShape	Window shape indicator of the previous frame.
len	Number of samples in Src buffer.

Description

The `ippsMDCTInv_AAC_32s_I` function is declared in the `ippac.h` file. This function performs inverse MDCT operation. Then, after applying the windowing operation, the first half of the windowed sequence is added to the second half of the previous frame windowed sequence (contents of *pSrcDstOverlapBuf*) to reconstruct 1024 output samples. The second half of the windowed sequence is stored to *pSrcDstOverlapBuf*.

This function is implemented for using in AAC decoder. It does not perform any saturation so the user should consider overflow possibility. The source and destination have different positions of the decimal point (Q format). The Q format of destination depends on the window sequence indicator.

WinSequence = 0 (long window): If *Src* is in Qn format then *Dst* in Q(n+5) format

WinSequence = 1 (long start window): If *Src* is in Qn format then *Dst* in Q(n+5) format

WinSequence = 2 (eight short window): If *Src* is in Qn format then *Dst* in Q(n+2) format

WinSequence = 3 (long stop window): If *Src* is in Qn format then *Dst* in Q(n+5) format.

For example, to have *Dst* in Q7 format, the *Src* format should be Q5 in the case of eight short window and in Q2 in other cases.

Only *len* = 1024 is supported in the current implementation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the pointers passed to the function is NULL.
ippStsAacWinSeqErr	Indicates an error when *winSequence* exceeds [0,3].
ippStsAacWinShapeErr	Indicates an error when *winShape* or *prevWinShape* exceeds [0,1].
ippStsSizeErr	Indicates an error when *len* is not equal to 1024.

MPEG-4 AAC Primitives

This section introduces primitive functions for MPEG-4 AAC operations.

DecodeMainHeader_AAC

Gets main header information and main layer information from bit stream.

Syntax

```
IppStatus ippsDecodeMainHeader_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACMainHeader* pAACMainHeader, int channelNum, int monoStereoFlag);
```

Parameters

ppBitStream	Double pointer to bitstream buffer. **ppBitStream* will be updated after decoding.
pOffset	Pointer to the offset in one byte. **pOffset* will be updated after decoding.
channelNum	Number of channels.
monoStereoFlag	Current frame has mono and stereo layers.

pAACMainHeader	Pointer to the main element header.

Description

This function is declared in the `ippac.h` file. The function gets main header information and main layer information from bit stream.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when *pOffset* exceeds `[0,7]`.
`ippStsAacChanErr`	Indicates an error when *channelNum* exceeds `[1,2]`.
`ippStsAacMonoStereoErr`	Indicates an error when *monoStereoFlag* exceeds `[0,1]`.

DecodeExtensionHeader_AAC

Gets extension header information and extension layer information from bit stream.

Syntax

```
IppStatus ippsDecodeExtensionHeader_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACExtHeader* pAACExtHeader, int monoStereoFlag, int thisLayerStereo, int
    monoLayerFlag, int preStereoMaxSfb, int hightstMonoMaxSfb, int winSequence);
```

Parameters

ppBitStream	Double pointer to bitstream buffer. **ppBitStream* will be updated after decoding.
pOffset	Pointer to the offset in one byte. **pOffset* will be updated after decoding.
monoStereoFlag	Flag indicating that the current frame has mono and stereo layers.
thisLayerStereo	Flag indicating that the current layer is stereo.
monoLayerFlag	Flag indicating that the current frame has a mono layer.
preStereoMaxSfb	Previous stereo layer *maxSfb*.
hightstMonoMaxSfb	Last mono layer *maxSfb*.

winSequence	Window type, short or long.
pAACExtHeader	Pointer to the extension element header.

Description

This function is declared in the `ippac.h` file. The function gets extension header information and extension layer information from the bitstream.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when *pOffset* is out of the range of `[0,7]`.
`ippStsAacStereoLayerErr`	Indicates an error when *thisLayerStereo* exceeds `[0,1]`.
`ippStsAacMonoLayerErr`	Indicates an error when *monoLayerFlag* exceeds `[0,1]`.
`ippStsAacMaxSfbErr`	Indicates an error when at least one of the parameters *preStereoMaxSfb*, *hightstMonoMaxSfb* or the number of scale factor bands used in this layer exceeds `[0,IPP_AAC_MAX_SFB]`.
`ippStsAacMonoStereoErr`	Indicates an error when *monoStereoFlag* exceeds `[0,1]`.
`ippStsAacWinSeqErr`	Indicates an error when *winSequence* exceeds `[0,3]`.

DecodePNS_AAC

Implements perceptual noise substitution (PNS) coding within individual channel stream (ICS).

Syntax

```
IppStatus ippsDecodePNS_AAC_32s(Ipp32s* pSrcDstSpec, int* pSrcDstLtpFlag, Ipp8u*
    pSfbCb, Ipp16s* pScaleFactor, int maxSfb, int numWinGrp, int* pWinGrpLen,
    int samplingFreqIndex, int winLen, int* pRandomSeed);
```

Parameters

pSrcDstSpec	Pointer to spectrum coefficients for perceptual noise substitution (PNS).
pSrcDstLtpFlag	Pointer to long term predict (LTP) flag.
pSfbCb	Pointer to the scale factor codebook.
pScaleFactor	Pointer to the scalefactor value.
maxSfb	Number of scale factor bands used in this layer.
numWinGrp	Number of window groups.
pWinGrpLen	Pointer to the length of every window group.
samplingFreqIndex	Sampling frequency index.
winLen	Window length. 1024 for long windows, 128 for short windows.
pRandomSeed	Random seed for PNS.

Description

This function is declared in the `ippac.h` file. The function implements perceptual noise substitution (PNS) coding within the individual channel stream (ICS). Certain sets of spectral coefficients are derived from random vectors rather than from Huffman-coded symbols and inverse quantization process.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacMaxSfbErr`	Indicates an error when *maxSfb* exceeds `[0,IPP_AAC_MAX_SFB]`.
`ippStsAacSmplRateIdxErr`	Indicates an error when *samplingFreqIndex* exceeds [0,16].
`ippStsAacWinLenErr`	Indicates an error when *winLen* is not equal to 128 or 1024.

DecodeMsPNS_AAC

Implements perceptual noise substitution (PNS) coding in the case of joint coding.

Syntax

```
IppStatus ippsDecodeMsPNS_AAC_32s(Ipp32s* pSrcDstSpec, int* pSrcDstLtpFlag, Ipp8u*
    pSfbCb, Ipp16s* pScaleFactor, int maxSfb, int numWinGrp, int* pWinGrpLen,
    int samplingFreqIndex, int winLen, int* pRandomSeed, int channel, Ipp8u* pMsUsed,
    int* pNoiseState);
```

Parameters

pSrcDstSpec	Pointer to spectrum coefficients for perceptual noise substitution (PNS).
pSrcDstLtpFlag	Pointer to long term predict (LTP) flag.
pSfbCb	Pointer to the scalefactor code book.
pScaleFactor	Pointer to the scalefactor coefficients.
maxSfb	Number of maximum scalefactor band.
numWinGrp	Number of window groups.
pWinGrpLen	Pointer to the length of every window group.
samplingFreqIndex	Sampling frequency index.
winLen	Window length.
pRandomSeed	Random seed.
channel	Index of the current channel: 0 means left, 1 means right.
pMsUsed	Pointer to MS-used buffer in CPE structure.
pNoiseState	Pointer to the noise state buffer, which stores the left channel noise random seed for every scalefactor band. When *pMsUsed*[sfb]==1, the content in this buffer is used for the right channel.

Description

This function is declared in the `ippac.h` file. The function implements perceptual noise substitution (PNS) coding in the case of joint coding. Certain sets of spectral coefficients are derived from random vectors rather than from Huffman-coded symbols and inverse quantization process.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsAacMaxSfbErr	Indicates an error when *maxSfb* exceeds [0,IPP_AAC_MAX_SFB].
ippStsAacWinGrpErr	Indicates an error when *maxWinGrp* exceeds [0,8].
ippStsAacSmplRateIdxErr	Indicates an error when *samplingFreqIndex* exceeds [0,16].
ippStsAacWinLenErr	Indicates an error when *winLen* is not equal to 128 or 1024.

DecodeChanPairElt_MP4_AAC

Gets channel_pair_element from the input bitstream.

Syntax

```
IppStatus ippsDecodeChanPairElt_MP4_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACIcsInfo* pIcsInfo, IppAACChanPairElt* pChanPairElt, IppAACMainHeader*
    pAACMainHeader, int predSfbMax, int audioObjectType);
```

Parameters

ppBitStream	Double pointer to the current byte. **ppBitStream* will be updated after decoding.
pOffset	Pointer to the bit position in the byte pointed to by **ppBitStream*. Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
pIcsInfo	Pointer to IppAACIcsInfo structure.
pChanPairElt	Pointer to IppAACChanPairElt structure.
pAACMainHeader	Pointer to the main element header.
predSfbMax	Maximum prediction scale factor bands.

audioObjectType	Audio object type indicator: 1 indicates the main type 2 indicates the LC type 6 indicates the scalable mode.

Description

The `ippsDecodeChanPairElt_MP4_AAC` function is declared in the `ippac.h` file. This function gets the channel pair element from the input bitstream. Individual channel stream is not included.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when *pOffset* is out of the range of `[0,7]`.
`ippStsAacMaxSfbErr`	Indicates an error when *pIcsInfo->maxSfb* decoded from bitstream is greater than `IPP_AAC_MAX_SFB`, the maximum scale factor band for all sampling frequencies.

LongTermReconstruct_AAC

Uses Long Term Reconstruct (LTR) to reduce signal redundancy between successive coding frames.

Syntax

```
IppStatusippsLongTermReconstruct_AAC_32s(Ipp32s*pSrcEstSpec,Ipp32s*pSrcDstSpec,
    int* pLtpFlag, int winSequence, int samplingFreqIndex);
```

Parameters

pSrcDstSpec	Pointer to spectral coefficients for LTP.
pSrcEstSpec	Pointer to the frequency domain vector.
winSequence	Window type, long or short.
samplingFreqIndex	Sampling frequency index.
pLtpFlag	Pointer to the LTP flag.

Description

This function is declared in the `ippac.h` file. The function uses Long Term Reconstruct (LTP) to reduce signal redundancy between successive coding frames.

LTP is a forward adaptive predictor that is inherently less sensitive to the round-off numerical errors in the decoder or bi-errors in the transmitted spectral coefficients.

You should add the vector of decoded spectral coefficients and the corresponding frequency domain vector to get the vector of reconstructed spectral coefficients.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacSmplRateIdxErr`	Indicates an error when *`samplingFreqIndex`* exceeds `[0,12]`.
`ippStsAacWinSeqErr`	Indicates an error when *`winSequence`* exceeds `[0,3]`.

MDCTFwd_AAC_32s

Generates spectrum coefficient of PCM samples.

Syntax

```
IppStatus ippsMDCTFwd_AAC_32s(Ipp32s* pSrc, Ipp32s* pDst, Ipp32s*
    pSrcDstOverlapAdd, int winSequence, int winShape, int preWinShape, Ipp32s*
    pWindowedBuf);
```

Parameters

`pSrc`	Pointer to temporal signals to do MDCT.
`pDst`	Output of MDCT, the spectral coefficients of PCM samples.
`pSrcDstOverlapAdd`	Pointer to overlap buffer. Not used for MPEG-4 AAC decoding.
`winSequence`	Window sequence indicating if the block is long or short.
`winShape`	Current window shape.
`preWinShape`	Previous window shape.
`pWindowedBuf`	Work buffer for MDCT. Should be at least 2048 words.

Description

This function is declared in the `ippac.h` file. The function generates the spectrum coefficient of PCM samples in the MDCT Long Term Reconstruct (LTP) loop.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacWinSeqErr`	Indicates an error when *`winSequence`* exceeds `[0,3]`.
`ippStsAacWinShapeErr`	Indicates an error when *`preWinShape`* exceeds `[0,1]`.

MDCTFwd_AAC_32s_I

Computes forward modified discrete cosine transform (MDCT) of windowed signals.

Syntax

```
IppStatus ippsMDCTFwd_AAC_32s_I(Ipp32s* pSrcDst, int winSequence, int winShape,
    int prevWinShape, int len);
```

Parameters

`pSrcDst`	Pointer to the source and destination vector for the in-place operation.
`winSequence`	Window sequence indicator.
`winShape`	Window shape indicator.
`prevWinShape`	Window shape indicator of the previous frame.
`len`	Number of samples in Src buffer.

Description

The `ippsMDCTFwd_AAC_32s_I` function is declared in the `ippac.h` file. This function performs forward MDCT operation. Before MDCT is calculated, the source data is multiplied by the window described with *`winSequence`*, *`winShape`*, and *`prevWinShape`* parameters.

This function is implemented for using in AAC decoder. It does not perform any saturation so the user should consider overflow possibility. The source and destination have different positions of the decimal point (Q format). The Q format of destination depends on the window sequence indicator.

WinSequence = 0 (long window): If *Src* is in Qn format then *Dst* in Q(n-12) format

WinSequence = 1 (long start window): If *Src* is in Qn format then *Dst* in Q(n-12) format

WinSequence = 2 (eight short window): If *Src* is in Qn format then *Dst* in Q(n-9) format

WinSequence = 3 (long stop window): If *Src* is in Qn format then *Dst* in Q(n-12) format.

For example, if *Src* is in Q14 format, then *Dst* is in Q5 format in the case of eight short window and in Q2 in other cases.

Only *len* = 2048 is supported in the current implementation.

See Example 10-13 of using `ippsMDCTFwd_AAC_32s_I` function.

Example 10-13 `ippsMDCTFwd_AAC` Usage

```
#undef ONLY_LONG_SEQUENCE
#undef LONG_START_SEQUENCE
#undef EIGHT_SHORT_SEQUENCE
#undef LONG_STOP_SEQUENCE

#undef WINDOW_SHAPE_FHG
#undef WINDOW_SHAPE_DOLBY

#define ONLY_LONG_SEQUENCE   0
#define LONG_START_SEQUENCE  1
#define EIGHT_SHORT_SEQUENCE 2
#define LONG_STOP_SEQUENCE   3

#define WINDOW_SHAPE_FHG     0
#define WINDOW_SHAPE_DOLBY   1

IppStatus mdctfwdaac(void)
{
  Ipp32s pSrcDst[2048];
  IppStatus st;
  int i;

  for (i = 0; i < 2048; i++)
    pSrcDst[i] = (Ipp32s)(cos(IPP_2PI * i / 2048) * 16384); /* Q14 */

  st = ippsMDCTFwd_AAC_32s_I(pSrcDst, ONLY_LONG_SEQUENCE,
                             WINDOW_SHAPE_DOLBY, WINDOW_SHAPE_DOLBY, 2048);

  /* Output will be in Q2 format */
  printf("\n pSrcDst = ");
```

Example 10-13 ippsMDCTFwd_AAC Usage (continued)

```
  for (i = 0; i < 10; i++)
    printf("%i ", pSrcDst[i]);
  printf("\n");
  return st;
}

//Output (first 10 elements):
//   pSrcDst = 3174 -2549 -333 -300 26 -8 -3 -1 1 1
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the pointers passed to the function is NULL.
ippStsAacWinSeqErr	Indicates an error when *winSequence* exceeds [0,3].
ippStsAacWinShapeErr	Indicates an error when *winShape* or *prevWinShape* exceeds [0,1].
ippStsSizeErr	Indicates an error when *len* is not equal to 2048.

EncodeTNS_AAC

Performs reversion of TNS in the Long Term Reconstruct loop in-place.

Syntax

```
IppStatus ippsEncodeTNS_AAC_32s_I(Ipp32s* pSrcDst, const int* pTnsNumFilt,
    const int* pTnsRegionLen, const int* pTnsFiltOrder, const int*
    pTnsFiltCoefRes, const Ipp8s* pTnsFiltCoef, const int* pTnsDirection, int
    maxSfb, int profile, int samplingRateIndex, int winLen);
```

Parameters

pSrcDst	On input, pointer to the spectral coefficients for the TNS encoding operation. On output, pointer to the spectral coefficients after the TNS encoding operation.
pTnsNumFilt	Pointer to the number of TNS filters.
pTnsRegionLen	Pointer to the length of TNS filter.

pTnsFiltOrder	Pointer to the TNS filter order.
pTnsFiltCoefRes	Pointer to the TNS coefficient resolution flag.
pTnsFiltCoef	Pointer to the TNS filter coefficients.
pTnsDirection	Pointer to the TNS direction flag.
maxSfb	Maximum scale factor number.
profile	Audio profile.
samplingRateIndex	Sampling frequency index.
winLen	Window length.

Description

This function is declared in the `ippac.h` file. The function performs in-place reversion of TNS in the LTP loop, or Analysis Temporal Noise Shaping.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsTnsProfileErr`	Indicates an error when *profile* `!= 1`.
`ippStsAacTnsNumFiltErr`	Indicates an error when `*`*pTnsNumFilt* exceeds `[0, 1]` for the short window sequence or `[0, 3]` for the long window sequence.
`ippStsAacTnsLenErr`	Indicates an error when `*`*pTnsRegionLen* exceeds `[0,`*numSwb*`]`.
`ippStsAacTnsOrderErr`	Indicates an error when `*`*pTnsFiltOrder* exceeds `[0, 7]` for the short window sequence or `[0, 12]` for the long window sequence.
`ippStsAacTnsCoefResErr`	Indicates an error when `*`*pTnsFiltCoefRes* exceeds `[3, 4]`.
`ippStsAacTnsCoefErr`	Indicates an error when `*`*pTnsFiltCoef* exceeds `[-8, 7]`.
`ippStsAacTnsDirectErr`	Indicates an error when `*`*pTnsDirection* exceeds `[0,1]`.
`ippStsAacSmplRateIdxErr`	Indicates an error when *samplingRateIndex* exceeds `[0, 11]`.
`ippStsAacWinLenErr`	Indicates an error when *winLen* is not equal to 128 or 1024.

LongTermPredict_AAC

Gets the predicted time domain signals in the Long Term Reconstruct (LTP) loop.

Syntax

```
IppStatus ippsLongTermPredict_AAC_32s(Ipp32s* pSrcTimeSignal, Ipp32s*
    pDstEstTimeSignal, IppAACLtpInfo* pAACLtpInfo, int winSequence);
```

Parameters

pSrcTimeSignal	Pointer to the temporal signals to be predicted in the temporary domain.
pDstEstTimeSignal	Pointer to the output of samples after LTP.
pAACLtpInfo	Pointer to the LTP information.
winSequence	Window type, short or long.
pDstEstTimeSignal	Pointer to the prediction output in time domain.

Description

This function is declared in the `ippac.h` file. The function gets the predicted time domain signals in the LTP loop.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers is `NULL`.
`ippStsAacWinSeqErr`	Indicates an error when *winSequence* exceeds `[0,3]`.

NoiselessDecode_AAC

Performs noiseless decoding.

Syntax

```
IppStatus ippsNoiselessDecode_AAC(Ipp8u** ppBitStream, int* pOffset,
    IppAACMainHeader* pAACMainHeader, Ipp16s* pDstScaleFactor, Ipp32s*
    pDstQuantizedSpectralCoef, Ipp8u* pDstSfbCb, Ipp8s* pDstTnsFiltCoef,
    IppAACChanInfo* pChanInfo, int winSequence, int maxSfb, int commonWin, int
    scaleFlag, int audioObjectType);
```

Parameters

ppBitStream	Double pointer to the bitstream to be parsed. **ppBitStream* is be updated after decoding.
pOffset	Pointer to the offset in one byte. **pOffset* is be updated after decoding.
pAACMainHeader	Pointer to main header information. Not used for scalable objects. When *commonWin* `==` `0` `&&` *scaleFlag*`==0`, you need to decode LTP information and save it in *pAACMainHeader->pLtpInfo*`[]`.
pChanInfo	Pointer to channel information structure.
windowSequence	Window type, short or long.
maxSfb	Number of scale factor bands.
commonWin	Indicates if the channel pair uses the same ICS information.
scaleFlag	Flag indicating whether the scalable type is used.
audioObjectType	Audio object type indicator: 1 indicates the main type 2 indicates the LC type 6 indicates the scalable mode.
pDstScaleFactor	Pointer to the parsed scalefactor.
pDstQuantizedSpectralCoef	Pointer to the quantized spectral coefficients after Huffman decoding.
pDstSfbCb	Pointer to the scale factor codebook index.

pDstTnsFiltCoef	Pointer to TNS filter coefficients. Not used for scalable objects.

Description

This function is declared in the `ippac.h` file. This is a general noiseless decoding module for MPEG-2 and MPEG-4 objects.

In case one scale factor band uses PNS in MPEG-4 AAC scalable object, **pDstScaleFactor* contains the noise energy for this scale factor band and *pDstSfbCb*[sfb] should be `NOISE_HCB(13)`. The spectrum in this scale factor band is not necessarily Huffman-decoded, and the *pDstQuantizedSpectralCoef* of this scale factor band can be zero.

In AAC scalable object, *pDstTnsFiltCoef* and *pAACMainHeader* are not used.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when *pOffset* is out of the range `[0,7]`.
`ippStsAacComWinErr`	Indicates an error when *commonWin* exceeds `[0,1]`.
`ippStsAacMaxSfbErr`	Indicates an error when *maxSfb* exceeds `[0,IPP_AAC_MAX_SFB]`.
`ippStsAacSmplRateIdxErr`	Indicates an error when *pChanInfo->samplingRateIndex* exceeds `[0,11]`.
`ippStsAacCoefValErr`	Indicates an error when the quantized coefficients value pointed to by *pDstCoef* exceeds the range of `[-8191,8191]`.

LtpUpdate_AAC

Performs required buffer update in the Long Term Reconstruct (LTP) loop.

Syntax

```
IppStatus ippsLtpUpdate_AAC_32s(Ipp32s* pSpecVal, Ipp32s* pLtpSaveBuf, int
    winSequence, int winShape, int preWinShape, Ipp32s* pWorkBuf);
```

Parameters

pSpecVal	Pointer to spectral value after TNS decoder in LTP loop.
pLtpSaveBuf	Pointer to save buffer for LTP. Buffer length should be 3*frameLength. The value is saved for the next frame.
winSequence	Window type: 0 stands for long 1 stands for long start 2 stands for short 3 stands for long stop.
winShape	KBD or SIN window shape.
preWinShape	Previous window shape.
pWorkBuf	Work buffer for LTP update, length of *pWorkBuf* should be at least 2048*3 = 6144 words.

Description

This function is declared in the ippac.h file. The function performs required buffer update in the Long Term Reconstruct (LTP) loop. This operation includes IMDCT and updating the save buffer.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsAacWinSeqErr	Indicates an error when *winSequence* exceeds [0,3].
ippStsAacWinShapeErr	Indicates an error when *winShape* or *preWinShape* exceeds [0,1].

Spectral Band Replication Functions

SBR (Spectral Band Replication) is a new audio coding bandwidth extension technology, which is standardized in ISO/IEC 14496-3:2001/Amd.1:2003 [ISO14496A].

It improves the performance of low bitrate audio and speech codecs by either increasing the audio bandwidth at a given bitrate or by improving coding efficiency at a given quality level. For example, SBR can be used in conjunction with MP3 (mp3PRO), AAC (HE-AAC), CELP, HVXC to achieve a quality at 64 kbps stereo (MP3, AAC) that compares to conventional stereo at a bitrate exceeding 100 kbps.

SBR can be used with mono and stereo as well as with multichannel audio.

HE-AAC profile is combining AAC LC profile and SBR technology. HE-AAC is widely for mobile multimedia services.

In accordance with the mathematical model used by SBR in most audio materials a very high correlation is observed between the lower frequencies and the higher frequencies of a spectrum. Therefore “smart” transposition allows the transmitted signals in higher frequencies to be restructured so as to avoid coding and transmitting them as spectral data. The subsequent SBR adjustment block improves the highest reconstructed frequency.

SBR Audio Encoder Functions

This section provides a reference guide to Intel IPP for SBR audio encoder. Figure shows use of SBR Encoder for improvement of the core audio encoder.

Figure 10-4 SBR Encoder Usage

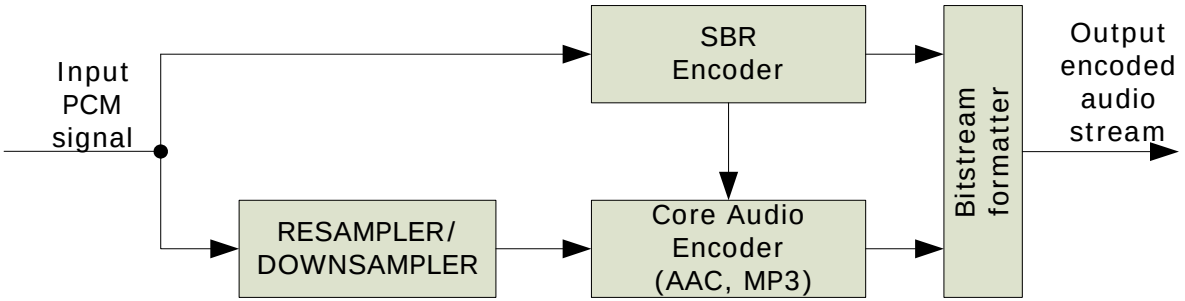
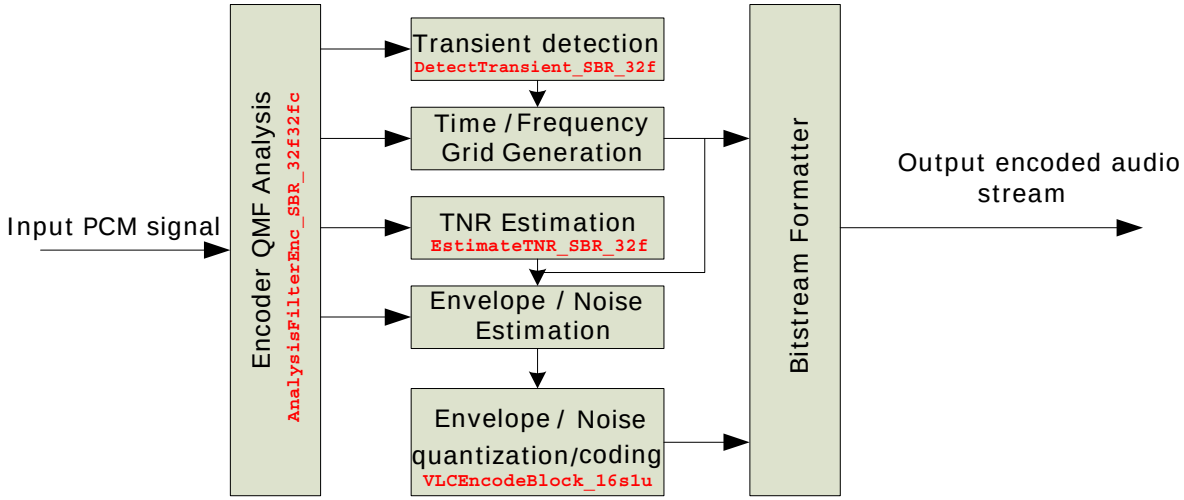


Figure 10-5 shows the common pipeline of SBR encoder.

Figure 10-5 SBR Encoder Pipeline



Intel IPP functions described in this section implement algorithms that can be used to develop HE-AAC Encoder compliant with the ETSI/3GPP ([EC126]) standards.

DetectTransient_SBR

Estimates stationarity of signal by calculating deviation of spectrum audio signal.

Syntax

```
IppStatus ippsDetectTransient_SBR_32f(const Ipp32f* pSrc, Ipp32f*
    pInOutThr, Ipp32f* pDst);
```

Parameters

pSrc	Pointer to the vector that stores the spectrum component of the audio signal.
pInOutThr	Pointer to the threshold of deviation.

pDst	Pointer to the current relative deviations.

Description

The function `ippsDetectTransient_SBR_32f` is declared in the `ippac.h` file. This function implements transient detection algorithm described by [EC126] standard. The function may be used by SBR encoder to estimate stationarity of the signal by calculation of 2 thresholds. The output from this function is the *pDst* vector.

pInOutThr and *pDst* are static channel-dependent arrays of length 64 that need to be stored between calls of `ippsDetectTransient`. On start-up, all elements in both arrays should be set to zero. These arrays are updated by the function.

Before the first call of `ippsDetectTransient` (over one frame), the *pDst* vector has to be updated by the external application as follows:

$$pDst[n] = pDst[n+32], \quad 0 \le n < 16;$$

$$pDst[n] = 0, \quad 16 \le n < 48.$$

The functionality of `ippsDetectTransient` is described as follows:

$$m = \frac{1}{48} \sum_{n=16}^{63} pSrc[n]$$

$$\sigma = \sqrt{\frac{1}{47} \sum_{n=16}^{63} (m - pSrc[n])^2}$$

pInOutThr[0]=*MAX*(128000, 0.66·*pInOutThr*[0]+0.34·σ

```
for(n=16; n<48; n++)
   for(d=1; d<4; d++)
```

$$L = 0.5 \cdot \left\{ pSrc\left[2 \cdot NINT\left(\frac{n-d}{2}\right)\right] + pSrc\left[2 \cdot NINT\left(\frac{n-d}{2}\right)\right] + 1 \right\},$$

$$R = 0.5 \cdot \left\{ pSrc\left[2 \cdot NINT\left(\frac{n+d}{2}\right)\right] + pSrc\left[2 \cdot NINT\left(\frac{n+d}{2}\right)\right] + 1 \right\},$$

$$\begin{cases} pDst[n] = pDst[n] + \dfrac{R - L - pInOutThr[i]}{pInOutThr[i]}, & R - L > pThr[i] \\ pDst[n] = pDst[n], & \text{otherwise} \end{cases}$$

This function is called 64 times per one frame for each channel.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.

EstimateTNR_SBR

Measures Tonality-to-Noise Ratio of complex QMF subband samples.

Syntax

```
IppStatus ippsEstimateTNR_SBR_32f(const Ipp32fc* pSrc, Ipp32f* pTNR0,
    Ipp32f* pTNR1, Ipp32f* pMeanNrg);
```

Parameters

pSrc	Pointer to the vector that stores the QMF-processed subband samples for one subband.
pTNR0	Pointer to the first TNR estimate of the input signal.
pTNR1	Pointer to the second TNR estimate of the input signal.
pMeanNrg	Pointer to mean energy of QMF-processed subband.

Description

The function `ippsEstimateTNR_SBR_32f` is declared in the `ippac.h` file. This function implements Tonality-to-Noise Ratio algorithm described by [EC126] standard. The function may be used by SBR encoder to estimate TNR of the input signal. The outputs from this function are TNR measurements of the input signal. These estimates may be are used by the noise estimation algorithm, inverse filtering estimation algorithm, and sines estimation algorithm.

The functionality of `ippsEstimateTNR` is described as follows:

$$\phi_l(i,\ j) = \sum_{n=2}^{15} pSrc[n-i+16\cdot\ l]\cdot\ conj(pSrc[n-j+16\cdot\ l]),\begin{cases}0 \le i < 3\\ 1 \le j < 3\\ 0 \le l < 1\end{cases}$$

$$d^l = \phi_l(2,\ 2)\cdot\ \phi_l(1,\ 1) - \frac{1}{1+10^{-6}}\cdot\ |\phi_l(1,\ 2)|^2,$$

$$\alpha_1^l = \begin{cases}\dfrac{\phi_l(0,\ 1)\cdot\ \phi_l(1,\ 2) - \phi_l(0,\ 2)\cdot\ \phi_l(1,\ 1)}{d^l}, & d^l \ne 0\\ 0 & d^l = 0\end{cases}$$

$$\alpha_0^l = \begin{cases}-\dfrac{\phi_l(0,\ 1) + \alpha_1^l\cdot\ conj(\phi_l(1,\ 2))}{\phi_l(1,\ 1)}, & \phi_l(1,\ 1) \ne 0\\ 0 & \phi_l(1,\ 1) = 0\end{cases}$$

$$\begin{cases}pTNR0[0] = f(0)\\ pTNR1[0] = f(1)\end{cases},$$

$$f(l) = \frac{\mathrm{real}\{\alpha_0^l\cdot\ conj(\phi_l(0,\ 1)) + \alpha_1^l\cdot\ conj(\phi_l(0,\ 2))\}}{\mathrm{real}\{\phi_l(0,\ 0)\} - \mathrm{real}\{\alpha_0^l\cdot\ conj(\phi_l(0,\ 1)) + \alpha_1^l\cdot\ conj(\phi_l(0,\ 2))\}}$$

$$pMeanNrg[0] = \frac{1}{2}\sum_{l=0}^{1} Re\{\phi_l(0,\ 0)\}$$

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

AnalysisFilterEncGetSize_SBR

Returns size of analysis FilterSpec_SBR *specification structures, init and work buffers.*

Syntax

```
IppStatus ippsAnalysisFilterEncGetSize_SBR_32f(int* pSizeSpec, int*
    pSizeInitBuf);
```

Parameters

pSizeSpec	Address of the FilterSpec_SBR specification structure size value in bytes.
pSizeInitBuf	Address of size value (in bytes) of the buffer for initialization functions.

Description

This function is declared in the ippac.h file. The function returns the size of the specification structure and the initialization process buffer.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

AnalysisFilterEncInit_SBR

Initializes analysis specification structure.

Syntax

```
IppStatus ippsAnalysisFilterEncInit_SBR_32f(IppsFilterSpec_SBR_C_32f*
    pFilterSpec, Ipp8u *pInitBuf);
```

Parameters

pFilterSpec	Pointer to the `FilterSpec_SBR` specification structure.
pInitBuf	pointer to the `Init` buffer.

Description

This function is declared in the `ippac.h` file. The function initializes the SBR specification structure `pFilterSpec`. Before calling this function, the size of the SBR specification structure should be computed by `AnalysisFilterEncGetSize_SBR`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.

AnalysisFilterEnc_SBR

Performs subband filtering of the input audio signal.

Syntax

```
IppStatus ippsAnalysisFilterEnc_SBR_32f32fc(const Ipp32f* pSrc, Ipp32fc*
    pDst, const IppsFilterSpec_SBR_C_32f* pFilterSpec);
```

Parameters

pSrc	Pointer to the input audio signal. • most recent 576 (640-64) samples should be contained in the vector *pSrc*[*i*], where *i* = 0,1,..,575. • samples associated with the current frame should be contained in the vector *pSrc*[*j*], where *j* = 576, ..,639.
pDst	Pointer to the vector; contains subband filtering samples.
pFilterSpec	Pointer to the `FilterSpec_SBR` specification structure.

Description

This function is declared in the `ippac.h` file. The function performs subband filtering of the audio signal. Call the `ippsAnalysisEncFilter_SBR_32f32fc` function 32 times per frame on each channel.

The functionality of `ippsAnalysisFilterEnc_SBR_32f32fc` is described as follows:

$$z[n] = pSrc[n] \cdot c[n], \quad 0 \le n < 640$$

$$u[n] = \sum_{j=0}^{4} z[n + 128 \cdot j], \quad 0 \le n < 640$$

$$pDst[k] = \sum_{n=0}^{127} u[n] \cdot \exp\left\{\frac{\pi}{128} \cdot i \cdot \left(k + \frac{1}{2}\right)(2n+1)\right\}, \quad 0 \le k < 64$$

where coefficients $c[n]$ are coefficients of the QMF bank window [ISO14496A].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the *`pFilterSpec`* identifier is incorrect.

AnalysisFilterEncInitAlloc_SBR

Creates and initializes analysis specification structure.

Syntax

```
IppStatus
    ippsAnalysisFilterEncInitAlloc_SBR_32f(IppsFilterSpec_SBR_C_32f**
    ppFilterSpec);
```

Parameters

`ppFilterSpec`	Pointer to pointer to the analysis specification structure to be created.

Description

This function is declared in the `ippac.h` file. The function creates and initializes the analysis specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

AnalysisFilterEncFree_SBR

Closes analysis filter specification structure.

Syntax

```
IppStatus ippsAnalysisFilterEncFree_SBR_32f(IppsFilterSpec_SBR_C_32f*
    pFilterSpec);
```

Parameters

`pFilterSpec`	Pointer to the analysis specification structure to be closed.

Description

This function is declared in the `ippac.h` file. The function closes the analysis specification structure *`pFilterSpec`* by freeing all memory associated with the specification created by `AnalysisFilterEncInitAlloc_SBR`. Call `AnalysisFilterEncInitAlloc_SBR` after the transform is completed.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pFilterSpec`* is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the *`pFilterSpec`* identifier is incorrect.

SBR Audio Decoder Functions

This section provides a reference guide to Intel IPP for SBR audio decoder. Figure shows the audio decoder pipeline that supports SBR technology.

Figure 10-6 SBR Decoder Usage

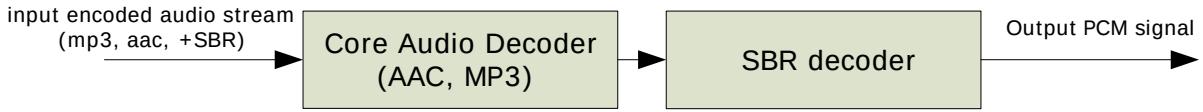
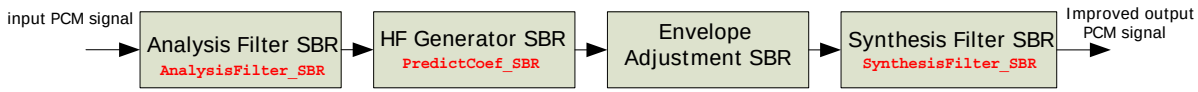


Figure shows the common pipeline of SBR decoder.

Figure 10-7 SBR Decoder Pipeline



In accordance with the [EC126] standard the SBR decoding technology can operate on complex-valued signals in the mode called High Quality (HQ) or on real-valued signals in the mode called Low Power (LP)). On the other hand, the SBR decoding technology can produce output signals with core sample rate or double sample rate. All these modes are supported by Intel IPP for SBR audio decoder.

QMF Functions

A Quadrature Mirror Filter (QMF) bank is used by MPEG-4 SBR decoder. QMF consists of:

— Analysis Filter

— Synthesis Filter

— Synthesis Down Filter.

- Analysis Filter bank is used to split the time domain signal output from the core decoder into 32 subband signals. The output from the filterbank, that is, the subband samples, is complex-valued for HQ SBR mode or real-valued for LP SBR mode.

- Synthesis Filter bank is used to filter SBR-processed subband signals. The output from the filterbank is real-valued time domain samples. Filtering is achieved using a 64-subband Filter bank.
- Down Synthesis Filter bank is used to filter SBR-processed subband signals. The output from the filterbank is real-valued time domain samples. Filtering is achieved using a 32-subband Filter bank.

QMF filterbanks use the functions given below in low power (LP) and high quality (HQ) modes. Each mode is characterized by three function groups:

— AnalysisFilter

— SynthesisFilter

— SynthesisDownFilter.

To use the SBR_QMF functions, initialize the `FilterSpec_SBR` specification structure, which contains tables of twiddle factors and internal buffer. To allocate memory and initialize the structure, call `ipps(**)FilterInitAlloc_SBR_(***)` functions, where `(**)` is a function group name, that is `Analysis`, `Synthesis` or `SynthesisDown`, and `(***)` is a flavor descriptor (for example, `_SBRHQ_32s32sc`, `_SBRLP_32s`, etc.) Currently, this method of the specification structure initialization is implemented only for operations on fixed-point signal.

Alternatively, the structure can be initialized by functions for initialization and size getting. In that case, the general operational algorithm is as follows:

1. Call `ipps(**)FIlterGetSize_SBR_(***)`, where `(**)` is a function group name, that is `Analysis`, `Synthesis` or `SynthesisDown`, and `(***)` is a flavor descriptor (for example, `_RToC_32f32fc`, `_RToR_32f`, etc.)
2. Allocate memory (external operation)
3. Call `ipps(**)FilterInit_SBR_(***)` to initialize the corresponding main function
4. Use `ipps(**)Filter_SBR_(***)` for the main operation.
5. Deallocate the memory (external operation).

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from
http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm .

Memory Allocation and Initialization

These functions allocate memory and initialize SBR specification structure for real (LP) and complex (HQ) signals.

AnalysisFilterInitAlloc_SBR

Allocates memory and initializes analysis SBR specification structure for real and complex signals.

Syntax

```
IppStatus ippsAnalysisFilterInitAlloc_SBRHQ_32s32sc(IppsFilterSpec_SBR_C_32sc**
    ppFilterSpec);
IppStatus ippsAnalysisFilterInitAlloc_SBRLP_32s(IppsFilterSpec_SBR_R_32s**
    ppFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the `FilterSpec_SBR` specification structure to be created.

Description

These functions are declared in the `ippac.h` file. The functions allocate memory, create, and initialize the analysis `FilterSpec_SBR` specification structure.

The `ippsAnalysisFilterInitAlloc_SBRHQ_32s32sc` function allocates memory and initializes the complex specification structure.

The `ippsAnalysisFilterInitAlloc_SBRLP_32s` function allocates memory and initializes the real specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *ppFilterSpec* pointer is `NULL`.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

SynthesisFilterInitAlloc_SBR

Allocates memory and initializes synthesis SBR specification structure for real and complex signals.

Syntax

```
IppStatus
    ippsSynthesisFilterInitAlloc_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc**
    ppFilterSpec);
IppStatus ippsSynthesisFilterInitAlloc_SBRLP_32s(IppsFilterSpec_SBR_R_32s**
    ppFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the `FilterSpec_SBR` specification structure to be created.

Description

These functions are declared in the `ippac.h` file. The functions allocate memory, create, and initialize the synthesis `FilterSpec_SBR` specification structure.

The `ippsSynthesisFilterInitAlloc_SBRHQ_32sc32s` function allocates memory and initializes the complex specification structure.

The `ippsSynthesisFilterInitAlloc_SBRLP_32s` function allocates memory and initializes the real specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *ppFilterSpec* pointer is `NULL`.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

SynthesisDownFilterInitAlloc_SBR

Allocates memory and initializes downsample synthesis SBR specification structure for real and complex signals.

Syntax

```
IppStatus
    ippsSynthesisDownFilterInitAlloc_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc**
    ppFilterSpec);
IppStatus ippsSynthesisDownFilterInitAlloc_SBRLP_32s(IppsFilterSpec_SBR_R_32s**
    ppFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the `FilterSpec_SBR` specification structure to be created.

Description

These functions are declared in the `ippac.h` file. The functions allocate memory, create, and initialize the downsample synthesis `FilterSpec_SBR` specification structure.

The `ippsSynthesisDownFilterInitAlloc_SBRHQ_32sc32s` function allocates memory and initializes the complex specification structure.

The `ippsSynthesisDownFilterInitAlloc_SBRLP_32s` function allocates memory and initializes the real specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *ppFilterSpec* pointer is `NULL`.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

Memory Release

These functions close SBR specification structure for real and complex signals.

AnalysisFilterFree_SBR

Closes analysis SBR specification structure for real and complex signals.

Syntax

```
IppStatus ippsAnalysisFilterFree_SBRHQ_32s32sc(IppsFilterSpec_SBR_C_32sc*
    pFilterSpec);
IppStatus ippsAnalysisFilterFree_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
    pFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the `FilterSpec_SBR` specification structure to be closed.

Description

These functions are declared in the `ippac.h` file. The functions close the analysis `FilterSpec_SBR` specification structure by freeing all memory associated with the specification created by `ippsAnalysisFilterInitAlloc_SBRHQ_32s32sc` or `ippsAnalysisFilterInitAlloc_SBRLP_32s`. Call `ippsAnalysisFilterFree` after the transform is completed.

The `ippsAnalysisFilterFree_SBRHQ_32s32sc` function closes the complex SBR specification structure.

The `ippsAnalysisFilterInitAlloc_SBRLP_32s` function closes the real SBR specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pFilterSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pFilterSpec* is incorrect.

SynthesisFilterFree_SBR

Closes synthesis SBR specification structure for real and complex signals.

Syntax

```
IppStatus ippsSynthesisFilterFree_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc*
    pFilterSpec);
IppStatus ippsSynthesisFilterFree_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
    pFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the `FilterSpec_SBR` specification structure to be closed.

Description

These functions are declared in the `ippac.h` file. The functions close the synthesis `FilterSpec_SBR` specification structure by freeing all memory associated with the specification created by `ippsSynthesisFilterInitAlloc_SBRHQ_32sc32s` or `ippsSynthesisFilterInitAlloc_SBRLP_32s`. Call `ippsSynthesisFilterFree` after the transform is completed.

The `ippsSynthesisFilterFree_SBRHQ_32sc32s` function closes the complex SBR specification structure.

The `ippsSynthesisFilterInitAlloc_SBRLP_32s` function closes the real SBR specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pFilterSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pFilterSpec* is incorrect.

SynthesisDownFilterFree_SBR

Closes downsample synthesis SBR specification structure for real and complex signals.

Syntax

```
IppStatus ippsSynthesisDownFilterFree_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc*
    pFilterSpec);
IppStatus ippsSynthesisDownFilterFree_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
    pFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the `FilterSpec_SBR` specification structure to be closed.

Description

These functions are declared in the `ippac.h` file. The functions close the synthesis `FilterSpec_SBR` specification structure by freeing all memory associated with the specification created by `ippsSynthesisDownFilterInitAlloc_SBRHQ_32sc32s` or `ippsSynthesisDownFilterInitAlloc_SBRLP_32s`. Call `ippsSynthesisDownFilterFree` after the transform is completed.

The `ippsSynthesisDownFilterFree_SBRHQ_32sc32s` function closes the complex SBR specification structure.

The `ippsSynthesisDownFilterInitAlloc_SBRLP_32s` function closes the real SBR specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pFilterSpec* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pFilterSpec* is incorrect.

Size Getting

These functions return the size of `FilterSpec` specification structures in bytes. The functions with floating values (`_32f`, `_32fc`) also return the initialization process buffer and the work buffer.

AnalysisFilterGetSize_SBR

Returns size of analysis `FilterSpec_SBR` *specification structures, init and work buffers.*

Syntax

Case 1: Operation on float-point signal.

```
IppStatus ippsAnalysisFilterGetSize_SBR_RToC_32f32fc(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippsAnalysisFilterGetSize_SBR_RToC_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippsAnalysisFilterGetSize_SBR_RToR_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsAnalysisFilterGetSize_SBRHQ_32s32sc(int* pSizeSpec);
IppStatus ippsAnalysisFilterGetSize_SBRLP_32s(int* pSizeSpec);
```

Parameters

pSizeSpec	Address of the `FilterSpec_SBR` specification structure size value in bytes.
pSizeInitBuf	Address of size value (in bytes) of the buffer for initialization functions.
pSizeWorkBuf	Address of the work buffer size value in bytes.

Description

These functions are declared in the `ippac.h` file. The functions return the size of the specification structure for `Init` functions for Analysis Filter bank. The functions with floating point values, that is, with suffixes `_32f32fc` and `_32f` also return the initialization process buffer and the work buffer.

`ippsAnalysisFilterGetSize_SBR_RToC_32f32fc` returns the specified sizes for `ippsAnalysisFilter_SBR_RToC_32f32fc_D2L`.

`ippsAnalysisFilterGetSize_SBR_RToC_32f` returns the specified sizes for `ippsAnalysisFilter_SBR_RToC_32f_D2L`.

`ippsAnalysisFilterGetSize_SBR_RToR_32f` returns the specified sizes for `ippsAnalysisFilter_SBR_RToR_32f_D2L`.

`ippsAnalysisFilterGetSize_SBRHQ_32s32sc` returns the size of specification structure for `ippsAnalysisFilter_SBRHQ_32s32sc`.

`ippsAnalysisFilterGetSize_SBRLP_32s` returns the size of specification structure for `ippsAnalysisFilter_SBRLP_32s`.

AAC decoder included into IPP Samples makes use of `ippsAnalysisFilterGetSize_SBR_RToC_32f` and `ippsAnalysisFilterGetSize_SBR_RToR_32f` functions in the float-point version and `ippsAnalysisFilterGetSize_SBRHQ_32s32sc` and `ippsAnalysisFilterGetSize_SBRLP_32s` functions in the fixed-point version. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.

SynthesisFilterGetSize_SBR

Returns size of synthesis `FilterSpec_SBR` *specification structure, init and work buffers.*

Syntax

Case 1: Operation on float-point signal.

```
IppStatus ippsSynthesisFilterGetSize_SBR_CToR_32fc32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippsSynthesisFilterGetSize_SBR_CToR_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
```

```
IppStatus ippsSynthesisFilterGetSize_SBR_RToR_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisFilterGetSize_SBRHQ_32sc32s(int* pSizeSpec);
IppStatus ippsSynthesisFilterGetSize_SBRLP_32s(int* pSizeSpec);
```

Parameters

pSizeSpec	Address of the `FilterSpec_SBR` specification structure size value in bytes.
pSizeInitBuf	Address of size value (in bytes) of the buffer for initialization functions.
pSizeWorkBuf	Address of the work buffer size value in bytes.

Description

These functions are declared in the `ippac.h` file. The functions return the size of the specification structure for `Init` functions for Synthesis Filter bank. The functions with floating point values, that is, with suffixes `_32fc32f` and `_32f` also return the initialization process buffer and the work buffer.

`ippsSynthesisFilterGetSize_SBR_CToR_32fc32f` returns the specified sizes for `ippsSynthesisFilter_SBR_CToR_32fc32f_D2L`.

`ippsSynthesisFilterGetSize_SBR_CToR_32f` returns the specified sizes for `ippsSynthesisFilter_SBR_CToR_32f_D2L`.

`ippsSynthesisFilterGetSize_SBR_RToR_32f` returns the specified sizes for `ippsSynthesisFilter_SBR_RToR_32f_D2L`.

`ippsSynthesisFilterGetSize_SBRHQ_32sc32s` returns the size of specification structure for `ippsSynthesisFilter_SBRHQ_32sc32s`.

`ippsSynthesisFilterGetSize_SBRLP_32s` returns the size of specification structure for `ippsSynthesisFilter_SBRLP_32s`.

AAC decoder included into IPP Samples makes use of `ippsSynthesisFilterGetSize_SBR_RToC_32f` and `ippsSynthesisFilterGetSize_SBR_RToR_32f` functions in the float-point version and `ippsSynthesisFilterGetSize_SBRHQ_32sc32s` and `ippsSynthesisFilterGetSize_SBRLP_32s` functions in the fixed-point version. See introduction to this section.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

SynthesisDownFilterGetSize_SBR

Returns size of downsample synthesis FilterSpec_SBR *specification structure, init and work buffers.*

Syntax

Case 1: Operation on float-point signal.

```
IppStatus ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippsSynthesisDownFilterGetSize_SBR_CToR_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
IppStatus ippsSynthesisDownFilterGetSize_SBR_RToR_32f(int* pSizeSpec, int*
    pSizeInitBuf, int* pSizeWorkBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisDownFilterGetSize_SBRHQ_32sc32s(int* pSizeSpec);
IppStatus ippsSynthesisDownFilterGetSize_SBRLP_32s(int* pSizeSpec);
```

Parameters

pSizeSpec	Address of the FilterSpec_SBR specification structure size value in bytes.
pSizeInitBuf	Address of size value (in bytes) of the buffer for initialization functions.
pSizeWorkBuf	Address of the work buffer size value in bytes.

Description

These functions are declared in the ippac.h file. The functions return the size of the specification structure for Init functions for Synthesis Down Filter bank. The functions with floating point values, that is, with suffixes _32fc32f and _32f also return the initialization process buffer and the work buffer.

ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f returns the specified sizes for ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L.

ippsSynthesisDownFilterGetSize_SBR_CToR_32f returns the specified sizes for ippsSynthesisDownFilter_SBR_CToR_32f_D2L.

ippsSynthesisDownFilterGetSize_SBR_RToR_32f returns the specified sizes for ippsSynthesisDownFilter_SBR_RToR_32f_D2L.

ippsSynthesisDownFilterGetSize_SBRHQ_32sc32s returns the size of specification structure for ippsSynthesisDownFilter_SBRHQ_32sc32s.

ippsSynthesisDownFilterGetSize_SBRLP_32s returns the size of specification structure for ippsSynthesisDownFilter_SBRLP_32s.

AAC decoder included into IPP Samples makes use of ippsSynthesisDownFilterGetSize_SBR_RToC_32f and ippsSynthesisDownFilterGetSize_SBR_RToR_32f functions in the float-point version and ippsSynthesisDownFilterGetSize_SBRHQ_32sc32s and ippsSynthesisDownFilterGetSize_SBRLP_32s functions in the fixed-point version. See introduction to this section.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

Initialization

These functions initialize the `FilterSpec` context.

AnalysisFilterInit_SBR

Initializes analysis specification structure.

Syntax

Case 1: Operation on float-point signal.

```
IppStatus ippsAnalysisFilterInit_SBR_RToC_32f32fc(IppsFilterSpec_SBR_C_32fc**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsAnalysisFilterInit_SBR_RToC_32f(IppsFilterSpec_SBR_C_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsAnalysisFilterInit_SBR_RToR_32f(IppsFilterSpec_SBR_R_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsAnalysisFilterInit_SBRHQ_32s32sc(IppsFilterSpec_SBR_C_32sc*
    pFilterSpec);
IppStatus ippsAnalysisFilterInit_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
    pFilterSpec);
```

Parameters

ppFilterSpec	Pointer to where pointer to the `FilterSpec_SBR` specification structure is written. Only the main function uses this structure. Note, that for functions with integer values `_32s32sc` and `_32s` this parameter is a single pointer *pFilterSpec* to the `FilterSpec_SBR` specification structure.
pMemSpec	Pointer to the area for the `FilterSpec_SBR` specification structure.
pInitBuf	Pointer to the `Init` buffer.

Description

These functions are declared in the `ippac.h` file. The functions initialize the analysis structures:

`ippsAnalysisFilterInit_SBR_RToC_32f32fc` initializes the specification structure for `ippsAnalysisFilter_SBR_RToC_32f32fc_D2L`.

`ippsAnalysisFilterInit_SBR_RToC_32f` initializes the specification structure for `ippsAnalysisFilter_SBR_RToC_32f_D2L`.

`ippsAnalysisFilterInit_SBR_RToR_32f` initializes the specification structure for `ippsAnalysisFilter_SBR_RToR_32f_D2L`.

`ippsAnalysisFilterInit_SBRHQ_32s32sc` initializes the specification structure for `ippsAnalysisFilter_SBRHQ_32s32sc`.

`ippsAnalysisFilterInit_SBRLP_32s` initializes the specification structure for `ippsAnalysisFilter_SBRLP_32s`.

AAC decoder included into IPP Samples makes use of `ippsAnalysisFilterInit_SBR_RToC_32f` and `ippsAnalysisFilterInit_SBR_RToR_32f` functions in the float-point version and `ippsAnalysisFilterInit_SBRHQ_32s32sc` and `ippsAnalysisFilterInit_SBRLP_32s` functions in the fixed-point version. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.

SynthesisFilterInit_SBR

Initializes synthesis specification structure.

Syntax

Case 1: Operation on float-point signal.

```
IppStatus ippsSynthesisFilterInit_SBR_CToR_32fc32f(IppsFilterSpec_SBR_C_32fc**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsSynthesisFilterInit_SBR_CToR_32f(IppsFilterSpec_SBR_C_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsSynthesisFilterInit_SBR_RToR_32f(IppsFilterSpec_SBR_R_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisFilterInit_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc*
    pFilterSpec);
IppStatus ippsSynthesisFilterInit_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
    pFilterSpec);
```

Parameters

ppFilterSpec	Pointer to where pointer to the `FilterSpec_SBR` specification structure is written. Only the main function uses this structure. Note, that for functions with integer values `_32sc32s` and `_32s` this parameter is a single pointer *pFilterSpec* to the `FilterSpec_SBR` specification structure.
pMemSpec	Pointer to the area for the `FilterSpec_SBR` specification structure.
pInitBuf	Pointer to the `Init` buffer.

Description

These functions are declared in the `ippac.h` file. The functions initialize the synthesis structures:

`ippsSynthesisFilterInit_SBR_CToR_32fc32f` initializes the specification structure for `ippsSynthesisFilter_SBR_CToR_32fc32f_D2L`.

`ippsSynthesisFilterInit_SBR_CToR_32f` initializes the specification structure for `ippsSynthesisFilter_SBR_CToR_32f_D2L`.

`ippsSynthesisFilterInit_SBR_RToR_32f` initializes the specification structure for `ippsSynthesisFilter_SBR_RToR_32f_D2L`.

`ippsSynthesisFilterInit_SBRHQ_32sc32s` initializes the specification structure for `ippsSynthesisFilter_SBRHQ_32sc32s`.

`ippsSynthesisFilterInit_SBRLP_32s` initializes the specification structure for `ippsSynthesisFilter_SBRLP_32s`.

AAC decoder included into IPP Samples makes use of `ippsSynthesisFilterInit_SBR_RToC_32f` and `ippsSynthesisFilterInit_SBR_RToR_32f` functions in the float-point version and `ippsSynthesisFilterInit_SBRHQ_32sc32s` and `ippsSynthesisFilterInit_SBRLP_32s` functions in the fixed-point version. See introduction to this section.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

SynthesisDownFilterInit_SBR

Initializes synthesis down specification structure.

Syntax

Case 1: Operation on float-point signal.

```
IppStatus
    ippsSynthesisDownFilterInit_SBR_CToR_32fc32f(IppsFilterSpec_SBR_C_32fc**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsSynthesisDownFilterInit_SBR_CToR_32f(IppsFilterSpec_SBR_C_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
IppStatus ippsSynthesisDownFilterInit_SBR_RToR_32f(IppsFilterSpec_SBR_R_32f**
    ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisDownFilterInit_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc*
    pFilterSpec);
IppStatus ippsSynthesisDownFilterInit_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
    pFilterSpec);
```

Parameters

ppFilterSpec	Pointer to where pointer to the FilterSpec_SBR specification structure is written. Only the main function uses this structure. Note, that for functions with integer values _32sc32s and _32s this parameter is a single pointer *pFilterSpec* to the FilterSpec_SBR specification structure.
pMemSpec	Pointer to the area for the FilterSpec_SBR specification structure.
pInitBuf	Pointer to the Init buffer.

Description

These functions are declared in the `ippac.h` file. The functions initialize the down sample synthesis structures:

`ippsSynthesisDownFilterInit_SBR_CToR_32fc32f` initializes the specification structure for `ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L`.

`ippsSynthesisDownFilterInit_SBR_CToR_32f` initializes the specification structure for `ippsSynthesisDownFilter_SBR_CToR_32f_D2L`.

`ippsSynthesisDownFilterInit_SBR_RToR_32f` initializes the specification structure for `ippsSynthesisDownFilter_SBR_RToR_32f_D2L`.

`ippsSynthesisDownFilterInit_SBRHQ_32sc32s` initializes the specification structure for `ippsSynthesisDownFilter_SBRHQ_32sc32s`.

`ippsSynthesisDownFilterInit_SBRLP_32s` initializes the specification structure for `ippsSynthesisDownFilter_SBRLP_32s`.

AAC decoder included into IPP Samples makes use of `ippsSynthesisDownFilterInit_SBR_RToC_32f` and `ippsSynthesisDownFilterInit_SBR_RToR_32f` functions in the float-point version and `ippsSynthesisDownFilterInit_SBRHQ_32sc32s` and `ippsSynthesisDownFilterInit_SBRLP_32s` functions in the fixed-point version. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.

QMF Main Functions

AnalysisFilter_SBR

Transforms time domain signal output from the core decoder into frequency subband signals.

Syntax

Case 1: Operation on float-point signal.

```
IppStatus ippsAnalysisFilter_SBR_RToC_32f32fc_D2L(const Ipp32f* pSrc, Ipp32fc*
    pDst[], const Ipp32f* pSbrTableWindowDown, int NumLoop, int offset, int kx,
    const IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);
IppStatus ippsAnalysisFilter_SBR_RToC_32f_D2L(const Ipp32f* pSrc, Ipp32f*
    pDstRe[], Ipp32f* pDstIm[], const Ipp32f* pSbrTableWindowDown, int NumLoop,
    int offset, int kx, const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u*
    pWorkBuf);
IppStatus ippsAnalysisFilter_SBR_RToR_32f_D2L(const Ipp32f* pSrc, Ipp32f*
    pDst[], const Ipp32f* pSbrTableWindowDown, int NumLoop, int offset, int kx,
    const IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsAnalysisFilter_SBRHQ_32s32sc(const Ipp32s* pSrc, Ipp32sc* pDst,
    int kx, const IppsFilterSpec_SBR_C_32sc* pFilterSpec);
IppStatus ippsAnalysisFilter_SBRLP_32s(const Ipp32s* pSrc, Ipp32s* pDst, int
    kx, const IppsFilterSpec_SBR_R_32s* pFilterSpec);
```

Parameters

pSrc	Pointer to the input audio signal.
pDst[]	Array of pointers, holds subband samples. Note, that for functions with integer values _32s32sc and _32s this parameter is the *pDst* pointer to the subband samples.
pDstRe[]	Array of pointers that contains real parts of subband samples.
pDstIm[]	Array of pointers that contains imaginary parts of subband samples.
pSbrTableWindowDown	Pointer to the window table that is used by decoder SBR Analysis Filter. This parameter is used only for float-point cases.

offset	Desired displacement in number of rows when the matrix *pDst* is calculated; $offset \geq 0$
NumLoop	Parameter equal to 32 if frame size of the core decoded signal is 1024 30 if frame size of the core decoded signal is 960.
kx	First SBR subband in the SBR range ($0 \leq kx < 32$).
pFilterSpec	Pointer to the `FilterSpec_SBR` specification structure.
pWorkBuf	Pointer to the work buffer.

Description

These functions are declared in the `ippac.h` file. The functions filter input signal according to *pFilterSpec* specification parameters.

Float-point case. The function `ippsAnalysisFilter_SBR_RToC_32f32fc_D2L` uses the real input data type and complex output data type, processes the input real array *pSrc* and stores the result in the complex matrix *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function `ippsAnalysisFilterGetSize_SBR_RToC_32f32fc`.
- This function initializes the process using function `ippsAnalysisFilterInit_SBR_RToC_32f32fc`. Deallocate *pInitBuf* after completing initialization.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32fc*` type.

The function `ippsAnalysisFilter_SBR_RToC_32f_D2L` uses the real data type and stores the result separately in *pDstRe* matrix and *pDstIm* matrix.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function `ippsAnalysisFilterGetSize_SBR_RToC_32f`.
- This function initializes the process using function `ippsAnalysisFilterInit_SBR_RToC_32f`. After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32f*` type.

The functionality of `ippsAnalysisFilter_SBR_RToC_32f32fc_D2L` and `ippsAnalysisFilter_SBR_RToC_32f_D2L` is described as follows:

Step 1. **Buffer Updating**

$$x[n] = x[n-32],\ 320 > n \geq 32$$

$$x[n] = pSrc[l \times 32 + 31 - n],\ 31 \geq n \geq 0$$

Step 2. **Polyphase Filtering**

$$u[n] = \sum_{j=0}^{4} x[n + j \times 64] \times pSbrTableWindowDown[n + j \times 64],\ 0 \leq n < 64$$

Step 3. **Special Fourier Transform**

$$pDst[l + offset][k] = 2 \sum_{n=0}^{63} u[n] \times \exp\left[i\frac{\pi}{64} \times \left(k + \frac{1}{2}\right)\left(2n - \frac{1}{2}\right)\right],\ 0 \leq k < 32$$

Step 4. **Clearing Upper Part of Spectrum**

$$pDst[l + offset][k] = 0,\ kx \leq k < 32.$$

Note that Steps 1 through 4 are repeated for $0 \leq l < NumLoop$ and

$$pSbrTableWindowDown[n] = c[2 \times n],\ n = 0, \ldots, 320,$$

where coefficients $c[i]$ are coefficients of the QMF bank window from [ISO14496].

The function `ippsAnalysisFilter_SBR_RToR_32f_D2L` uses the real data type, processes the input real array *pSrc* and stores the result in the real matrix *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function `ippsAnalysisFilterGetSize_SBR_RToR_32f`.
- This function initializes the process using function `ippsAnalysisFilterInit_SBR_RToR_32f`. After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_R_32f*` type.

The functionality of `ippsAnalysisFilter_SBR_RToR_32f_D2L` can be described in the same way as the above `AnalysisFilter_SBR_D2L` functions but Step 3 changes as follows:

Step 3. **Special Fourier Transform**

$$pDst[l + offset][k] = 2 \sum_{n=0}^{63} u[n] \times \cos\left[\frac{\pi}{64} \times \left(k + \frac{1}{2}\right)(2n - 96)\right],\ 0 \leq k < 32.$$

The functions `ippsAnalysisFilter_SBR_RToC_32f_D2L` and `ippsAnalysisFilter_SBR_RToR_32f_D2L` are used in the float-point version of AAC decoder included into IPP Samples. See introduction to this section.

Fixed-point case. The functions with integer values (`_32s32sc` and `_32s`) do not saturate any intermediate or output data so the user should watch closely for overload.

In `ippsAnalysisFilter_SBRHQ_32s32sc` and `ippsAnalysisFilter_SBRLP_32s` the source and destination have different positions of decimation point (Q format). If *pSrc* has Q(N) format, *pDst* has Q(N-6) format. The recommended Q of the source is Q12.

The functionality of `ippsAnalysisFilter_SBRHQ_32s32sc` is described as follows:

Step 1. **Buffer Updating**

$$x[n] = x[n-32], \quad 32 \le n < 320$$

$$x[n] = pSrc[31-n], \quad 0 \le n < 32$$

Step 2. **Polyphase Filtering**

$$u[n] = \sum_{j=0}^{4} x[n+j\times 64] \times pWin[n+j\times 64], \quad 0 \le n < 64$$

Step 3. **Special Fourier Transform**

$$pDst[k] = 2\sum_{n=0}^{63} u[n] \times \exp\left[\frac{\pi i}{64} \times \left(k+\frac{1}{2}\right)\left(2n-\frac{1}{2}\right)\right], \quad 0 \le k < 32$$

Step 4. **Clearing Upper Part of Spectrum**

$$pDst[k] = 0, \quad kx \le k < 32.$$

The functionality of `ippsAnalysisFilter_SBRLP_32s` can be described in the same way as the above function but Step 3 changes as follows:

Step 3. **Special Fourier Transform**

$$pDst[k] = \sum_{n=0}^{63} u[n] \times \cos\left[\frac{\pi i}{64}\left(k+\frac{1}{2}\right)(2n-96)\right], \quad 0 \le k < 32.$$

Note that $pWin[n] = c[2\times n]$ and $c[i]$ coefficients are coefficients of the QMF bank window from [ISO14496A].

The functions `ippsAnalysisFilter_SBRHQ_32s32sc` and `ippsAnalysisFilter_SBRLP_32s` are used in the fixed-point version of AAC decoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the identifier *pFilterSpec* is incorrect.
`ippStsSizeErr`	Indicates an error when *kx* is less than 0 or greater than 32.

SynthesisFilter_SBR

Transforms SBR-processed subband signals into time domain samples.

Syntax

Case 1: Operation on float-point signal.

```
IppStatus ippsSynthesisFilter_SBR_CToR_32fc32f_D2L (const Ipp32fc* pSrc[],
    Ipp32f* pDst, const Ipp32f* pSbrTableWindow, int NumLoop, const
    IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);
IppStatus ippsSynthesisFilter_SBR_CToR_32f_D2L(const Ipp32f* pSrcRe[], const
    Ipp32f* pSrcIm[], Ipp32f* pDst, const Ipp32f* pSbrTableWindow, int NumLoop,
    const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u* pWorkBuf);
IppStatus ippsSynthesisFilter_SBR_RToR_32f_D2L(const Ipp32f* pSrc[], Ipp32f*
    pDst, const Ipp32f* pSbrTableWindow, int NumLoop, const
    IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisFilter_SBRHQ_32sc32s(const Ipp32sc* pSrc, Ipp32s* pDst,
    const IppsFilterSpec_SBR_C_32sc* pFilterSpec);
IppStatus ippsSynthesisFilter_SBRLP_32s(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFilterSpec_SBR_R_32s* pFilterSpec);
```

Parameters

pSrc[]	Array of pointers, holds SBR-processed subband signals. Note, that for functions with integer values _32sc32s and _32s this parameter is the *pSrc* pointer to the input vector.
pSrcRe[]	Array of pointers that contains real parts of SBR-processed subband signals.
pSrcIm[]	Array of pointers that contains imaginary parts of SBR-processed subband signals.
pDst	Pointer to the output vector, holds time domain output samples.
pSbrTableWindow	Pointer to the window table that is used by SBR Synthesis Filter. This parameter is used only for float-point cases.
NumLoop	Parameter equal to 32 if frame size of the core decoded signal is 1024 30 if frame size of the core decoded signal is 960.
pFilterSpec	Pointer to the FilterSpec_SBR specification structure.
pWorkBuf	Pointer to the work buffer used by the filter.

Description

These functions are declared in the ippac.h file. The functions transform SBR-processed subband signals into time domain samples according to *pFilterSpec* specification parameters.

Float-point case. The function ippsSynthesisFilter_SBR_CToR_32fc32f_D2L uses the complex input data type and real output data type, processes the input complex array of pointers *pSrc* and stores the result in the real vector *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function ippsSynthesisFilterGetSize_SBR_CToR_32fc32f.
- This function initializes the process using function ippsSynthesisFilterInit_SBR_CToR_32fc32f. Deallocate *pInitBuf* after completing initialization.
- *pFilterSpec* specification parameter has to be of IppsFilterSpec_SBR_C_32fc* type.

The function ippsSynthesisFilter_SBR_CToR_32f_D2L uses the real data type and stores the result separately in *pSrcRe* matrix and *pSrcIm* matrix.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function ippsSynthesisFilterGetSize_SBR_CToR_32f.

- This function initializes the process using function ippsSynthesisFilterInit_SBR_CToR_32f. After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of IppsFilterSpec_SBR_C_32f* type.

The functionality of ippsSynthesisFilter_SBR_CToR_32fc32f_D2L and ippsSynthesisFilter_SBR_CToR_32f_D2L is described as follows:

Step 1. **Buffer Updating**

$$v[n] = v[n-128], \quad 1280 > n \geq 128$$

Step 2. **Special Fourier Transform**

$$v[n] = \frac{1}{64}\sum_{k=0}^{63}\mathrm{Re}\left\{pSrc[l][k] \times \exp\left[i\frac{\pi}{128} \times \left(k+\frac{1}{2}\right)(2n-255)\right]\right\}, \quad 0 \leq n < 128$$

Step 3. **Polyphase Filtering**

$$\begin{cases} w[128 \times n + k] = v[256 \times n + k] \times pSbrTableWindow[128 \times n + k] \\ 0 \leq n < 5 \\ 0 \leq k < 64 \\ w[128 \times n + 64 + k] = v[256 \times n + 192 + k] \times pSbrTableWindow[128 \times n + 64 + k] \end{cases},$$

Step 4. **Output Vector Updating**

$$pDst[64 \times l + k] = \sum_{n=0}^{9} w[64 \times n + k], \quad 0 \leq k < 64.$$

Note that Steps 1 through 4 are repeated for $0 \leq l < NumLoop$ and $pSbrTableWindow[n] = c[n], \; n = 0, \ldots, 640$,

where coefficients $c[i]$ are coefficients of the QMF bank window from [ISO14496].

The function ippsSynthesisFilter_SBR_RToR_32f_D2L uses the real data type, processes the input real matrix *pSrc* and stores the result in the real array *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function ippsSynthesisFilterGetSize_SBR_RToR_32f.
- This function initializes the process using function ippsSynthesisFilterInit_SBR_RToR_32f. After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of IppsFilterSpec_SBR_R_32f* type.

The functionality of `ippsSynthesisFilter_SBR_RToR_32f` can be described in the same way as the above `SynthesisFilter_SBR` functions but Step 2 changes as follows:

Step 2. **Special Fourier Transform**

$$v[n] = \frac{1}{32}\sum_{k=0}^{63} pSrc[l][k] \times \cos\left[\frac{\pi}{128} \times \left(k + \frac{1}{2}\right)(2n - 64)\right],\ 0 \le n < 128 .$$

The functions `ippsSynthesisFilter_SBR_CToR_32f_D2L` and `ippsSynthesisFilter_SBR_RToR_32f_D2L` are used in the float-point version of AAC decoder included into IPP Samples. See introduction to this section.

Fixed-point case. The functions with integer values (`_32sc32s` and `_32s`) do not saturate any intermediate or output data so the user should watch closely for overload.

In `ippsSynthesisFilter_SBRHQ_32sc32s` and `ippsSynthesisFilter_SBRLP_32s` the source and destination have different positions of decimation point (Q format). If *pSrc* has Q(N) format, *pDst* has Q(N-6) format. The recommended Q of the source is Q12.

The functionality of `ippsSynthesisFilter_SBRHQ_32sc32s` is described as follows:

Step 1. **Buffer Updating**

$$v[n] = v[n - 128],\ 128 \le n < 1280$$

Step 2. **Special Fourier Transform**

$$v[n] = \sum_{k=0}^{63} \mathrm{Re}\left\{pSrc[k] \times \exp\left[\frac{\pi i}{128} \times \left(k + \frac{1}{2}\right)(2n - 255)\right]\right\},\ 0 \le n < 128$$

Step 3. **Polyphase Filtering**

$$\begin{aligned} w[128 \times n + k] &= v[256 \times n + k] \times pWin[128 \times n + k] \\ w[128 \times n + 64 + k] &= v[256 \times n + 192 + k] \times pWin[128 \times n + 64 + k] \end{aligned} ,\quad \begin{cases} 0 \le n < 5 \\ 0 \le k < 64 \end{cases}$$

Step 4. **Output Vector Updating**

$$pDst[k] = \sum_{n=0}^{9} w[64 \times n + k],\ 0 \le k < 64 ,$$

where $pWin[n] = c[n]$ and $c[n]$ coefficients are coefficients of the QMF bank window from [ISO14496A].

The functionality of `ippsSynthesisFilter_SBRLP_32s` can be described in the same way as the above function but Step 2 changes as follows:

Step 2. **Special Fourier Transform**

$$v[n] = \sum_{k=0}^{63} pSrc[k] \times \cos\left[\frac{\pi i}{128} \times \left(k + \frac{1}{2}\right)(2n - 64)\right], \ 0 \le n < 128 .$$

The functions `ippsSynthesisFilter_SBRHQ_32sc32s` and `ippsSynthesisFilter_SBRLP_32s` are used in the fixed-point version of AAC decoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the identifier *pFilterSpec* is incorrect.

SynthesisDownFilter_SBR

Transforms SBR-processed subband signals into time domain samples and performs downsampling at the same time.

Syntax

Case 1: Operation on float-point signal.

```
IppStatus ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L (const Ipp32fc* pSrc[],
    Ipp32f* pDst, const Ipp32f* pSbrTableWindowDown, const int NumLoop, const
    IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);
IppStatus ippsSynthesisDownFilter_SBR_CToR_32f_D2L(const Ipp32f* pSrcRe[],
    const Ipp32f* pSrcIm[], Ipp32f* pDst, const Ipp32f* pSbrTableWindow, const
    int NumLoop, const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

```
IppStatus ippsSynthesisDownFilter_SBR_RToR_32f_D2L(const Ipp32f* pSrc[],
    Ipp32f* pDst, const Ipp32f* pSbrTableWindow, const int NumLoop, const
    IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisDownFilter_SBRHQ_32sc32s(const Ipp32sc* pSrc, Ipp32s*
    pDst, const IppsFilterSpec_SBR_C_32sc* pFilterSpec);
IppStatus ippsSynthesisDownFilter_SBRLP_32s(const Ipp32s* pSrc, Ipp32s* pDst,
    const IppsFilterSpec_SBR_R_32s* pFilterSpec);
```

Parameters

pSrc[]	Array of pointers, holds SBR-processed subband signals. Note, that for functions with integer values `_32sc32s` and `_32s` this parameter is the *pSrc* pointer to the input vector.
pSrcRe[]	Array of pointers that contains real parts of SBR-processed subband signals.
pSrcIm[]	Array of pointers that contains imaginary parts of SBR-processed subband signals.
pDst	Pointer to the output vector, holds time domain output samples.
pSbrTableWindowDown	Pointer to the window table that is used by SBR SynthesisDown Filter. This parameter is used only for float-point cases.
NumLoop	Parameter equal to 32 if frame size of the core decoded signal is 1024 30 if frame size of the core decoded signal is 960.
pFilterSpec	Pointer to the `FilterSpec_SBR` specification structure.
pWorkBuf	Pointer to the work buffer used by the filter.

Description

These functions are declared in the `ippac.h` file. The functions transform SBR-processed subband signals into time domain samples and perform downsampling at the same time according to *pFilterSpec* specification parameters.

Float-point case. The function `ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L` uses the complex input data type and real output data type, processes the input complex matrix *pSrc* and stores the result in the real array *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function `ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f`.

- This function initializes the process using function ippsSynthesisDownFilterInit_SBR_CToR_32fc32f. Deallocate *pInitBuf* after completing initialization.
- *pFilterSpec* specification parameter has to be of IppsFilterSpec_SBR_C_32fc* type.

The function ippsSynthesisDownFilter_SBR_CToR_32f_D2L uses the real data type and stores the result separately in *pSrcRe* matrix and *pSrcIm* matrix.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function ippsSynthesisDownFilterGetSize_SBR_CToR_32f.
- This function initializes the process using function ippsSynthesisDownFilterInit_SBR_CToR_32f. After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of IppsFilterSpec_SBR_C_32f* type.

The functionality of ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L and ippsSynthesisDownFilter_SBR_CToR_32f_D2L is described as follows:

Step 1. **Buffer Updating**

$$v[n] = v[n-64], \quad 640 > n \geq 64$$

Step 2. **Special Fourier Transform**

$$v[n] = \frac{1}{64}\sum_{k=0}^{31}\mathrm{Re}\left\{pSrc[l][k] \times \exp\left[i\frac{\pi}{64} \times \left(k+\frac{1}{2}\right)(2n-127)\right]\right\}, \quad 0 \leq n < 64$$

Step 3. **Polyphase Filtering**

$$\begin{aligned} w[64 \times n + k] &= v[128 \times n + k] \times pSbrTableWindow[128 \times n + k] \\ w[64 \times n + 32 + k] &= v[128 \times n + 96 + k] \times pSbrTableWindow[128 \times n + 96 + k] \end{aligned}, \quad \begin{cases} 0 \leq n < 5 \\ 0 \leq k < 32 \end{cases}$$

Step 4. **Output Vector Updating**

$$pDst[32 \times l + k] = \sum_{n=0}^{9} w[32 \times n + k], \quad 0 \leq k < 32.$$

Note that Steps 1 through 4 are repeated for $0 \leq l < NumLoop$ and

$$pSbrTableWindowDown[n] = c[2 \times n], \quad n = 0, \ldots, 320,$$

where coefficients $c[i]$ are coefficients of the QMF bank window from [ISO14496].

The function ippsSynthesisDownFilter_SBR_RToR_32f_D2L uses the real data type, processes the input real matrix *pSrc* and stores the result in the real array *pDst*.

- This function gets sizes of the work buffer, init buffer, and specification structure using the function ippsSynthesisDownFilterGetSize_SBR_RToR_32f.
- This function initializes the process using function ippsSynthesisDownFilterInit_SBR_RToR_32f. After completing initialization *pInitBuf* can be deallocated.
- *pFilterSpec* specification parameter has to be of IppsFilterSpec_SBR_R_32f* type.

The functionality of ippsSynthesisDownFilter_SBR_RToR_32f_D2L can be described in the same way as the above SynthesisFilter_SBR functions but Step 2 changes as follows:

Step 2. **Special Fourier Transform**

$$v[n] = \frac{1}{32}\sum_{k=0}^{31} pSrc[l][k] \times \cos\left[\frac{\pi}{64} \times \left(k+\frac{1}{2}\right)(2n-32)\right],\ 0 \le n < 64 .$$

The functions ippsSynthesisDownFilter_SBR_CToR_32f_D2L and ippsSynthesisDownFilter_SBR_RToR_32f_D2L are used in the float-point version of AAC decoder included into IPP Samples. See introduction to this section.

Fixed-point case. The functions with integer values (_32sc32s and _32s) do not saturate any intermediate or output data so the user should watch closely for overload.

In ippsSynthesisDownFilter_SBRHQ_32sc32s and ippsSynthesisDownFilter_SBRLP_32s the source and destination have different positions of decimation point (Q format). If *pSrc* has Q(N) format, *pDst* has Q(N-6) format. The recommended Q of the source is Q12.

The functionality of ippsSynthesisDownFilter_SBRHQ_32sc32s is described as follows:

Step 1. **Buffer Updating**

$$v[n] = v[n-64],\ 64 \le n < 640$$

Step 2. **Special Fourier Transform**

$$v[n] = \sum_{k=0}^{31} \mathrm{Re}\left\{pSrc[k] \times \exp\left[\frac{\pi i}{128} \times \left(k+\frac{1}{2}\right)(2n-127.5)\right]\right\},\ 0 \le n < 64$$

Step 3. **Polyphase Filtering**

$$\begin{aligned} w[64\times n+k] &= v[128\times n+k]\times pWin[64\times n+k] \\ w[64\times n+32+k] &= v[128\times n+96+k]\times pWin[64\times n+32+k] \end{aligned}, \quad \begin{cases} 0\le n<5 \\ 0\le k<32 \end{cases}$$

Step 4. **Output Vector Updating**

$$pDst[k] = \sum_{n=0}^{9} w[32\times n+k],\ 0\le k<32,$$

where $pWin[n] = c[2\times n]$ and $c[n]$ coefficients are coefficients of the QMF bank window from [ISO14496A].

The functionality of `ippsSynthesisDownFilter_SBRLP_32s` can be described in the same way as the above function but Step 2 changes as follows:

Step 2. **Special Fourier Transform**

$$v[n] = \sum_{k=0}^{31} pSrc[k]\times \cos\left[\frac{\pi i}{128}\times\left(k+\frac{1}{2}\right)(2n-32)\right],\ 0\le n<64.$$

The functions `ippsSynthesisDownFilter_SBRHQ_32sc32s` and `ippsSynthesisDownFilter_SBRLP_32s` are used in the fixed-point version of AAC decoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the identifier *pFilterSpec* is incorrect.

Example of Filter Usage

Before calling `ippsAnalysis/Synthesis/SynthesisDown`, initialize the filter specification structure by calling corresponding functions.

Example 10-14 demonstrates Analysis and SynthesisDown filters in use. This example illustrates the typical filter usage and is applicable to all QMF functions.

Example 10-14Analysis and SynthesisDown Filters Usage

```
int filter_SBR_sample( void )
```

Example 10-14Analysis and SynthesisDown Filters Usage (continued)

```
{
  IppStatus status = ippStsNoErr;
  int sizeSpec[2], sizeWorkBuf[2], sizeInitBuf[2];
  /*********************************************************************/
  /* [0] - index for analysisFilter, [1] - index for synthesisDownFilter  */
  /*********************************************************************/

  IppsFilterSpec_SBR_R_32f* pFilterSpec[2];

  Ipp8u* pMemSpec;
  Ipp8u* pInitBuf;
  Ipp8u* pWorkBuf;

  Ipp32f bufSrc[1024];
  Ipp32f bufDst[1024];

  Ipp32f bufMem[2*1024];
  Ipp32f* pSbr[32];

  Ipp32f* pWinTab = SBR_TABLE_QMF_WINDOW_320;
  /*********************************************************************/
  /*  pWinTab[i] = c[2*i],                                             */
  /*  where coefficients c[i] are coefficients of the QMF bank window  */
  /*  from [ISO14496].                                                 */
  /*********************************************************************/

  int delay = 289; // delay of signal after analysis and synthesis

  int i;

  int nBand = 32;// numLoop
  int offset = 0;
  int kx = 32;  // wo SBR range

  /* init process  */
  ippsZero_32f(bufSrc, 1024);
  ippsZero_32f(bufDst, 1024);
  ippsZero_32f(bufMem, 2*1024);

  status = ippsAnalysisFilterGetSize_SBR_RToR_32f(&sizeSpec[0],
           &sizeInitBuf[0], &sizeWorkBuf[0]);

  status = ippsSynthesisDownFilterGetSize_SBR_RToR_32f(&sizeSpec[1],
           &sizeInitBuf[1], &sizeWorkBuf[1]);

  sizeInitBuf[0] = MAX(sizeInitBuf[0], sizeInitBuf[1]);

  sizeWorkBuf[0] = MAX(sizeWorkBuf[0], sizeWorkBuf[1]);

  sizeWorkBuf[0] = MAX(sizeWorkBuf[0], 10);
```

Example 10-14Analysis and SynthesisDown Filters Usage (continued)

```
  pMemSpec = ippsMalloc_8u(sizeSpec[0] + sizeSpec[1]);

  pWorkBuf = ippsMalloc_8u(sizeWorkBuf[0]);

  pInitBuf = ippsMalloc_8u(sizeInitBuf[0]);

  status = ippsAnalysisFilterInit_SBR_RToR_32f(&pFilterSpec[0], pMemSpec,
           pInitBuf);

  status = ippsSynthesisDownFilterInit_SBR_RToR_32f(&pFilterSpec[1], pMemSpec
           + sizeSpec[0], pInitBuf);

  /* generation data */
  srand( 0 );

  for(i=0; i<1024; i++) bufSrc[i] = rand() / 1000.f;

  for(i=0; i<32; i++) pSbr[i] = bufMem + 64*i;

  /* main process */
  status = ippsAnalysisFilter_SBR_RToR_32f_D2L(bufSrc, pSbr, pWinTab, nBand,
           offset, kx, pFilterSpec[0], pWorkBuf);

  status = ippsSynthesisDownFilter_SBR_RToR_32f_D2L(pSbr, bufDst, pWinTab,
           nBand, pFilterSpec[1], pWorkBuf);

  for(i=0; i<734; i++){
    printf("orig = %15.10f, dst = %15.10f, diff = %15.10f\n", bufSrc[i],
    bufDst[delay+i], bufSrc[i] - bufDst[delay+i] );
  }

  /* free memory */
  ippsFree(pMemSpec);
  ippsFree(pWorkBuf);
  ippsFree(pInitBuf);

  return 0;
}
```

Prediction Functions

SBR prediction functions provide inverse filtering for correct reconstruction of high frequency transmitted signals.

PredictCoef_SBR

Obtains prediction filter coefficients using covariance method.

Syntax

```
IppStatus ippsPredictCoef_SBR_C_32fc_D2L(const Ipp32fc* pSrc[], Ipp32fc*
    pAlpha0, Ipp32fc* pAlpha1, int k0, int len);
IppStatus ippsPredictCoef_SBR_C_32f_D2L(const Ipp32f* pSrcRe[], const Ipp32f*
    pSrcIm[], Ipp32f* pAlpha0Re, Ipp32f* pAlpha0Im, Ipp32f* pAlpha1Re, Ipp32f*
    pAlpha1Im, int k0, int len);
IppStatus ippsPredictCoef_SBR_R_32f_D2L(const Ipp32f* pSrc[], Ipp32f* pAlpha0,
    Ipp32f* pAlpha1, int k0, int len);
```

Parameters

pSrc[]	Array of pointers that holds input real or complex matrix ([40][32]), holds Low Frequency QMF-processed subband signals.
pSrcRe[]	Array of pointers ([40][32]) that contains real parts of Low Frequency SBR-processed subband signals.
pSrcIm[]	Array of pointers ([40][32]) that contains imaginary parts of Low Frequency SBR-processed subband signals.
pAlpha0, pAlpha1	Prediction filter coefficients used by High Frequency filter.
pAlpha0Re, pAlpha1Re	Pointers to the arrays that contain real parts of the corresponding complex filter coefficients.
pAlpha0Im, pAlpha1Im	Pointers to the arrays that contain imaginary parts of the corresponding complex filter coefficients.
k0	First QMF subband in the `f_master` table.
len	Size autocorrelation.

Description

These functions are declared in the `ippac.h` file. The function `PredictCoef_SBR` obtains prediction filter coefficients using covariance method.

The function `ippsPredictCoef_SBR_C_32fc` uses the complex input data type and complex output data type, processes the input complex matrix *pSrc* and stores the result in complex arrays *pAlpha0* and *pAlpha1*.

The function `ippsPredictCoef_SBR_C_32f_D2L` uses the real input and output data type, uses complex input data separately in matrix *pSrcRe* and matrix *pSrcIm* and stores complex output data separately in *pAlpha0Re*, *pAlpha0Im*, *pAlpha1Re*, *pAlpha1Im*.

The functions `ippsPredictCoef_SBR_C_32fc_D2L` and `ippsPredictCoef_SBR_C_32f_D2L` are used in high quality SBR mode.

The function `ippsPredictCoef_SBR_R_32f_D2L` uses the real input and output data type. This function is used in low power SBR mode.

The prediction filter coefficients are obtained with the covariance method using the following formula:

$$\phi_k(i,\ j) = \sum_{n=0}^{len-1} pSrc[n-i+2][k] \cdot conj(pSrc[n-j+2][k]),$$

$0 \le i < 3,\ 1 \le j < 3,\ 0 \le k < k_0$.

The coefficients *pAlpha0*(*k*) and *pAlpha1*(*k*) are calculated as follows:

$$d(k) = \phi_k(2,\ 2) \cdot \phi_k(1,\ 1) - \frac{1}{1+\varepsilon_{inv}} \left|\phi_k(1,\ 2)\right|^2,$$

$$pAlpha1(k) = \begin{cases} \dfrac{\phi_k(0,\ 1) \times \phi_k(1,\ 2) - \phi_k(0,\ 2) \times \phi_k(1,\ 1)}{d(k)}, & d(k) \ne 0 \\ 0, & d(k) = 0 \end{cases}$$

$$pAlpha0(k) = \begin{cases} \dfrac{\phi_k(0,\ 1) \times pAlpha1(k) \times conj(\phi_k(1,2))}{\phi_k(1,1)}, & \phi_k(1,1) \ne 0 \\ 0, & \phi_k(1,1) = 0 \end{cases}$$

The functions `ippsPredictCoef_SBR_C_32f_D2L` and `ippsPredictCoef_SBR_R_32f_D2L` are used in the float-point version of AAC decoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

PredictOneCoef_SBR

Obtains a prediction filter coefficient using covariance method.

Syntax

```
IppStatus ippsPredictOneCoef_SBRHQ_32sc_D2L(const Ipp32sc* pSrc[], Ipp32sc*
    pAlpha0, Ipp32sc* pAlpha1, int k, int lenCorr);
IppStatus ippsPredictOneCoef_SBRLP_32s_D2L(const Ipp32s* pSrc[], Ipp32s*
    pAlpha0, Ipp32s* pAlpha1, Ipp32s* pRefCoef, int k, int lenCorr, int flag);
```

Parameters

pSrc[]	Array of pointers that holds input real or complex matrix, holds Low Frequency QMF-processed subband signals.
pAlpha0, pAlpha1	Prediction filter coefficients used by High Frequency filter.
pRefCoef	Reflection coefficient that is used by the aliasing detection algorithm (LP mode only).
k	Number of the input matrix column. This parameter is used for calculating prediction coefficients.
lenCorr	Size autocorrelation. This parameter is used for calculating prediction coefficients.
flag	Flag that specifies the result of calculation (LP mode only). If *flag* is NULL (0), then only the reflection coefficient is calculated. Otherwise both prediction and reflection coefficients are calculated.

Description

These functions are declared in the ippac.h file and obtain a prediction filter coefficient using covariance method. The ippsPredictOneCoef_SBRHQ_32sc_D2L function is used in high quality SBR mode and the ippsPredictOneCoef_SBRLP_32s_D2L function is used in low power SBR mode.

Prediction and reflection coefficients have the same position of decimation point (Q format), independent of Q format of *pSrc*. It is Q(29). Q format of *pSrc* can be arbitrary but the recommended value is Q(5).

The prediction filter coefficient is obtained with the covariance method using the following formula:

$$\phi(i,\ j) = \sum_{n=0}^{lenCorr-1} pSrc[n-i+2][k] \cdot conj(pSrc[n-j+2][k]),$$

$$d = \phi(2,\ 2) \cdot \phi(1,\ 1) - \frac{1}{1+\varepsilon_{inv}} |\phi(1,\ 2)|^2,$$

$$pAlpha1[0] = \begin{cases} \dfrac{\phi(0,\ 1) \times \phi(1,\ 2) - \phi(0,\ 2) \times \phi(1,\ 1)}{d}, & d \neq 0 \\ 0, & d = 0 \end{cases}$$

$$pAlpha0[0] = \begin{cases} \dfrac{\phi(0,\ 1) \times pAlpha[0] \times conj(\phi(1,1))}{\phi(1,1)}, & \phi(1,1) \neq 0 \\ 0, & (\ \ \phi)(1,1) = 0 \end{cases}$$

$$pRefCoef[0] = \begin{cases} \min(\max(-\dfrac{\phi(0,\ 1)}{\phi(1,1)}, -1), 1), & \phi(1,1) \neq 0 \\ 0, & \phi(1,1) = 0 \end{cases}$$

$$\begin{cases} pAlpha0[0] = 0 \\ pAlpha1[0] = 0 \end{cases}, \quad \text{if } |pAlpha0[0]| + |pAlpha1[0]| \geq 4$$

The functions `ippsPredictOneCoef_SBRHQ_32sc_D2L` and `ippsPredictOneCoef_SBRLP_32s_D2L` are used in the fixed-point version of AAC decoder included into IPP Samples. See introduction to this section.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.

DTS Audio Coding Functions

DTS (Digital Theater Systems) is a multi-channel digital surround sound format, also known as DTS Coherent Acoustics. DTS Coherent Acoustics standard is published by the ETSI (ETSI 102 114 v1.2.1) [ETSI02].

The core DTS specification supports up to 6 audio channels (5.1 configuration) with the sampling frequency between 8kHz and 48kHz. Further specification extensions allow for an additional 2 channels and/or additional high frequency data to be included. Audio is encoded by splitting it into 32 subbands, which are then encoded using ADPCM.

Table lists the DTS macro and constant definitions.

Table 10-9 DTS Macro and Constant Definitions

Global Macro Name	Definition	Notes
IPPAC_DTS_NONPERFECT	0	nonperfect reconstruction filter bank
IPPAC_DTS_PERFECT	1	perfect reconstruction filter bank

DTS Audio Decoder

This section describes Intel IPP functions for DTS decoding operations.

SynthesisFilterInit_DTS

Initializes DTS synthesis filter specification structure.

Syntax

```
IppStatus ippsSynthesisFilterInit_DTS_32f(IppsFilterSpec_DTS_32f* pFilterSpec);
```

Parameters

pFilterSpec	Pointer to the IppsFilterSpec_DTS_32f specification structure. Only the main function uses this structure.

Description

This function is declared in the `ippac.h` file. The function creates and initializes a DTS synthesis filter specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the specified pointer is `NULL`.

SynthesisFilterInitAlloc_DTS

Allocates memory for DTS synthesis filter specification structure and initilaizes it.

Syntax

```
IppStatus ippsSynthesisFilterInitAlloc_DTS_32f(IppsFilterSpec_DTS_32f**
    ppFilterSpec);
```

Parameters

ppFilterSpec	Pointer to the pointer to the `IppsFilterSpec_DTS_32f` specification structure. Only the main function uses this structure.

Description

This function is declared in the `ippac.h` file. The function allocates memory for a DTS synthesis filter specification structure, creates, and initializes this structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the specified pointer is `NULL`.

SynthesisFilterGetSize_DTS

Returns size of DTS synthesis filter specification structure.

Syntax

```
IppStatus ippsSynthesisFilterGetSize_DTS_32f(int* pSizeSpec);
```

Parameters

pSizeSpec	Address of size value of the `IppsFilterSpec_DTS_32f` specification structure.

Description

This function is declared in the `ippac.h` file. The function gets size of the DTS synthesis filter specification structure and stores the result in *pSizeSpec*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the specified pointer is `NULL`.

SynthesisFilterFree_DTS

Frees memory allocated for DTS synthesis filter specification structure.

Syntax

```
IppStatus ippsSynthesisFilterFree_DTS_32f(IppsFilterSpec_DTS_32f* pFilterSpec);
```

Parameters

pFilterSpec	Pointer to the `IppsFilterSpec_DTS_32f` specification structure.

Description

This function is declared in the `ippac.h` file. The function frees memory that `ippsSynthesisFilterInitAlloc_DTS_32f` allocates for the DTS synthesis filter specification structure.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the specified pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the *`pFilterSpec`* identifier is incorrect.

SynthesisFilter_DTS

Transforms QMF DTS-processed subband signals into time domain samples.

Syntax

```
IppStatus ippsSynthesisFilter_DTS_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    filtType, const IppsFilterSpec_DTS_32f* pFilterSpec);
```

Parameters

`pSrc`	Array of pointers, holds QMF DTS-processed subband signals.
`pDst`	Pointer to the output vector, holds time domain output samples.
`filtType`	Flag that indicates the filter bank: `IPPAC_DTS_PERFECT` - indicates perfect reconstruction filter bank, `IPPAC_DTS_NONPERFECT` - indicates nonperfect reconstruction filter bank.
`pFilterSpec`	Pointer to `IppsFilterSpec_DTS_32f` specification structure.

Description

This function is declared in the `ippac.h` file. The function transforms QMF DTS-processed subband signals into time domain samples. For each input block (32 subband samples), this function generates an output sequence of 32 PCM samples in the vector pointed to by

pDst. The *filtType* flag indicates the filter bank to be used: IPPAC_DTS_PERFECT indicates perfect reconstruction filter bank, IPPAC_DTS_NONPERFECT - nonperfect reconstruction filter bank.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsBadArgErr	Indicates an error when *filtType* is neither IPPAC_DTS_PERFECT nor IPPAC_DTS_NONPERFECT.
ippStsContextMatchErr	Indicates an error when the *pFilterSpec* identifier is incorrect.

String Functions

11

This chapter This chapter describes Intel® IPP functions that perform operations with a text. First part describes the functions for simple string manupulation. Second part contains functions that perform more sophisticated matching operation using patterns of the regular expressions.

The full list of functions in this group is given in Table 11-1.

Table 11-1 Intel IPP String Functions

Function Base Name	Operation
String Manipulation Functions	
Find, FindRev	Looks for the first occurrence of the substring matching the specified string.
FindC, FindRevC	Looks for the first occurrence of the specified element within the source string.
FindCAny, FindRevCAny	Looks for the first occurrence of any element of the specified array within the source string.
Insert	Inserts a string into another string.
Remove	Removes a specified number of elements from the string.
Compare	Compares two strings of the fixed length.
CompareIgnoreCase, CompareIgnoreCaseLatin	Compares two strings of the fixed length ignoring case.
Equal	Compares two strings of the fixed length for equality.
TrimC	Deletes all occurrences of a specified symbol both in the beginning and in the end of the string.
TrimCAny	Deletes all occurrences of any of the specified symbols both in the beginning and in the end of the source string.

Table 11-1 Intel IPP String Functions

Function Base Name	Operation
TrimStartCAny TrimEndCAny	Deletes all occurrences of any of the specified symbols either in the beginning or in the end of the source string, respectively.
ReplaceC	Replaces all occurrences of a specified element in the source string with another element.
Uppercase, UppercaseLatin	Converts alphabetic characters of a string to all uppercase symbols.
Lowercase, LowercaseLatin	Converts alphabetic characters of a string to all lowercase symbols.
Hash	Calculates a hash value for the string.
Concat	Concatenates several strings together.
ConcatC	Concatenates several strings together and inserts symbol delimiters between them.
SplitC	Splits source string into separate parts.
Functions for Work with Regural Expressions	
RegExpInitAlloc	Allocates memory and initializes the structure for processing matching operation with regular expressions..
RegExpFree	Frees the memory allocated for a regular expression state structure..
RegExpInit	Initializes the structure for processing matching operation with regular expressions.
RegExpGetSize	Computes the size of the regular expression state structure..
RegExpSetMatchLimit	Sets the value of the *matchLimit* parameter.
RegExpFind	Looks for the occurrences of the substrings matching the specified regular expression.

String Manipulation

This section decsrtibes the Intel IPP functions that perform operations with strings. Intel IPP string functions for do not consider zero as the end of the string, but require that the length of the string (number of elements) be specified explicitly.

Overlapping of the strings is not supported (for not in-place operations).

Intel IPP string functions operate with two data types, `Ipp8u` and `Ipp16u`.

Find, FindRev

Looks for the first occurrence of the substring matching the specified string.

Syntax

```
IppStatus ippsFind_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind, int
    lenFind, int* pIndex);
IppStatus ippsFind_16u(const Ipp16u* pSrc, int len, const Ipp16u* pFind, int
    lenFind, int* pIndex);
IppStatus ippsFindRev_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind, int
    lenFind, int* pIndex);
IppStatus ippsFindRev_16u(const Ipp16u* pSrc, int len, const Ipp16u* pFind, int
    lenFind, int* pIndex);
```

Parameters

pSrc	Pointer to the source string.
len	Number of elements in the source string.
pFind	Pointer to the reference string.
lenFind	Number of elements in the reference string.
pIndex	Pointer to the result index.

Description

The functions `ippsFind` and `ippsFindRev` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for a substring of elements that match the specified reference string *pFind*. Starting point of the first occurrence of the matching substring is stored in *pIndex*. If no matching substring is found, then *pIndex* is set to `-1`. The function `ippsFindRev` searches the source string in the reverse direction.
The search is case-sensitive.

Example 11-1 shows how to use the function `ippsFind_8u`.

Example 11-1 Using the function ippsFind

```
IppStatus find( void ) {
   IppStatus st;
   int result;
   char str[]    = "matching";
   char string[] = "This function looks for the matching substring";
   char fmt1[]   = "          1         2         3         4         5";
   char fmt2[] =   "0123456789012345678901234567890123456789012345678901234567890";
   printf( "String to be searched:\n\t%s\n", string );
   printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );
   st = ippsFind_8u((Ipp8u*)string, strlen(string), (Ipp8u*)str,
strlen(str), &result);
   printf( "%s found at position %d\n\n", str, result );
   return st;
}

Output:
String to be searched:
   This function looks for the matching substring
             1         2         3         4         5
   0123456789012345678901234567890123456789012345678901234567890

matching found at position 28
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len* or *lenFind* is negative.

FindC, FindRevC

Looks for the first occurrence of the specified element within the source string.

Syntax

```
IppStatus ippsFindC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind, int* pIndex);
IppStatus ippsFindC_16u(const Ipp16u* pSrc, int len, Ipp16u valFind, int*
    pIndex);
IppStatus ippsFindRevC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind, int*
    pIndex);
IppStatus ippsFindRevC_16u(const Ipp16u* pSrc, int len, Ipp16u valFind, int*
    pIndex);
```

Parameters

pSrc	Pointer to the source string.
len	Number of elements in the source string.
valFind	Value of the specified element.
pIndex	Pointer to the result index.

Description

The functions `ippsFindC` and `ippsFindRevC` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for the first occurrence of the specified element with the value *valFind*. The position of this element is stored in *pIndex*. If no matching element is found, then *pIndex* is set to `-1`. The function `ippsFindRev` searches the source string in the reverse direction. The search is case-sensitive.

Example 11-2 shows how to use the function `ippsFindC_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len* is negative.

Example 11-2 Using the Function ippsFindC

```
IppStatus findc( void ) {
   IppStatus st;
   int result;
   char ch = 'r';
   char string[] = "This function looks for the specified character";
   char fmt1[]   = "          1         2         3         4         5";
   char fmt2[] =   "0123456789012345678901234567890123456789012345678901234567890";
   printf( "String to be searched:\n\t%s\n", string );
   printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );
   /* Search forward. */
   st = ippsFindC_8u((Ipp8u*)string, strlen(string), ch, &result);
   printf( "Result:\tfirst %c found at position %d\n\n", ch, result );
   /* Search backward. */
   st = ippsFindRevC_8u((Ipp8u*)string, strlen(string), ch, &result);
   printf( "Result:\tlast %c found at position %d\n\n", ch, result );
   return st;
}

Output:
String to be searched:
        This function looks for the specified character
                  1         2         3         4         5
        0123456789012345678901234567890123456789012345678901234567890

Result: first r found at position 22
Result: last r found at position 46
```

FindCAny, FindRevCAny

Looks for the first occurrence of any element of the specified array within the source string.

Syntax

```
IppStatus ippsFindCAny_8u(const Ipp8u* pSrc, int len, const Ipp8u* pAnyOf, int
    lenAnyOf, int* pIndex);
```

```
IppStatus ippsFindCAny_16u(const Ipp16u* pSrc, int len, const Ipp16u* pAnyOf,
    int lenAnyOf, int* pIndex);
IppStatus ippsFindRevCAny_8u(const Ipp8u* pSrc, int len,
    const Ipp8u* pAnyOf, int lenAnyOf, int* pIndex);
IppStatus ippsFindRevCAny_16u(const Ipp16u* pSrc, int len, const Ipp16u*
    pAnyOf, int lenAnyOf, int* pIndex);
```

Parameters

pSrc	Pointer to the source string.
len	Number of elements in the source string.
pAnyOf	Pointer to the array containing reference elements.
lenAnyOf	Number of elements in the array.
pIndex	Pointer to the result index.

Description

The functions `ippsFindCAny` and `ippsFindRevCAny` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for the first occurrence of any reference element from the specified array *pAnyOf*. The position of this element is stored in *pIndex*. If no matching element is found, then *pIndex* is set to `-1`. The function `ippsFindRevCAny` searches the source string in the reverse direction.
The search is case-sensitive.

Example 11-3 shows how to use the function `ippsFindCAny_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.

ippStsLengthErr	Indicates an error condition if *len* or *lenAnyOf* is negative.

Example 11-3 Using the Function ippsFindCAny

```
IppStatus findcany( void ) {
   IppStatus st;
   char string[] = "xyzabc";
   char AnyOf[] = "bac";
   int  pos;
   st = ippsFindCAny_8u((Ipp8u*)string, strlen(string), (Ipp8u*)AnyOf,
strlen(AnyOf), &pos);
   printf( "First a, b or c in %s is at position %d\n", string, pos );
   return st;
}

Output:
First a, b or c in xyzabc is at position 3
```

Insert

Inserts a string into another string.

Syntax

```
IppStatus ippsInsert_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pInsert,
    int insertLen, Ipp8u* pDst, int startIndex);
IppStatus ippsInsert_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u* pInsert,
    int insertLen, Ipp16u* pDst, int startIndex);
IppStatus ippsInsert_8u_I(const Ipp8u* pInsert, int insertLen, Ipp8u* pSrcDst,
    int* pSrcDstLen, int startIndex);
IppStatus ippsInsert_16u_I(const Ipp16u* pInsert, int insertLen, Ipp16u*
    pSrcDst, int* pSrcDstLen, int startIndex);
```

Parameters

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pInsert	Pointer to the string to be inserted.
insertLen	Number of elements in the string to be inserted.

pDst	Pointer to the destination string.
pSrcDst	Pointer to the source and destination string for in-place operation.
pSrcDstLen	Pointer to the number of elements in the source and destination string for in-place operation.
startIndex	Index of the insertion point.

Description

The function `ippsInsert` is declared in the `ippch.h` file. This function inserts the string *pInsert* containing *insertLen* elements into a source string *pSrc* of length *srcLen*. Insertion position is specified by *startIndex*. The result is stored in the *pDst*.

The in-place flavors of `ippsInsert` insert the string *pInsert* in the source string *pSrcDst* and store the result in the destination string *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if one of the *srcLen*, *insertLen*, *pSrcSdtLen*, *startIndex* is negative, or *startIndex* is greater than *srcLen* or *pSrcDstLen*.

Remove

Removes a specified number of elements from the string.

Syntax

```
IppStatus ippsRemove_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int
    startIndex, int len);
IppStatus ippsRemove_16u(const Ipp16u* pSrc, int srcLen, Ipp16u* pDst, int
    startIndex, int len);
IppStatus ippsRemove_8u_I(Ipp8u* pSrcDst, int* pSrcDstLen, int startIndex, int
    len);
IppStatus ippsRemove_16u_I(Ipp16u* pSrcDst, int* pSrcDstLen, int startIndex,
    int len);
```

Parameters

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pDst	Pointer to the destination string.
pSrcDst	Pointer to the source and destination string for in-place operation.
pSrcDstLen	Pointer to the number of elements in the source and destination string for in-place operation.
startIndex	Index of the starting point.
len	Number of elements to be removed.

Description

The function `ippsRemove` is declared in the `ippch.h` file. This function removes the *len* elements from a source string *pSrc* of length *srcLen*. Starting position is specified by *startIndex*. The result is stored in the *pDst*.

The in-place flavors of `ippsRemove` remove the *len* elements from the source string *pSrcDst* and store the result in the destination string *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if one of the *srcLen*, *len*, *pSrcSdtLen*, *startIndex* is negative, or *(startIndex+len)* is greater than *srcLen* or *pSrcDstLen*.

Compare

Compares two strings of the fixed length.

Syntax

```
IppStatus ippsCompare_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len,
    int* pResult);
IppStatus ippsCompare_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, int len,
    int* pResult);
```

Parameters

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Description

The function `ippsCompare` is declared in the `ippch.h` file. This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. The value *pResult* = *pSrc1*[i]-*pSrc2*[i] is computed successively for each i-th element, *i* =0,...*len*-1. When the first pair of non-matching elements occurs (that is, when *pResult* is not equal to zero), the function stops operation and returns the value *pResult*. The returned value is positive when *pSrc1*[i]>*pSrc2*[i] and negative when *pSrc1*[i]<*pSrc2*[i]. If the strings are equal, the function returns *pResult* = 0. The comparison is case-sensitive.

Example 11-4 shows how to use the function `ippsCompare_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len* is negative.

Example 11-4 Using the Functions ippsCompare and ippsCompareIgnoreCaseLatin

```
IppStatus compare( void ) {
   IppStatus st;
   char tmp[20];
   int result;
   char string1[] = "These functions compare two strings";
   char string2[] = "These FUNCTIONS compare two strings";
   /* Case sensitive */
   printf( "Compare strings:\n\t%s\n\t%s\n\n", string1, string2 );
  st = ippsCompare_8u((Ipp8u*)string1, (Ipp8u*)string2, strlen(string1),
&result);
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
  printf( "\tippsCompare_8u:                String 1 is %s string 2\n",
tmp );
   /* Case insensitive */
   st = ippsCompareIgnoreCaseLatin_8u((Ipp8u*)string1, (Ipp8u*)string2,
strlen(string1), &result);
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
  printf( "\tippsCompareIgnoreCaseLatin_8u:  String 1 is %s string 2\n",
tmp );
   return st;
}

Output:
Compare strings:
        These functions compare two strings
        These FUNCTIONS compare two strings

        ippsCompare_8u:                 String 1 is greater than string 2
        ippsCompareIgnoreCaseLatin_8u:  String 1 is equal to string 2
```

CompareIgnoreCase, CompareIgnoreCaseLatin

Compares two strings of the fixed length ignoring case.

Syntax

```
IppStatus ippsCompareIgnoreCase_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
    int len, int* pResult);
IppStatus ippsCompareIgnoreCaseLatin_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
    int len, int* pResult);
IppStatus ippsCompareIgnoreCaseLatin_16u(const Ipp16u* pSrc1, const Ipp16u*
    pSrc2, int len, int* pResult);
```

Parameters

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Description

The functions `ippsCompareIgnoreCase` and `ippsCompareIgnoreCaseLatin` are declared in the `ippch.h` file. These functions compare first *len* elements of two strings *pSrc1* and *pSrc2*. If all pairs of elements in the strings are equal, the function returns *pResult* = 0.
If the pair of non-matching elements occurs in the `i`-th position, the function stops operation and returns *pResult*. The returned value is positive when *pSrc1*[i]>*pSrc2*[i] and negative when *pSrc1*[i]<*pSrc2*[i]. The comparison is case-insensitive.

The function `ippsCompareIgnore` operates with Unicode characters.
The function `ippsCompareIgnoreLatin` operates with ASCII characters.

Example 11-4 shows how to use the function `ippsCompareIgnoreCaseLatin_8u`.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Equal

Compares two string of the fixed length for equality.

Syntax

```
IppStatus ippsEqual_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len,
    int* pResult);
IppStatus ippsEqual_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, int len,
    int* pResult);
```

Parameters

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Description

The function ippsEqual is declared in the ippch.h file. This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. Each element of the first string is compared with the corresponding element of the second string. When the first pair of non-matching elements is found, the function stops operation and stores 0 in *pResult*. If the strings are equal, the function stores 1 in *pResult*.
The comparison is case-sensitive.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

TrimC

Deletes all occurrences of a specified symbol in the beginning and in the end of the string.

Syntax

```
IppStatus ippsTrimC_8u(const Ipp8u* pSrc, int srcLen, Ipp8u odd, Ipp8u* pDst,
    int* pDstLen);
IppStatus ippsTrimC_16u(const Ipp16u* pSrc, int srcLen, Ipp16u odd, Ipp16u*
    pDst, int* pDstLen);
IppStatus ippsTrimC_8u_I(Ipp8u* pSrcDst, int* pLen, Ipp8u odd);
IppStatus ippsTrimC_16u_I(Ipp16u* pSrcDst, int* pLen, Ipp16u odd);
```

Parameters

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pSrcDst	Pointer to the source and destination string for the in-place operation.
pDst	Pointer to the destination string.
pDstLen	Pointer to the computed number of elements in the destination string.
pLen	Pointer to the number of elements in the source and destination string for the in-place operation.
odd	Symbol to be deleted.

Description

The function `ippsTrimC` is declared in the `ippch.h` file. This function deletes all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrc* containing *srcLen* elements. The function stores the result string containing *pDstLen* elements in *pDst*.

The in-place flavors of `ippsTrimC` delete all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrcDst* containing *pLen* elements. These functions store the result string containing *pLen* elements in *pSrcDst*.

The operation is case-sensitive.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *srcLen* or *pLen* is negative.

TrimCAny, TrimStartCAny, TrimEndCAny

Deletes all occurrences of any of the specified symbols in the beginning and in the end of the source string.

Syntax

```
IppStatus ippsTrimCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pTrim,
    int trimLen, Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u* pTrim,
    int trimLen, Ipp16u* pDst, int* pDstLen);
IppStatus ippsTrimStartCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u*
    pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimStartCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
    pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);
IppStatus ippsTrimEndCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pTrim,
    int trimLen, Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimEndCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
    pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);
```

Parameters

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pTrim	Pointer to the array containing the specified elements.
trimLen	Number of elements in the array.
pDst	Pointer to the destination string.

pDstLen	Pointer to the computed number of elements in the destination string.

Description

The functions `ippsTrimCAny`, `ippsTrimStartCAny`, and `ippsTrimEndCAny` are declared in the `ippch.h` file.
The function `ippsTrimCAny` deletes all occurrences of any of the specified elements stored in the array *pTrim* if they are present either in the beginning or in the end of the source string *pSrc*, and stores the result string containing *pDstLen* elements in *pDst*. The function stops operation when it finds the first non-matching element.
The functions `ippsTrimStartCAny` and `ippsTrimEndCAny` perform this operation only in the beginning or in the end of the source string *pSrc,* respectively.
The operation is case-sensitive.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *srcLen* or *trimLen* is negative.

ReplaceC

Replaces all occurrences of a specified element in the source string with another element.

Syntax

```
IppStatus ippsReplaceC_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, Ipp8u
    oldVal, Ipp8u newVal);
IppStatus ippsReplaceC_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len, Ipp16u
    oldVal, Ipp16u newVal);
```

Parameters

pSrc	Pointer to the source string.
len	Number of elements in the source string.
pDst	Pointer to the destination string.
oldVal	Element to be replaced.

newVal	Element that replaces *oldVal*.

Description

The function ippsReplaceC is declared in the ippch.h file. This function replaces all occurrences of a specified element *oldVal* in the source string *pSrc* with another specified element *newVal*, and stores the new string in the *pDst*.
The operation is case-sensitive.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Uppercase, UppercaseLatin

Converts alphabetic characters of a string to all uppercase symbols.

Syntax

```
IppStatus ippsUppercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsUppercase_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsUppercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsUppercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsUppercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsUppercaseLatin_16u_I(Ipp16u* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source string.
pDst	Pointer to the destination string.
pSrcDst	Pointer to the source and destination string for the in-place operation.
len	Number of elements in the string.

Description

The functions `ippsUppercase` and `ippsUppercaseLatin` are declared in the `ippch.h` file. These functions convert each alphabetic character of the source string *pSrc* to upper case and stores the result in *pDst*.

The in-place flavors of these functions convert each alphabetic character of the source string *pSrcDst* to upper case and store the result in *pSrcDst*.

The function `ippsUppercase` operates with Unicode characters.
The function `ippsUppercaseLatin` operates with ASCII characters.

Example 11-5 shows how to use the function `ippsUppercaseLatin_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len* is negative.

Lowercase, LowercaseLatin

Converts alphabetic characters of a string to all lowercase symbols.

Syntax

```
IppStatus ippsLowercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsLowercase_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsLowercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLowercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsLowercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsLowercaseLatin_16u_I(Ipp16u* pSrcDst, int len);
```

Parameters

pSrc	Pointer to the source string.
pDst	Pointer to the destination string.

pSrcDst	Pointer to the source and destination string for the in-place operation.
len	Number of elements in the string.

Description

The functions `ippsLowercase` and `ippsLowercaseLatin` are declared in the `ippch.h` file. These functions convert each alphabetic character of the source string *pSrc* to lower case and store the result in *pDst*.

The in-place flavors of these functions convert each alphabetic character of the source string *pSrcDst* to lower case and store the result in *pSrcDst*.

The function `ippsLowercase` operates with Unicode characters.
The function `ippsLowercaseLatin` operates with ASCII characters.

Example 11-5 shows how to use the function `ippsLowercaseLatin_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len* is negative.

Example 11-5 Using the functions ippsUppercaseLatin and ippsLowercaseLatin

```
IppStatus case( void ) {
   IppStatus st;
   char string[] = "These Functions Vary the Case!";
   char copy1[40], copy2[40];
   st = ippsLowercaseLatin_8u((Ipp8u*)string, (Ipp8u*)copy1,
strlen(string)+1);
   st = ippsUppercaseLatin_8u((Ipp8u*)string, (Ipp8u*)copy2,
strlen(string)+1);
   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );
   return resOK;
}
```

Example 11-5 Using the functions ippsUppercaseLatin and ippsLowercaseLatin

```
Output:
Mixed: These Functions Vary the Case!
Lower: these functions vary the case!
Upper: THESE FUNCTIONS VARY THE CASE!
```

Hash

Calculates the hash value for the string.

Syntax

```
IppStatus ippsHash_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHash_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);
```

Parameters

pSrc	Pointer to the source string.
len	Number of elements in the string.
pHashVal	Pointer to the result value.

Description

The function `ippsHash` is declared in the `ippch.h` file. This function produces the hash value *pHasVal* for the specified string *pSrc*. The hash value is fairly unique to the given string. If hash values of two strings are different, the strings are different as well. If the hush values are the same, the strings are probably identical. The hash value is calculated in the following manner: initial value is set to 0, then the hash value is calculated successively for each `i`-th string element as
pHashVal`[i] = 2*`*pHashVal*`[i-1]^`*pSrc*`[i]`, where ^ denotes a bitwise exclusive OR (XOR) operator. The hash value for the last element is the hash value for the whole string.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len* is negative.

Concat

Concatenates several strings together.

Syntax

```
IppStatus ippsConcat_8u_D2L(const Ipp8u* const pSrc[], const int* pSrcLen[],
    int numSrc, Ipp8u* pDst);
IppStatus ippsConcat_16u_D2L(const Ipp16u* const pSrc[], const int* pSrcLen[],
    int numSrc, Ipp16u* pDst);
IppStatus ippsConcat_8u(const Ipp8u* pSrc1, int len1, const Ipp8u* pSrc2, int
    len2, Ipp8u* pDst);
IppStatus ippsConcat_16u(const Ipp16u* pSrc1, int len1, const Ipp16u* pSrc2,
    int len2, Ipp16u* pDst);
```

Parameters

pSrc1	Pointer to the first source string.
len1	Number of elements in the first string.
pSrc2	Pointer to the second source string.
len2	Number of elements in the second string.
pSrc	Pointer to the array of source strings.
pSrcLen	Pointer to the array of lengths of the source strings.
numSrc	Number of source strings.
pDst	Pointer to the destination string.

Description

The function `ippsConcat` is declared in the `ippch.h` file. This function concatenates several strings together. Functions with `D2L` suffix operate with multiple *numSrc* strings *pSrc*[], while functions without this suffix operate with two strings *pSrc1* and *pSrc2* only. Resulting string is stored in the *pDst*. Necessary memory blocks should be allocated for the destination string before the function is called. Summary length of the strings to be concatenated can not be greater than `IPP_MAX_32S`.

Example 11-6 shows how to use the function `ippsConcat_8u`.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len1* or *len2* is negative, or *srcLen*[i] is negative for i < *numSrc*.
ippStsSizeErr	Indicates an error condition if *numSrc* is equal to or less than 0.

Example 11-6 Using the function ippsConcat

```
IppStatus concat( void ) {
   IppStatus st;
   char string[80] = "This is the initial string!";
   char suffix[] = " extra text to add to the string...";
   /* Combine strings with no more than 19 characters of suffix: */
   printf( "Before: %s\n", string );
   st = ippsConcat_8u((Ipp8u*)string, strlen(string), (Ipp8u*)suffix,
strlen(suffix),
      (Ipp8u*)string);
   printf( "After:  %s\n", string );
   return st;
}

Output:
Before: This is the initial string!
After:  This is the initial string! extra text to add to the string...
```

ConcatC

Concatenates several strings together and inserts symbol delimiters between them.

Syntax

```
IppStatus ippsConcatC_8u_D2L(const Ipp8u* const pSrc[], const int* pSrcLen[],
    int numSrc, Ipp8u delim, Ipp8u* pDst);
IppStatus ippsConcatC_16u_D2L(const Ipp16u* const pSrc[], const int* pSrcLen[],
    int numSrc, Ipp16u delim, Ipp16u* pDst);
```

Parameters

pSrc	Pointer to the array of source strings.

pSrcLen	Pointer to the array of lengths of the source strings.
numSrc	Number of source strings.
delim	Symbol delimiter.
pDst	Pointer to the destination string.

Description

The function `ippsConcatC` is declared in the `ippch.h` file. This function concatenates *numSrc* strings *pSrc* together and inserts a specified delimiter symbol *delim* between them in the resulting string *pDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *srcLen*[i] is negative for i < *numSrc*.
`ippStsSizeErr`	Indicates an error condition if *numSrc* is equal to or less than 0.

SplitC

Splits source string into separate parts.

Syntax

```
IppStatus ippsSplitC_8u_D2L(const Ipp8u* pSrc, int srcLen, Ipp8u delim,
    Ipp8u* pDst[], int* pDstLen[], int* pNumDst);
IppStatus ippsSplitC_16u_D2L(const Ipp16u* pSrc, int srcLen, Ipp16u delim,
    Ipp16u* pDst[], int* pDstLen[], int* pNumDst);
```

Parameters

pSrc	Pointer to the source strings.
srcLen	Number of elements in the source string.
delim	Symbol delimiter.
pDst	Pointer to the array of the destination strings.
pDstLen	Pointer to array of the destination string lengths.

pNumDst	Number of destination strings.

Description

The function `ippsSplitC` is declared in the `ippch.h` file. This function breaks source string *pSrc* into *pNumDst* separate strings *pDst* using a specified symbol *delim* as a delimiter.
If `n` specified delimiters occur in the beginning or in the end of the source string, then `n` empty strings are appended to the array of destination strings.
If `n` specified delimiters occur in a certain position within the source string, then (`n-1`) empty strings will be inserted into the array of destination strings, where `n` is the number of delimiter occurrences in this position.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *srcLen* is negative, or *dstLen* is negative for `i < pNumDst`.
`ippStsSizeErr`	Indicates an error condition if *pNumDst* is equal to or less than 0.
`ippStsOvermatchStrings`	Indicates a warning if number of output strings exceeds the initially specified number *pNumDst*; in this case odd strings are discarded.
`ippStsOverlongString`	Indicates a warning if in some output strings the number of elements exceeds the initially specified value *pDstLen*; in this case corresponding strings are truncated to initial lengths.

Regular Expressions

This section describes the Intel IPP functions that perform matching operations with the Perl-compatible regular expression patterns. See http://search.cpan.org/dist/perl/pod/perlre.pod for more details about Perl-compatible regular expressions.

The current version of the Intel IPP functions for regular expressions have some limitations, specifically they do not support literal (metacharacters `\l`, `\L`, `\u`, `\U`, `\N{name}`), embedded Perl code (`?{code}`), extended regular expression (`??{code}`) and all metacharacters for working with Unicode.

RegExpInitAlloc

Allocates memory and initializes the structure for processing matching operation with regular expressions.

Syntax

```
IppStatus ippsRegExpInitAlloc(const char* pPattern, const char* pOptions,
    IppRegExpState** ppRegExpState, int* pErrOffset);
```

Parameters

pPattern	Pointer to the pattern of regular expression.
pOptions	Pointer to options for compiling and executing regular expressions (possible values `i`, `s`, `m`, `x`, `g`). It must be `NULL` if no options are required.
ppRegExpState	Double pointer to the structure containing internal form of a regular expression.
pErrOffset	Pointer to the offset in the pattern if compiling is break.

Description

The function `ippsRegExpInitAlloc` is declared in the `ippch.h` file. This function allocates memory and initializes a regular expression state structure *ppRegExpState*. The function compiles the initial pattern of regular expressions *pPattern* in accordance with the compiling options specified by *pOptions*, converts it to the specific internal form, and stores it in the initialized structure. This structure is used by the function ippsRegExpFind to perform matching operation.

If the compiling is not completed, the function returns the pointer *pErrOffset* pointed to the position in the pattern where the compiling is interrupted.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.
`ippStsRegExpOptionsErr`	Indicates an error if specified options are incorrect
`ippStsRegExpQuantifierErr`	Indicates an error if the quantifier is incorrect

ippStsRegExpGroupingErr	Indicates an error if the grouping is incorrect
ippStsRegExpBackRefErr	Indicates an error if the back reference is incorrect
ippStsRegExpChClassErr	Indicates an error if the character class is incorrect
ippStsRegExpMetaChErr	Indicates an error if the metacharacter is incorrect

RegExpFree

Frees the memory allocated for a regular expression state structure.

Syntax

```
IppStatus ippsRegExpFree(IppsRegExpState* pRegExpState);
```

Parameters

pRegExpState	Pointer to the pattern of regular expression.

Description

The function ippsRegExpFree is declared in the ippch.h file. This function frees memory allocated by the function ippsRegExpInitAlloc for the pyramid structure *pRegExpState*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if *pRegExpState* pointer is NULL.

RegExpInit

Initializes the structure for processing matching operation with regular expressions.

Syntax

```
IppStatus ippsRegExpInit(const char* pPattern, const char* pOptions,
    IppRegExpState* pRegExpState, int* pErrOffset)
```

Parameters

pPattern	Pointer to the pattern of regular expression.
pOptions	Pointer to options for compiling and executing regular expressions (possible values `i`, `s`, `m`, `x`, `g`). It must be `NULL` if no options are required.
pRegExpState	Pointer to the structure containing internal form of a regular expression.
pErrOffset	Pointer to the offset in the pattern if compiling is break.

Description

The function `ippsRegExpInit` is declared in the `ippch.h` file. This function initializes a regular expression state structure *pRegExpState* in the external buffer. The size of this buffer should be computed previously by calling the function `ippsRegExpGetSize`. The function compiles the initial pattern of regular expressions *pPattern* in accordance with the compiling options specified by *pOptions*, converts it to the specific internal form, and stores it in the initialized structure. This structure is used by the function `ippsRegExpFind` to perform matching operation.

If the compiling is not completed, the function returns the pointer *pErrOffset* pointed to the position in the pattern where the compiling is interrupted.

CAUTION. The parameter *pPattern* must be the same for both functions `ippsRegExpInit` and `ippsRegGetSize`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.
`ippStsRegExpOptionsErr`	Indicates an error if specified options are incorrect
`ippStsRegExpQuantifierErr`	Indicates an error if the quantifier is incorrect
`ippStsRegExpGroupingErr`	Indicates an error if the grouping is incorrect
`ippStsRegExpBackRefErr`	Indicates an error if the back reference is incorrect
`ippStsRegExpChClassErr`	Indicates an error if the character class is incorrect
`ippStsRegExpMetaChErr`	Indicates an error if the metacharacter is incorrect

RegExpGetSize

Computes the size of the regular expression state structure.

Syntax

```
IppStatus ippsRegExpGetSize(const char* pPattern, int* pRegExpStateSize);
```

Parameters

pPattern	Pointer to the pattern of regular expression.
pRegExpStateSize	Pointer to the computed size of the regular expression structure.

Description

The function `ippsRegExpGetSize` is declared in the `ippch.h` file. This function computes the size of the memory that is necessary to initialize by the function `ippsRegExpInit` the regular expression state structure containing the pattern *pPattern* in the internal form. The value of the computed size is stored in the *pRegExpStateSize*.

CAUTION. The parameter *pPattern* must be the same for both functions `ippsRegExpInit` and `ippsRegGetSize`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

RegExpSetMatchLimit

Sets the value of the `matchLimit` parameter.

Syntax

```
IppStatus ippsRegExpSetMatchLimit(int matchLimit, IppRegExpState*
    pRegExpState);
```

Parameters

matchLimit	Value of the of the matches kept in stack.
pRegExpState	Pointer to the structure containing internal form of a regular expression.

Description

The function `ippsRegExpSetMatchLimit` is declared in the `ippch.h` file. This function sets the value of the parameter *matchLimit* that specifies how many times the function `ippsRegExpFind` can be called through the single execution avoiding the possible stack overflow. The default value is set very large, so you should set this parameter to the reasonable value in accordance with your needs.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if the *pRegExpState* pointer is `NULL`.

RegExpFind

Looks for the occurrences of the substrings matching the specified regular expression.

Syntax

```
IppStatus ippsRegExpFind_8u(const Ipp8u* pSrc, int srcLen, IppRegExpState*
    pRegExpState, IppRegExpFind* pFind, int* pNumFind);
```

Parameters

pSrc	Pointer to the source strings.
srcLen	Number of elements in the source string.
pRegExpState	Pointer to the structure containing internal form of regular expression.
pFind	Array of pointers to the matching substrings.
pNumFind	Size of the array *pFind* on input, number of matching substrings on output.

Description

The function `ippsRegExpFind` is declared in the `ippch.h` file. This function search through the *srcLen* elements of the source string *pSrc* for substrings that match the specified regular expression in accordance with the regular expression pattern that is stored in the structure *pRegExpState*. This structure must be initialized by the functions either `ippsRegExpInit` or `ippsRegExpInitAlloc` beforehand.
Initially the parameter *pNumFind* specifies the size of array *pFind*, the output parameter *pNumFind* returns the number of the matching substrings.

pFind->pFindOffset specifies the offset of the pointer to the matching substring, and *pFind->lenFind* - number of elements in the matching substring. *pFind*[0] points to the substring that matches the whole regular expression, *pFind*[1] points to the substring that matches the first grouping, *pFind*[2] points to the substring that matches the second grouping, and so on.

CAUTION. It is recommended to set the default value of the parameter *matchLimit* in accordance with real necessity by calling the function `ippsRegExpSetMatchLimit` beforehand.

Example 11-7 shows how to use the function `ippsRegExpFind_8u`.

Example 11-7 Using of the Function ippsRegExpFind

```
IppRegExpFind pFind[2];
IppRegExpState *pState;
IppStatus status;
Ipp8u pSrc[] = "Hello World!";
int   srcLen = sizeof(pSrc)/sizeof(pSrc[0]) - 1;
int   num    = sizeof(pFind)/sizeof(pFind[0]);
int   i, err;


// to perform ordinary matching:
if( (status = ippsRegExpInitAlloc( "[hH]ello (\\w+)", NULL, &pState, &err )) !=
   ippStsNoErr ) {

// to perform global matching:
if( (status = ippsRegExpInitAlloc( "[hH]ello (\\w+)", "g", &pState, &err )) !=
   ippStsNoErr ) {

  printf( "Error: \"%s\" in position %d\n", ippGetStatusString( status ), err );
  return 1;
}
ippsRegExpFind_8u( pSrc, srcLen, pState, pFind, &num );



//Output: ordinary matching:

num = 2;
pFind[0].pFind = "Hello World"; pFind[0].lenFind = 11;
pFind[1].pFind = "World"; pFind[1].lenFind = 5;

//Output: global matching:

num = 1;
pFind[0].pFind = "World"; pFind[0].lenFind = 5;
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error condition if *`srcLen`* is negative, or *`pNumFind`* is less than or equal to 0.
`ippStsRegExpErr`	The state structure *`pRegExpState`* contains wrong data.
`ippStsRegExpMatchLimitErr`	The match limit has been exhausted.

Fixed-Accuracy Arithmetic Functions

12

This chapter describes Intel® IPP fixed-accuracy transcendental mathematical real and complex functions of vector arguments. These functions take an input vector as argument, compute values of the respective elementary function element-wise, and return the results in an output vector.

Function specifications comply with the common API agreement of Intel IPP, but include some new features essential to scientific arithmetic functions. The main feature is a more elaborate specification of accuracy that differs from the common definition in adding several new levels of accuracy, besides original levels introduced by single precision and double precision data formats.

Fixed-accuracy vector functions implementation supports IEEE-754 standard in all flavors, which means that:

- All functions have a precisely determined and guaranteed level of accuracy for all argument values.
- All special value processing and exceptions handling requirements are met, which implies that when accuracy is below the standard level, the function meets IEEE-754 requirements in all other respects.

The choice of accuracy levels should be based on practical experience and identified application demands. Available options are specified in the function name suffix and include `A11`, `A21`, or `A24` for the single precision, and `A50` or `A53` for the double precision data format. Flavors `A11`, `A21`, and `A50` provide approximately 3, 6, and 15 exact decimal digits, respectively. For flavors `A24` and `A53`, the maximum guaranteed error is within 1 ulp and in most cases does not exceed 0.55 ulp.

Fixed-accuracy arithmetic functions subset of Intel IPP has the similar functionality as the respective part of Intel® Math Kernel Library (Intel®MKL) .

However, Intel IPP provides lower-level transcendental functions that have separate flavors for each mode of operations and data type and are better suitable for multimedia and signal processing in real time applications.
The full list of these functions is given in Table 12-1.

Table 12-1 Intel IPP Fixed-Accuracy Arithmetic Functions

Function Short Name	Data Types	Description
Power and Root Functions		
Inv	32f, 64f	Computes inverse value of each vector element
Div	32f, 64f	Divides elements of one vector by corresponding elements of another vector
Sqrt	32f, 64f, 32fc, 64fc	Computes square root of each vector element
InvSqrt	32f, 64f	Computes inverse square root of each vector element
Cbrt	32f, 64f	Computes cube root of each vector element
InvCbrt	32f, 64f	Computes inverse cube root of each vector element
Pow	32f, 64f, 32fc, 64fc	Raises each element of one vector to the power of corresponding element of another vector
Powx	32f, 64f, 32fc, 64fc	Raises each element of a vector to a constant power
Hypot	32f, 64f	Computes a square root of sum of two squared elements
Exponential and Logarithmic Functions		
Exp	32f, 64f, 32fc, 64fc	Raises *e* to the power of each vector element
Ln	32f, 64f, 32fc, 64fc	Computes natural logarithm of each vector element
Log10	32f, 64f, 32fc, 64fc	Computes common logarithm of each vector element
Trigonometric Functions		
Cos	32f, 64f, 32fc, 64fc	Computes cosine of each vector element
Sin	32f, 64f, 32fc, 64fc	Computes sine of each vector element
SinCos	32f, 64f	Computes sine and cosine of each vector element
Tan	32f, 64f, 32fc, 64fc	Computes tangent of each vector element

Table 12-1 Intel IPP Fixed-Accuracy Arithmetic Functions (continued)

Function Short Name	Data Types	Description
Acos	32f, 64f, 32fc, 64fc	Computes inverse cosine of each vector element
Asin	32f, 64f, 32fc, 64fc	Computes inverse sine of each vector element
Atan	32f, 64f, 32fc, 64fc	Computes inverse tangent of each vector element
Atan2	32f, 64f	Computes four-quadrant inverse tangent of elements of two vectors
Hyperbolic Functions		
Cosh	32f, 64f, 32fc, 64fc	Computes hyperbolic cosine of each vector element
Sinh	32f, 64f, 32fc, 64fc	Computes hyperbolic sine of each vector element
Tanh	32f, 64f, 32fc, 64fc	Computes hyperbolic tangent of each vector element.
Acosh	32f, 64f, 32fc, 64fc	Computes inverse hyperbolic cosine of each vector element.
Asinh	32f, 64f, 32fc, 64fc	Computes inverse hyperbolic sine of each vector element
Atanh	32f, 64f, 32fc, 64fc	Computes inverse hyperbolic tangent of each vector element
Special Functions		
Erf	32f, 64f	Computes the error function value
Erfc	32f, 64f	Computes the complementary error function value
ErfInv	32f, 64f	Computes the inverse error function value
Rounding Functions		
Floor	32f, 64f	Computes integer value rounded toward minus infinity for each vector element
Ceil	32f, 64f	Computes integer value rounded toward plus infinity for each vector element
Trunc	32f, 64f	Computes integer value rounded toward zero for each vector element
Round	32f, 64f	Computes integer value rounded to nearest for each vector element

Table 12-1 Intel IPP Fixed-Accuracy Arithmetic Functions (continued)

Function Short Name	Data Types	Description
NearbyInt	32f, 64f	Computes rounded integer value in current rounding mode for each vector element
Rint	32f, 64f	Computes rounded integer value in current rounding mode for each vector element with inexact result exception raised for each changed value
Modf	32f, 64f	Computes truncated integer value and remaining fraction part for each vector element

NOTE. You should not confuse fixed-accuracy arithmetic functions described in this chapter with functions in chapter 5 that have similar functionality but follow different accuracy specifications.

Intel IPP fixed-accuracy arithmetic functions may return status codes of normal execution (IppStsNoErr), error conditions IppStsSizeErr, IppStsNullPtrErr (see Table 2-3 in Chapter 2), and the following specific warnings: IppStsDomain, IppStsSingularity, IppStsOverflow, IppStsUnderflow. In case of warnings, the value returned is positive and the computation is continued.
Table 12-2 lists status codes and the corresponding messages for these warnings.

Table 12-2 Warning Status Codes for Fixed-Accuracy Arithmetic Functions

Status	Value	Message
IppStsOverflow	12	Overflow occurred in the operation.
IppStsUnderflow	17	Underflow occurred in the operation.
IppStsSingularity	18	Singularity occurred in the operation.
IppStsDomain	19	Argument is out of the function domain.

See "Handling of Special Cases" for more information on function operation in cases when their arguments take on specific values that are outside the range of function definition.

NOTE. All functions described in this chapter support in-place operation.

Power and Root Functions

Inv

Computes inverse value of each vector element.

Syntax

```
IppStatus ippsInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsInv` is declared in the `ippvm.h` file. This function computes the inverse value of each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{pSrc[n]}, \; 0 \le n < len \; .$$

Example 12-1 shows how to use the function `ippsInv`.

Example 12-1 Using `ippsInv` Function

```
IppStatus ippsInv_32f_A21_sample(void)
{
const Ipp32f x[4] = {-9.975, 1.272, -6.134, 6.175};
Ipp32f      y[4];

      IppStatus st = ippsInv_32f_A21( x, y, 4 );

      printf(" ippsInv_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsInv_32f_A21:
 x = -9.975 1.272 -6.134 6.175
 y = -0.100 0.786 -0.163 0.162
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Div

Divides each element of the first vector by corresponding element of the second vector.

Syntax

```
IppStatus ippsDiv_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsDiv_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f*  pDst, int len);
IppStatus ippsDiv_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f*  pDst, int len);
IppStatus ippsDiv_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsDiv_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsDiv` is declared in the `ippvm.h` file. This function divides each element of the vector *pSrc1* by the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsDiv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsDiv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsDiv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsDiv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsDiv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as $pDst[n] = \frac{pSrc1[n]}{pSrc2[n]}, 0 \le n < len$.

Example 12-2 shows how to use the function `ippsDiv`.

Example 12-2 Using `ippsDiv` Function

```
IppStatus ippsDiv_32f_A21_sample(void)
{
const Ipp32f x1[4] = {599.088, 735.034, 572.448, 151.640};
const Ipp32f x2[4] = {385.297, 609.005, 361.403, 225.182};
Ipp32f          y[4];

      IppStatus st = ippsDiv_32f_A21( x1, x2, y, 4 );

      printf(" ippsDiv_32f_A21:\n");
      printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
      printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2],
x2[3]);
      printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],
y[3]);
      return st;
}

Output results:

 ippsDiv_32f_A21:
 x1 = 599.088 735.034 572.448 151.640
 x2 = 385.297 609.005 361.403 225.182
 y  = 1.555 1.207 1.584 0.673
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc2* is equal to 0.

Sqrt

Computes square root of each vector element.

Syntax

```
IppStatus ippsSqrt_32f_A11 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_32f_A21 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_32f_A24 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_64f_A50 ( const Ipp64f*  pSrc, Ipp64f*  pDst, int len );
IppStatus ippsSqrt_64f_A53 ( const Ipp64f*  pSrc, Ipp64f*  pDst, int len );

IppStatus ippsSqrt_32fc_A11 ( const Ipp32fc*  pSrc, Ipp32fc*  pDst, int len );
IppStatus ippsSqrt_32fc_A21 ( const Ipp32fc*  pSrc, Ipp32fc*  pDst, int len );
IppStatus ippsSqrt_32fc_A24 ( const Ipp32fc*  pSrc, Ipp32fc*  pDst, int len );
IppStatus ippsSqrt_64fc_A50 ( const Ipp64fc*  pSrc, Ipp64fc*  pDst, int len );
IppStatus ippsSqrt_64fc_A53 ( const Ipp64fc*  pSrc, Ipp64fc*  pDst, int len );
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsSqrt` is declared in the `ippvm.h` file. This function computes square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsSqrt_32f_A11` and `ippsSqrt_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsSqrt_32f_A21` and `ippsSqrt_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsSqrt_32f_A24` and `ippsSqrt_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSqrt_64f_A50` and `ippsSqrt_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsSqrt_64f_A53` and `ippsSqrt_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sqrt{pSrc[n]}, 0 \le n < len$.

Example 12-3 shows how to use the function `ippsSqrt`.

Example 12-3 Using `ippsSqrt` Function

```
IppStatus ippsSqrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {5850.093, 4798.730, 3502.915, 8959.624};
Ipp32f          y[4];

      IppStatus st = ippsSqrt_32f_A21( x, y, 4 );

      printf(" ippsSqrt_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsSqrt_32f_A21:
 x = 5850.093 4798.730 3502.915 8959.624
 y = 76.486 69.273 59.185 94.655
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.

InvSqrt

Computes inverse square root of each vector element.

Syntax

```
IppStatus ippsInvSqrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvSqrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsInvSqrt` is declared in the `ippvm.h` file. This function computes inverse square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsInvSqrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInvSqrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInvSqrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInvSqrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInvSqrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{\sqrt{pSrc[n]}}, \; 0 \le n < len.$$

Example 12-4 shows how to use the function `ippsInvSqrt`.

Example 12-4 Using `ippsInvSqrt` Function

```
IppStatus ippsInvSqrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {7105.043, 5135.398, 3040.018, 149.944};
Ipp32f          y[4];

      IppStatus st = ippsInvSqrt_32f_A21( x, y, 4 );

      printf(" ippsInvSqrt_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsInvSqrt_32f_A21:
 x = 7105.043 5135.398 3040.018 149.944
 y = 0.012 0.014 0.018 0.082
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Cbrt

Computes cube root of each vector element.

Syntax

```
IppStatus ippsCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsCbrt` is declared in the `ippvm.h` file. This function computes cube root of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \sqrt[3]{pSrc[n]},\ 0 \le n < len.$$

Example 12-5 shows how to use the function `ippsCbrt`.

Example 12-5 Using `ippsCbrt` Function

```
IppStatus ippsCbrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {6456.801, 4932.096, -6517.838, 7178.869};
Ipp32f          y[4];

      IppStatus st = ippsCbrt_32f_A21( x, y, 4 );

      printf(" ippsCbrt_32f_A21:\n");
```

Example 12-5 Using `ippsCbrt` Function (continued)

```
        printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
        printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
        return st;
}

Output results:

 ippsCbrt_32f_A21:
 x = 6456.801 4932.096 -6517.838 7178.869
 y = 18.621 17.022 -18.680 19.291
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

InvCbrt

Computes inverse cube root of each vector element.

Syntax

```
IppStatus ippsInvCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsInvCbrt` is declared in the `ippvm.h` file. This function computes inverse cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsInvCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInvCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInvCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInvCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInvCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{\sqrt[3]{pSrc[n]}}, \; 0 \le n < len.$$

Example 12-6 shows how to use the function `ippsInvCbrt`.

Example 12-6 Using `ippsInvCbrt` Function

```
IppStatus ippsInvCbrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {914.120, 3644.584, 1473.214, 1659.070};
Ipp32f          y[4];

      IppStatus st = ippsInvCbrt_32f_A21( x, y, 4 );

      printf(" ippsInvCbrt_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsInvCbrt_32f_A21:
 x = 914.120 3644.584 1473.214 1659.070
 y = 0.103 0.065 0.088 0.084
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsSingularity	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Pow

Raises each element of the first vector to the power of corresponding element of the second vector.

Syntax

```
IppStatus ippsPow_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsPow_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);

IppStatus ippsPow_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
    Ipp32fc* pDst, int len);
IppStatus ippsPow_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
    Ipp32fc* pDst, int len);
IppStatus ippsPow_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
    Ipp32fc* pDst, int len);
IppStatus ippsPow_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
    Ipp64fc* pDst, int len);
IppStatus ippsPow_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
    Ipp64fc* pDst, int len);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsPow` is declared in the `ippvm.h` file. This function raises each element of vector *pSrc1* to the power of the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsPow_32f_A11` and `ippsPow_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsPow_32f_A21` and `ippsPow_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsPow_32f_A24` and `ippsPow_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsPow_64f_A50` and `ippsPow_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsPow_64f_A53` and `ippsPow_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Note that for `Pow` complex functions there may be argument ranges where the accuracy specification does not hold.

The computation is performed as follows:

$$pDst[n] = (pSrc1[n])^{pSrc2[n]},\ 0 \le n < len.$$

Example 12-7 shows how to use the function `ippsPow`.

Example 12-7 Using `ippsPow` Function

```
IppStatus ippsPow_32f_A21_sample(void)
{
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
Ipp32f          y[4];
```

Example 12-7 Using `ippsPow` Function (continued)

```
        IppStatus st = ippsPow_32f_A21( x1, x2, y, 4 );

        printf(" ippsPow_32f_A21:\n");
        printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
        printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2],
x2[3]);
        printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],
y[3]);
        return st;
}


Output results:

 ippsPow_32f_A21:
 x1 = 0.483 0.565 0.776 0.252
 x2 = 0.823 0.991 0.411 0.692
 y  = 0.549 0.568 0.901 0.386
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one pair of the source elements meets the following condition: element of *pSrc1* is finite, less than 0, and element of *pSrc2* is finite, non-integer.
`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of *pSrc1* is equal to 0, and element of *pSrc2* is integer and less than 0.

Powx

Raises each element of a vector to a constant power.

Syntax

```
IppStatus ippsPowx_32f_A11 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_32f_A21 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_32f_A24 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_64f_A50 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
    Ipp64f* pDst, int len);
IppStatus ippsPowx_64f_A53 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
    Ipp64f* pDst, int len);

IppStatus ippsPowx_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue,
    Ipp32fc* pDst, int len);
IppStatus ippsPowx_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue,
    Ipp32fc* pDst, int len);
IppStatus ippsPowx_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue,
    Ipp32fc* pDst, int len);
IppStatus ippsPowx_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc ConstValue,
    Ipp64fc* pDst, int len);
IppStatus ippsPowx_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc ConstValue,
    Ipp64fc* pDst, int len);
```

Parameters

pSrc1	Pointer to the source vector.
ConstValue	Constant value.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsPowx` is declared in the `ippvm.h` file. This function raises each element of the vector *pSrc1* to the constant power *ConstValue* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsPowx_32f_A11` and `ippsPowx_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsPowx_32f_A21` and `ippsPowx_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsPowx_32f_A24` and `ippsPowx_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsPowx_64f_A50` and `ippsPowx_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsPowx_64f_A53` and `ippsPowx_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Note that for `Powx` complex functions there may be argument ranges where the accuracy specification does not hold.

The computation is performed as follows:

$$pDst[n] = (pSrc1[n])^{ConstValue}, \; 0 \le n < len.$$

Example 12-8 shows how to use the function `ippsPowx`.

Example 12-8 Using `ippsPowx` Function

```
IppStatus ippsPowx_32f_A21_sample(void)
{
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2 = 0.823;
Ipp32f          y[4];

      IppStatus st = ippsPowx_32f_A21( x1, x2, y, 4 );

      printf(" ippsPowx_32f_A21:\n");
      printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
      printf(" x2 = %.3f \n", x2);
      printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],
y[3]);
      return st;
```

Example 12-8 Using `ippsPowx` Function (continued)

```
}

Output results:

 ippsPowx_32f_A21:
 x1 = 0.483 0.565 0.776 0.252
 x2 = 0.823
 y  = 0.549 0.568 0.901 0.386
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one pair of the elements meets the following condition: element of *pSrc1* is finite, less than 0, and *ConstValue* is finite, non-integer.
`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of *pSrc1* is equal to 0, and *ConstValue* is integer and less than 0.

Hypot

Computes a square root of sum of two squared elements.

Syntax

```
IppStatus ippsHypot_32f_A11 (const Ipp32f* pSrc1, const Ipp32f pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsHypot_32f_A21 (const Ipp32f* pSrc1, const Ipp32f pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsHypot_32f_A24 (const Ipp32f* pSrc1, const Ipp32f pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsHypot_64f_A50 (const Ipp64f* pSrc1, const Ipp64f pSrc2,
    Ipp64f* pDst, int len);
```

```
IppStatus ippsHypot_64f_A53 (const Ipp64f* pSrc1, const Ipp64f pSrc2,
    Ipp64f* pDst, int len);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsHypot` is declared in the `ippvm.h` file. This function computes square of each element of the *pSrc1* and *pSrc2* vectors, sums corresponding elements, computes square roots of each sum and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsHypot_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsHypot_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsHypot_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsHypot_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsHypot_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Note that for `Powx` complex functions there may be argument ranges where the accuracy specification does not hold.

The computation is performed as follows:

$$pDst[n] = \sqrt{(pSrc1[n])^2 + (pSrc2[n])^2},\ 0 \le n < len.$$

Example 12-9 shows how to use the function `ippsHypot`.

Example 12-9 Using `ippsHypot` Function

```
IppStatus ippsHypot_32f_A21_sample(void)
{
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
```

Example 12-9 Using `ippsHypot` Function (continued)

```
Ipp32f y[4];

      IppStatus st = ippsHypot_32f_A21( x1, x2, y, 4 );
      printf(" ippsHypot_32f_A21:\n");
      printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
      printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2],
x2[3]);
      printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],
y[3]);
      return st;
}


Output results:

 ippsHypot_32f_A21:
 x1 = 0.483 0.565 0.776 0.252
 x2 = 0.823 0.991 0.411 0.692
 y  = 0.954 1.141 0.878 0.736
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Exponential and Logarithmic Functions

Exp

Raises e to the power of each vector element.

Syntax

```
IppStatus ippsExp_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsExp_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsExp_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsExp_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsExp_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsExp_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsExp` is declared in the `ippvm.h` file. This function raises *e* to the power of each element of *pSrc* and stores the result in the corresponding element of *pDst*.
For single precision data:

function flavors `ippsExp_32f_A11` and `ippsExp_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsExp_32f_A21` and `ippsExp_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsExp_32f_A24` and `ippsExp_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsExp_64f_A50` and `ippsExp_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsExp_64f_A53` and `ippsExp_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = e^{pSrc[n]}, 0 \le n < len$.

Example 12-10 shows how to use the function `ippsExp`.

Example 12-10Using `ippsExp` Function

```
IppStatus ippsExp_32f_A21_sample(void)
{
const Ipp32f x[4] = {4.885, -0.543, -3.809, -4.953};
Ipp32f          y[4];

      IppStatus st = ippsExp_32f_A21( x, y, 4 );

      printf(" ippsExp_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsExp_32f_A21:
 x = 4.885 -0.543 -3.809 -4.953
 y = 132.324 0.581 0.022 0.007
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsOverflow`	In real functions, indicates a warning that the function overflows, that is, at least one of elements of *pSrc* is greater than `Ln(FPMAX)`, where `FPMAX` is the maximum representable floating-point number.

IppStsUnderflow	In real functions, indicates a warning that the function underflows, that is, at least one of elements of *pSrc* is less than Ln(FPMIN), where FPMIN is the minimum positive floating-point value.

Ln

Computes natural logarithm of each vector element.

Syntax

```
IppStatus ippsLn_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsLn_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsLn_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsLn_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsLn_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsLn_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function ippsLn is declared in the ippvm.h file. This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors ippsLn_32f_A11 and ippsLn_32fc_A11 guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsLn_32f_A21` and `ippsLn_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsLn_32f_A24` and `ippsLn_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsLn_64f_A50` and `ippsLn_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsLn_64f_A53` and `ippsLn_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \log_e(pSrc[n]),\ 0 \le n < len.$$

Example 12-11 shows how to use the function `ippsLn`.

Example 12-11Using `ippsLn` Function

```
IppStatus ippsLn_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.188, 3.841, 5.363, 5.755};
Ipp32f          y[4];

      IppStatus st = ippsLn_32f_A21( x, y, 4 );

      printf(" ippsLn_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}


Output results:

 ippsLn_32f_A21:
 x = 0.188 3.841 5.363 5.755
 y = -1.670 1.346 1.680 1.750
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.
`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Log10

Computes common logarithm of each vector element.

Syntax

```
IppStatus ippsLog10_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLog10_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLog10_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsLog10_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsLog10_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsLog10_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsLog10_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsLog10` is declared in the `ippvm.h` file. This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsLog10_32f_A11` and `ippsLog10_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsLog10_32f_A21` and `ippsLog10_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsLog10_32f_A24` and `ippsLog10_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavosr `ippsLog10_64f_A50` and `ippsLog10_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsLog10_64f_A53` and `ippsLog10_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \log_{10}(pSrc[n]),\ 0 \le n < len.$$

Example 12-12 shows how to use the function `ippsLog10`.

Example 12-12 Using `ippsLog10` Function

```
IppStatus ippsLog10_32f_A21_sample(void)
{
const Ipp32f x[4] = {6.057, 6.111, 1.746, 6.664};
Ipp32f      y[4];

      IppStatus st = ippsLog10_32f_A21( x, y, 4 );

      printf(" ippsLog10_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsLog10_32f_A21:
 x = 6.057 6.111 1.746 6.664
 y = 0.782 0.786 0.242 0.824
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.
IppStsSingularity	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Trigonometric Functions

Cos

Computes cosine of each vector element.

Syntax

```
IppStatus ippsCos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsCos_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCos_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCos_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCos_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsCos_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function ippsCos is declared in the ippvm.h file. This function computes a cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors ippsCos_32f_A11 and ippsCos_32fc_A11 guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors ippsCos_32f_A21 and ippsCos_32fc_A21 guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors ippsCos_32f_A24 and ippsCos_32fc_A24 guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors ippsCos_64f_A50 and ippsCos_64fc_A50 guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors ippsCos_64f_A53 and ippsCos_64fc_A53 guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \cos(pSrc[n]),\ 0 \le n < len.$$

Example 12-13 shows how to use the function ippsCos.

Example 12-13Using ippsCos Function

```
IppStatus ippsCos_32f_A21_sample(void)
{
const Ipp32f x[4] = {-984.222, -2957.549, -8859.218, 2153.691};
Ipp32f          y[4];

      IppStatus st = ippsCos_32f_A21( x, y, 4 );

      printf(" ippsCos_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsCos_32f_A21:
 x = -984.222 -2957.549 -8859.218 2153.691
 y = -0.619 -0.258 0.997 0.129
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

Sin

Computes sine of each vector element.

Syntax

```
IppStatus ippsSin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsSin_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSin_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSin_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSin_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSin_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function ippsSin is declared in the ippvm.h file. This function computes a sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsSin_32f_A11` and `ippsSin_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsSin_32f_A21` and `ippsSin_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsSin_32f_A24` and`ippsSin_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSin_64f_A50` and `ippsSin_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsSin_64f_A53` and `ippsSin_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \sin(pSrc[n]),\ 0 \le n < len.$$

Example 12-14 shows how to use the function `ippsSin`.

Example 12-14Using `ippsSin` Function

```
IppStatus ippsSin_32f_A21_sample(void)
{
const Ipp32f x[4] = {5666.372, 6052.125, 397.656, -3960.997};
Ipp32f          y[4];

      IppStatus st = ippsSin_32f_A21( x, y, 4 );

      printf(" ippsSin_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsSin_32f_A21:
 x = 5666.372 6052.125 397.656 -3960.997
 y = -0.873 0.988 0.970 -0.524
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

SinCos

Computes sine and cosine of each vector element.

Syntax

```
IppStatus ippsSinCos_32f_A11 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A21 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A24 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A50 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A53 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
```

Parameters

pSrc	Pointer to the first source vector.
pDst1	Pointer to the destination vector for sine values.
pDst2	Pointer to the destination vector for cosine values.
len	Number of elements in the vectors.

Description

The function `ippsSinCos` is declared in the `ippvm.h` file. This function computes sine of each element of *pSrc* and stores the result in the corresponding element of *pDst1*; computes cosine of each element of *pSrc* and stores the result in the corresponding element of *pDst2*.

For single precision data:

function flavor `ippsSinCos_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsSinCos_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsSinCos_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsSinCos_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsSinCos_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst1[n] = \sin(pSrc[n])$, $pDst2[n] = \cos(pSrc[n])$, $0 \le n < len$.

Example 12-15 shows how to use the function `ippsSinCos`.

Example 12-15Using `ippsSinCos` Function

```
IppStatus ippsSinCos_32f_A21_sample(void)
{
const Ipp32f x[4] = {3857.845, -3939.024, -1468.856, -8592.486};
Ipp32f      y1[4];
Ipp32f      y2[4];

      IppStatus st = ippsSinCos_32f_A21( x, y1, y2, 4 );

      printf(" ippsSinCos_32f_A21:\n");
      printf(" x  = %.3f %.3f %.3f %.3f \n", x[0],  x[1],  x[2],
x[3]);
      printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2],
y1[3]);
      printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2],
y2[3]);
      return st;
}

Output results:

 ippsSinCos_32f_A21:
 x  = 3857.845 -3939.024 -1468.856 -8592.486
 y1 = -0.031 0.508 0.987 0.228
 y2 = 1.000 0.861 0.161 -0.974
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pDst1* or *pDst2* or *pSrc* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the *pSrc* elements is equal to ± INF.

Tan

Computes tangent of each vector element.

Syntax

```
IppStatus ippsTan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsTan_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsTan_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsTan_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsTan_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsTan_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsTan` is declared in the `ippvm.h` file. This function computes the tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsTan_32f_A11` and `ippsTan_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsTan_32f_A21` and `ippsTan_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors ippsTan_32f_A24 and ippsTan_32fc_A24 guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors ippsTan_64f_A50 and ippsTan_64fc_A50 guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors ippsTan_64f_A53 and ippsTan_64fc_A53 guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows: $pDst[n] = \tan(pSrc[n])$, $0 \le n < len$.

Example 12-16 shows how to use the function ippsTan.

Example 12-16Using ippsTan Function

```
IppStatus ippsTan_32f_A21_sample(void)
{
const Ipp32f x[4] = {7519.456, 4533.524, 9118.015, 8514.359};
Ipp32f          y[4];

      IppStatus st = ippsTan_32f_A21( x, y, 4 );

      printf(" ippsTan_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsTan_32f_A21:
 x = 7519.456 4533.524 9118.015 8514.359
 y = -18.656 0.209 2.028 0.750
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

Acos

Computes inverse cosine of each vector element.

Syntax

```
IppStatus ippsAcos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsAcos_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAcos_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAcos_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAcos_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsAcos_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsAcos` is declared in the `ippvm.h` file. This function computes the inverse cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAcos_32f_A11` and `ippsAcos_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAcos_32f_A21` and `ippsAcos_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAcos_32f_A24` and `ippsAcos_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAcos_64f_A50` and `ippsAcos_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAcos_64f_A53` and `ippsAcos_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \mathrm{acos}(pSrc[n]),\ 0 \le n < len.$$

Example 12-17 shows how to use the function `ippsAcos`.

Example 12-17 Using `ippsAcos` Function

```
IppStatus ippsAcos_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.079, -0.715, -0.076, -0.529};
Ipp32f          y[4];

      IppStatus st = ippsAcos_32f_A21( x, y, 4 );

      printf(" ippsAcos_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsAcos_32f_A21:
 x = 0.079 -0.715 -0.076 -0.529
 y = 1.492 2.368 1.647 2.129
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has an absolute value greater than 1.

Asin

Computes inverse sine of each vector element.

Syntax

```
IppStatus ippsAsin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsAsin_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAsin_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAsin_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAsin_64fc_A50 (const Ipp64fc* pSrc, Ipp64cf* pDst, int len);
IppStatus ippsAsin_64fc_A53 (const Ipp64fc* pSrc, Ipp64cf* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsAsin` is declared in the `ippvm.h` file. This function computes the inverse sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAsin_32f_A11` and `ippsAsin_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAsin_32f_A21` and `ippsAsin_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAsin_32f_A24` and `ippsAsin_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAsin_64f_A50` and `ippsAsin_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAsin_64f_A53` and `ippsAsin_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows: $pDst[n] = \mathrm{asin}(pSrc[n])$, $0 \le n < len$.

Example 12-18 shows how to use the function `ippsAsin`.

Example 12-18Using `ippsAsin` Function

```
IppStatus ippsAsin_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.724, -0.581, 0.559, 0.687};
Ipp32f          y[4];

      IppStatus st = ippsAsin_32f_A21( x, y, 4 );

      printf(" ippsAsin_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}


Output results:

 ippsAsin_32f_A21:
 x = 0.724 -0.581 0.559 0.687
 y = 0.810 -0.620 0.594 0.758
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has an absolute value greater than 1.

Atan

Computes inverse tangent of each vector element.

Syntax

```
IppStatus ippsAtan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsAtan_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAtan_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAtan_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAtan_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsAtan_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsAtan` is declared in the `ippvm.h` file. This function computes the inverse tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAtan_32f_A11` and `ippsAtan_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAtan_32f_A21` and `ippsAtan_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAtan_32f_A24` and `ippsAtan_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAtan_64f_A50` and `ippsAtan_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAtan_64f_A53` and `ippsAtan_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \mathrm{atan}(pSrc[n]),\ 0 \le n < len.$$

Example 12-19 shows how to use the function `ippsAtan`.

Example 12-19Using `ippsAtan` Function

```
IppStatus ippsAtan_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.994, 0.999, 0.223, -0.215};
Ipp32f          y[4];

      IppStatus st = ippsAtan_32f_A21( x, y, 4 );

      printf(" ippsAtan_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsAtan_32f_A21:
 x = 0.994 0.999 0.223 -0.215
 y = 0.782 0.785 0.219 -0.212
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

```
IppStatus ippsAtan2_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
IppStatus ippsAtan2_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
```

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsAtan2` is declared in the `ippvm.h` file. This function computes the angle between the `X` axis and the line from the origin to the point `(X,Y)`, for each element of *pSrc1* as a `Y` (the ordinate) and corresponding element of *pSrc2* as an `X` (the abscissa), and stores the result in the corresponding element of *pDst*. The result angle varies from -π to + π.

For single precision data:

function flavor `ippsAtan2_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAtan2_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsAtan2_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAtan2_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAtan2_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \arctan2(pSrc1[n], pSrc2[n]) \quad , 0 \le n < len .$$

Example 12-20 shows how to use the function `ippsAtan2`.

Example 12-20Using `ippsAtan2` Function

```
IppStatus ippsAtan2_32f_A21_sample(void)
{
const Ipp32f x1[4] = {1.492, 1.700, 1.147, 1.142};
const Ipp32f x2[4] = {1.064, 1.505, 1.950, 1.905};
Ipp32f          y[4];

      IppStatus st = ippsAtan2_32f_A21( x1, x2, y, 4 );

      printf(" ippsAtan2_32f_A21:\n");
      printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2],
x1[3]);
      printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2],
x2[3]);
      printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],
y[3]);
      return st;
}


Output results:

 ippsAtan2_32f_A21:
 x1 = 1.492 1.700 1.147 1.142
 x2 = 1.064 1.505 1.950 1.905
 y  = 0.951 0.846 0.532 0.540
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Hyperbolic Functions

Cosh

Computes hyperbolic cosine of each vector element.

Syntax

```
IppStatus ippsCosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsCosh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCosh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCosh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCosh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsCosh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsCosh` is declared in the `ippvm.h` file. This function computes the hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsCosh_32f_A11` and `ippsCosh_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsCosh_32f_A21` and `ippsCosh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsCosh_32f_A24` and `ippsCosh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsCosh_64f_A50` and `ippsCosh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsCosh_64f_A53` and `ippsCosh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \cosh(pSrc[n]),\ 0 \le n < len.$$

Example 12-21 shows how to use the function `ippsCosh`.

Example 12-21Using `ippsCosh` Function

```
IppStatus ippsCosh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-4.676, -4.054, 6.803, -9.525};
Ipp32f      y[4];

      IppStatus st = ippsCosh_32f_A21( x, y, 4 );

      printf(" ippsCosh_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}


Output results:

 ippsCosh_32f_A21:
 x = -4.676 -4.054 6.803 -9.525
 y = 53.661 28.833 450.219 6849.870
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

IppStsOverflow	In real functions, indicates a warning that the function overflows, that is, at least one of elements of *pSrc* has the absolute value greater than Ln(FPMAX) + Ln(2), where FPMAX is the maximum representable floating-point number.

Sinh

Computes hyperbolic sine of each vector element.

Syntax

```
IppStatus ippsSinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsSinh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSinh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSinh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSinh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSinh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function ippsSinh is declared in the ippvm.h file. This function computes the hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors ippsSinh_32f_A11 and ippsSinh_32fc_A11 guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsSinh_32f_A21` and `ippsSinh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsSinh_32f_A24` and `ippsSinh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSinh_64f_A50` and `ippsSinh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsSinh_64f_A53` and `ippsSinh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sinh(pSrc[n]),\ 0 \le n < len$.

Example 12-22 shows how to use the function `ippsSinh`.

Example 12-22Using `ippsSinh` Function

```
IppStatus ippsSinh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-2.483, -8.148, 3.544, -8.876};
Ipp32f      y[4];

      IppStatus st = ippsSinh_32f_A21( x, y, 4 );

      printf(" ippsSinh_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsSinh_32f_A21:
 x = -2.483 -8.148 3.544 -8.876
 y = -5.945 -1727.412 17.290 -3577.970
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

IppStsOverflow	In real functions, indicates a warning that the function overflows, that is, at least one of elements of *pSrc* has the absolute value greater than Ln(FPMAX) + Ln(2), where FPMAX is the maximum representable floating-point number.

Tanh

Computes hyperbolic tangent of each vector element.

Syntax

```
IppStatus ippsTanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsTanh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsTanh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsTanh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsTanh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsTanh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function ippsTanh is declared in the ippvm.h file. This function computes the hyperbolic tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsTanh_32f_A11` and `ippsTanh_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsTanh_32f_A21` and `ippsTanh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsTanh_32f_A24` and `ippsTanh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsTanh_64f_A50` and `ippsTanh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsTanh_64f_A53` and `ippsTanh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[x] = \tanh(pSrc[n]),\ 0 \le n < len.$$

Example 12-23 shows how to use the function `ippsTanh`.

Example 12-23Using `ippsTanh` Function

```
IppStatus ippsTanh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f      y[4];

      IppStatus st = ippsTanh_32f_A21( x, y, 4 );

      printf(" ippsTanh_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}


Output results:

 ippsTanh_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Acosh

Computes inverse hyperbolic cosine of each vector element.

Syntax

```
IppStatus ippsAcosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsAcosh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAcosh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAcosh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAcosh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsAcosh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsAcosh` is declared in the `ippvm.h` file. This function computes the inverse (nonnegative) hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAcosh_32f_A11` and `ippsAcosh_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAcosh_32f_A21` and `ippsAcosh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAcosh_32f_A24` and `ippsAcosh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAcosh_64f_A50` and `ippsAcosh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAcosh_64f_A53` and `ippsAcosh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[x] = \operatorname{acosh}(pSrc[n]),\ 0 \le n < len.$$

Example 12-24 shows how to use the function `ippsAcosh`.

Example 12-24Using `ippsAcosh` Function

```
IppStatus ippsAcosh_32f_A21_sample(void)
{
const Ipp32f x[4] = {588.321, 691.492, 837.773, 726.767};
Ipp32f      y[4];

      IppStatus st = ippsAcosh_32f_A21( x, y, 4 );

      printf(" ippsAcosh_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsAcosh_32f_A21:
 x = 588.321 691.492 837.773 726.767
 y = 7.070 7.232 7.424 7.282
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 1.

Asinh

Computes inverse hyperbolic sine of each vector element.

Syntax

```
IppStatus ippsAsinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsAsinh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAsinh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAsinh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAsinh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsAsinh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsAsinh` is declared in the `ippvm.h` file. This function computes the inverse hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAsinh_32f_A11` and `ippsAsinh_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAsinh_32f_A21` and `ippsAsinh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAsinh_32f_A24` and `ippsAsinh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAsinh_64f_A50` and `ippsAsinh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAsinh_64f_A53` and `ippsAsinh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \operatorname{asinh}(pSrc[n]),\ 0 \le n < len.$$

Example 12-25 shows how to use the function `ippsAsinh`.

Example 12-25Using `ippsAsinh` Function

```
IppStatus ippsAsinh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-30.122, -589.282, 487.472, -63.082};
Ipp32f      y[4];

      IppStatus st = ippsAsinh_32f_A21( x, y, 4 );

      printf(" ippsAsinh_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsAsinh_32f_A21:
 x = -30.122 -589.282 487.472 -63.082
 y = -4.099 -7.072 6.882 -4.838
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Atanh

Computes inverse hyperbolic tangent of each vector element.

Syntax

```
IppStatus ippsAtanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);

IppStatus ippsAtanh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAtanh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAtanh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsAtanh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsAtanh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsAtanh` is declared in the `ippvm.h` file. This function computes the inverse hyperbolic tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAtanh_32f_A11` and `ippsAtanh_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAtanh_32f_A21` and `ippsAtanh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAtanh_32f_A24` and `ippsAtanh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAtanh_64f_A50` and `ippsAtanh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAtanh_64f_A53` and `ippsAtanh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{atanh}(pSrc[n])$, $0 \le n < len$.

Example 12-26 shows how to use the function `ippsAtanh`.

Example 12-26Using `ippsAtanh` Function

```
IppStatus ippsAtanh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.076, 0.808, 0.440, -0.705};
Ipp32f      y[4];

      IppStatus st = ippsAtanh_32f_A21( x, y, 4 );

      printf(" ippsAtanh_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}

Output results:

 ippsAtanh_32f_A21:
 x = -0.076 0.808 0.440 -0.705
 y = -0.076 1.123 0.472 -0.877
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when len is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has absolute value greater than 1.

`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* has absolute value equal to 1.

Special Functions

Erf

Computes the error function value.

Syntax

```
IppStatus ippsErf_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsErf_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsErf` is declared in the `ippvm.h` file. This function computes the error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErf_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErf_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErf_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErf_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErf_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{erf}(pSrc[n])$, $0 \le n < len$, where $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

Example 12-27 shows how to use the function `ippsErf`.

Example 12-27Using `ippsErf` Function

```
IppStatus ippsErf_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f      y[4];

      IppStatus st = ippsErf_32f_A21( x, y, 4 );

      printf(" ippsErf_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}


Output results:

 ippsErf_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Erfc

Computes the complementary error function value.

Syntax

```
IppStatus ippsErfc_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsErfc_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsErfc` is declared in the `ippvm.h` file. This function computes the complementary error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErfc_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErfc_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErfc_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErfc_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErfc_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{erfc}(pSrc[n])$, $0 \le n < len$, where

$$\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt.$$

Example 12-28 shows how to use the function `ippsErfc`.

Example 12-28Using `ippsErfc` Function

```
IppStatus ippsErfc_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f      y[4];

      IppStatus st = ippsErfc_32f_A21( x, y, 4 );

      printf(" ippsErfc_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}


Output results:

 ippsErfc_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsUnderflow`	Indicates a warning that the function underflows, that is, at least one of *pSrc* elements is less than some threshold value, where the function result is less than the minimum positive floating-point value in target precision.

ErfInv

Computes the inverse error function value.

Syntax

```
IppStatus ippsErfInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsErfInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsErfInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsErfInv` is declared in the `ippvm.h` file. This function computes the inverse error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErfInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErfInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErfInv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErfInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErfInv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{erfinv}(pSrc[n])$, $0 \le n < len$, where $erfinv(x) = erf^{-1}(x)$

and $erf(x)$ denotes the error function defined as given by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

Example 12-28 shows how to use the function `ippsErfInv`.

Example 12-29Using `ippsErfInv` Function

```
IppStatus ippsErfInv_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.842, 0.638, -0.345, -0.774};
Ipp32f      y[4];

      IppStatus st = ippsErfInv_32f_A21( x, y, 4 );

      printf(" ippsErfInv_32f_A21:\n");
      printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
      printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
      return st;
}


Output results:

 ippsErfInv_32f_A21:
 x = -0.842 0.638 -0.345 -0.774
 y = -0.998 0.645 -0.316 -0.856
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDomain`	Indicates a warning that the the argument is out of the function domain, that is, at least one of *pSrc* elements has the absolute value greater than 1.
`ippStsSingularity`	Indicates a warning that the argument is a singularity point, that is, at least one of the elements of *pSrc* has the absolute value equal to 1.

Rounding Functions

Floor

Computes integer value rounded toward minus infinity for each vector element.

Syntax

```
IppStatus ippsFloor_32f (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsFloor_64f (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsFloor` is declared in the `ippvm.h` file. This function computes an integer value rounded towards minus infinity for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Example 12-30 shows how to use the `ippsFloor` function.

Example 12-30Using `ippsFloor` Function

```
IppStatus ippsFloor_32f_ sample(void)
{
    const Ipp32f x[4] = {-0.883, -0.265, 0.176, 0.752};
    Ipp32f y[4];
    IppStatus st = ippsFloor_32f ( x, y, 4 );
    printf(" ippsFloor_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:
ippsFloor_32f:
x = -0.883 -0.265 0.176 0.752
y = -1.000 -1.000 0.000 0.000
```

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Ceil

Computes integer value rounded toward plus infinity for each vector element.

Syntax

```
IppStatus ippsCeil_32f (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCeil_64f (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function ippsCeil is declared in the ippvm.h file. This function computes an integer value rounded towards plus infinity for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Example 12-31 shows how to use the ippsCeil function.

Example 12-31Using ippsCeil Function

```
IppStatus ippsCeil_32f_ sample(void)
{
    const Ipp32f x[4] = {-0.883, -0.265, 0.176, 0.752};
    Ipp32f y[4];
    IppStatus st = ippsCeil_32f ( x, y, 4 );
    printf(" ippsCeil_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:
```

Example 12-31Using `ippsCeil` Function (continued)

```
ippsCeil_32f:
x = -0.883 -0.265 0.176 0.752
y = 0.000 0.000 1.000 1.000
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Trunc

Computes integer value rounded toward zero for each vector element.

Syntax

```
IppStatus ippsTrunc_32f (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTrunc_64f (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsTrunc` is declared in the `ippvm.h` file. This function computes an integer value rounded towards zero for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Example 12-32 shows how to use the `ippsTrunc` function.

Example 12-32Using `ippsTrunc` Function

```
IppStatus ippsTrunc_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y[4];
    IppStatus st = ippsTrunc_32f ( x, y, 4 );
    printf(" ippsTrunc_32f:\n");
```

Example 12-32Using `ippsTrunc` Function (continued)

```
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:
ippsTrunc_32f:
x = -1.883 -0.265 0.176 1.752
y = -1.000 0.000 0.000 1.000
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Round

Computes integer value rounded to nearest for each vector element.

Syntax

```
IppStatus ippsRound_32f (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsRound_64f (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsRound` is declared in the `ippvm.h` file. This function computes a rounded to the nearest integer value for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*. Halfway values, that is, 0.5, -1.5, and the like, are rounded off away from zero, that is, 0.5 -> 1, -1.5 -> -2, etc.

Example 12-33 shows how to use the `ippsRound` function.

Example 12-33Using `ippsRound` Function

```
IppStatus ippsRound_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y[4];
    IppStatus st = ippsRound_32f ( x, y, 4 );
    printf(" ippsRound_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:
ippsRound_32f:
x = -1.883 -0.265 0.176 1.752
y = -2.000  0.000 0.000 2.000
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

NearbyInt

Computes rounded integer value in current rounding mode for each vector element.

Syntax

```
IppStatus ippsNearbyInt_32f (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsNearbyInt_64f (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsNearbyInt` is declared in the `ippvm.h` file. This function computes a rounded integer value in a current rounding mode for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Example 12-34 shows how to use the `ippsNearbyInt` function.

Example 12-34Using `ippsNearbyInt` Function

```
#include <fenv.h>

void ippsNearbyInt_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4], y2[4];

    fesetround(FE_TONEAREST);
    ippsNearbyInt_32f ( x, y2, 4 );

    fesetround(FE_TOWARDZERO);
    ippsNearbyInt_32f ( x, y2, 4 );

    printf(" ippsNearbyInt_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
}

Output results:
ippsNearInt_32f:
x = -1.883 -0.265 0.176 1.752
y = -2.000  0.000 0.000 2.000
y = -1.000  0.000 0.000 1.000
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Rint

Computes rounded integer value in current rounding mode for each vector element with inexact result exception raised for each changed value.

Syntax

```
IppStatus ippsRint_32f (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsRint_64f (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

The function `ippsRint` is declared in the `ippvm.h` file. This function computes a rounded integer value in a current rounding mode for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst* raising inexact result exception if the value has changed.

Example 12-35 shows how to use the `ippsRint` function.

Example 12-35Using `ippsRint` Function

```
#include <fenv.h>

void ippsRint_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4], y2[4];

    fesetround(FE_TONEAREST);
    ippsRint_32f ( x, y2, 4 );

    fesetround(FE_TOWARDZERO);
    ippsRint_32f ( x, y2, 4 );

    printf(" ippsRint_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
```

Example 12-35Using `ippsRint` Function (continued)

```
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
}

Output results:
ippsRint_32f:
x = -1.883 -0.265 0.176 1.752
y = -2.000  0.000 0.000 2.000
y = -1.000  0.000 0.000 1.000
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Modf

Computes truncated integer value and remaining fraction part for each vector element.

Syntax

```
IppStatus ippsModf_32f (const Ipp32f* pSrc, Ipp32f* pDst1, Ipp32f* pDst2, int
    len);
IppStatus ippsModf_64f (const Ipp64f* pSrc, Ipp64f* pDst1, Ipp64f* pDst2, int
    len);
```

Parameters

pSrc	Pointer to the source vector.
pDst1	Pointer to the first destination vector.
pDst2	Pointer to the second destination vector.
len	Number of elements in the vectors.

Description

The function `ippsModf` is declared in the `ippvm.h` file. This function computes a truncated value and a remainder of each element of the vector *pSrc*. The truncated integer value is stored in the corresponding element of the *pDst1* vector and the remainder is stored in the corresponding element of the *pDst2* vector.

Example 12-36 shows how to use the `ippsModf` function.

Example 12-36Using `ippsModf` Function

```
IppStatus ippsModf_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4] y2[4];
    IppStatus st = ippsModf_32f ( x, y1, y2, 4 );
    printf(" ippsModf_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
    return st;
}

Output results:
ippsModf_32f:
x = -1.883 -0.265 0.176 1.752
y1 = -1.000 0.000 0.000 1.000
y2 = -0.883 -0.265 0.176 0.752
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst1* or *pDst2* pointer is NULL.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Data Compression Functions

13

This chapter describes Intel® IPP functions for data compression that support a number of different compression methods: Huffman and variable-length coding, dictionary-based coding methods (including support of ZLIB compression), and methods based on Burrows-Wheeler Transform.

These functions are grouped into the following sections:

VLC and Huffman Coding Functions
Dictionary-Based Compression Functions
IBWT-Based Compression Functions

Each section starts with a table that lists functions described in more detail later in this section, together with the brief description of operations that these functions perform.

Application Notes

- The functions in this domain can be divided into two types: the functions that actually compress data, and transformation functions. The latters do not compress data but only modify them and prepare for further compression. The examples of such transformation are the Burrows-Weller Transform, or MoveToFront algorithm. To do data compression efficient, you should develop the proper consequence of functions of different type that will transform data and then compress them.
- Compression ratio depends on the statistics of input data. For some types of input data no compression could be achieved at all.
- The size of memory required for the output of data compression functions typically is not obvious. As a rule encoding functions uses less memory for output than the size of the input buffer, on the contrary decoding functions use more memory for output than the size of the input buffer. You should account these issues and allocate the proper

quantity of output memory using the techniques provided by functions in this domain. For example, you can use a double-pointer technique for automatic shifting the user submitted pointer. For some other functions it is possible to compute the upper limit of the size of the required output buffer.

VLC and Huffman Coding Functions

This section describes the Intel IPP functions for Huffman and variable-length coding.

The full list of these functions is given in Table 13-1.

Table 13-1 Intel IPP VLC and Huffman Coding Functions

Function Short Name	Description
VLC Functions	
VLCEncodeInitAlloc	Allocates memory and initializes the VLC encoder structure.
VLCEncodeFree	Frees memory allocated for the VLC encoder structure.
VLCEncodeInit	Initializes the VLC encoder structure.
VLCEncodeGetSize	Computes the size of the VLC encode structure.
VLCEncodeBlock	Performs VLC encoding of a block of data.
VLCEncodeOne	Performs VLC encoding of a single element.
VLCCountBits	Computes a number of bits required to encode the source block.
VLCDecodeInitAlloc	Allocates memory and initializes the VLC decoder structure.
VLCDecodeFree	Frees memory allocated for the VLC decoder structure.
VLCDecodeInit	Initializes the VLC decoder structure.
VLCDecodeGetSize	Computes the size of the VLC decoder structure.
VLCDecodeBlock	Decodes a block of VLC encoded data.
VLCDecodeOne	Decodes a single VLC encoded element.
VLCDecodeUTupleInitAlloc	Allocates memory and initializes the VLCDecodeUTuple structure based on the input parameters.
VLCDecodeUTupleFree	Frees memory allocated for the VLCDecodeUTuple structure.
VLCDecodeUTupleInit	Initializes the VLCDecodeUTuple structure based on the input parameters.
VLCDecodeUTupleGetSize	Computes the size of the VLCDecodUTuple structure.
VLCDecodeUTupleBlock	Decodes a block of VLC encoded data.
VLCDecodeUTupleOne	Decodes a single VLC encoded tuple.
Huffman Encoding and Decoding Functions	
EncodeHuffInitAlloc	Allocates memory and initializes the Huffman encoding structure.
HuffFree	Frees memory allocated for the Huffman coding state structures.

Table 13-1 Intel IPP VLC and Huffman Coding Functions (continued)

Function Short Name	Description
EncodeHuffInit	Initializes the Huffman encoding structure.
HuffGetSize	Computes size of the external buffer for the Huffman encoding structure.
EncodeHuffOne	Performs Huffman encoding of a single element.
EncodeHuff	Performs Huffman encoding.
EncodeHuffFinal	Performs Huffman encoding of the remainder.
HuffGetLenCodeTable	Returns the table containing lengths of Huffman codes.
DecodeHuffInitAlloc	Allocates memory and initializes the Huffman decoding structure.
DecodeHuffInit	Initializes the Huffman decoding structure.
DecodeHuffOne	Performs Huffman decoding of a single code word.
DecodeHuff	Performs Huffman decoding.
HuffGetDstBuffSize	Computes the size of a destination buffer for Huffman encoding and decoding by blocks
HuffLenCodeTablePack	Packs the table containing lengths of Huffman codes.
HuffLenCodeTableUnpack	Unpacks the table containing lengths of Huffman codes.

Data Compression VLC functions

Variable Length Coding (VLC) is a data compression method that uses statistical modeling to define which values occur more frequently than others to build tables for subsequent encoding and decoding operations. Data in the bitstream is encoded with VLC table so that the shortest codes correspond to the most frequent values and the longer codes correspond to the less frequent values Intel IPP functions use the structure `IppsVLCTable` for storing the VLC table. The Table 13-2 lists the fields of this structure.

Table 13-2 IppsVLCTable Structure

```
struct VLCTable_32s
{
   Ipp32s value,                    The current code.
   Ipp32s code,                     The actual bit code for the value.
   Ipp32s length;                   The bit length of the value.
```

Table 13-2 IppsVLCTable Structure

```
}
typedef struct VLCTable_32s IppsVLCTable_32s
```

Fill in the table with codes and code lengths for values, specifying them in the corresponding fields of the `IppsVLCTable` structure. The `value` can be both positive and negative. The `code` denotes the `length` least significant bits in binary representation of the code value. Thus, valid lengths are within the range of 0 to 32. Note that you are responsible for specifying correct tables with prefix codes for values specified in the `value` field of the `IppsVLCTable` structure.

VLCEncodeInitAlloc

Allocates memory and initializes the VLC encoder structure.

Syntax

```
ippsVLCEncodeInitAlloc_32s (const IppsVLCTable_32s* pInputTable, int
    inputTableSize, IppsVLCEncodeSpec_32s** ppVLCSpec);
```

Parameters

pInputTable	Pointer to the VLC table.
inputTableSize	Size of the VLC table.
ppVLCSpec	Double pointer to the VLC encoder specification structure.

Description

The function `ippsVLCEncodeInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the VLC encoder specification structure *ppVLCSpec* that uses the VLC table *pInputTable*. The VLC encoder specification structure is used by the functions `ippsVLCEncodeBlock` and `ippsVLCEncodeOne`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.

ippStsMemAllocErr	Indicates an error if memory allocation fails.
ippStsVLCInputDataErr	Indicates an error if *inputTableSize* is less than or equal to 0.

VLCEncodeFree

Frees memory allocated for the VLC encoder structure.

Syntax

```
void ippsVLCEncodeFree_32s (IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pVLCSpec	Pointer to the VLC encoder specification structure.

Description

The function `ippsVLCEncodeFree` is declared in the `ippdc.h` file. This function frees memory allocated for the VLC encode structure by the function `ippsVLCEncodeInitAlloc`.

VLCEncodeInit

Initializes the VLC encoder structure.

Syntax

```
ippsVLCEncodeInit_32s (const IppsVLCTable_32s* pInputTable, int inputTableSize,
    IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pInputTable	Pointer to the VLC table
inputTableSize	Size of the VLC table
pVLCSpec	Pointer to the VLC encoder specification structure.

Description

The function `ippsVLCEncodeInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the VLC encoder specification structure *pVLCSpec* that uses the VLC table *pInputTable*. The size of the external buffer should be computed previously by calling the function `ippsVLCEncodeGetSize`. Alternatively, the VLC encoder structure can be initialized by the function `ippsVLCEncodeInitAlloc`.

The VLC encoder specification structure is used by the functions `ippsVLCEncodeBlock` and `ippsVLCEncodeOne`.

NOTE. The parameters *pInputTable* and *inputTableSize* of the `ippsVLCEncodeInit` and `ippsVLCEncodeGetSize` functions must have equal respective values to avoid an error in subsequent encoding functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsVLCInputDataErr`	Indicates an error if *inputTableSize* is less than or equal to 0.

VLCEncodeGetSize

Computes the size of the VLC encoder structure.

Syntax

```
ippsVLCEncodeGetSize_32s (const IppsVLCTable_32s* pInputTable, int
    inputTableSize, Ipp32s* pSize);
```

Parameters

pInputTable	Pointer to the VLC table
inputTableSize	Size of the input table
pSize	Pointer to the size of the VLC encoder specification structure.

Description

The function `ippsVLCEncodeGetSize` is declared in the `ippdc.h` file. This function computes the size of the VLC encoder structure that uses the VLC table *pInputTable*. Its size is stored in the *pSize*. This function should be called prior to the function `ippsVLCEncodeInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsVLCInputDataErr`	Indicates an error if *inputTableSize* is less than or equal to 0.

VLCEncodeBlock

Performs VLC encoding of a block of data.

Syntax

```
ippsVLCEncodeBlock_16s1u (const Ipp16s* pSrc, int srcLen, Ipp8u** ppDst, int*
    pDstBitsOffset, const IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pSrc	Pointer to the source buffer.
srcLen	Number of elements in the *pSrc*.
ppDst	Double pointer to the destination bitstream.
pDstBitsOffset	Pointer to the input/output bit position in the *pDst*.
pVLCSpec	Pointer to the VLC encoder specification structure.

Description

The function `ippsVLCEncodeBlock` is declared in the `ippdc.h` file. This function encodes *srcLen* elements of the source vector *pSrc* and stores the result in the destination buffer *pDst*. After encoding the function shifts the *pDst* pointer by the number of successfully written bytes. The function updates *pDstBitsOffset* setting it to the actual bit position in the destination buffer.

The ippsVLCEncodeBlock function uses the VLC table from the encoder specification structure *pVLCSpec* that must be initialized by the functions ippsVLCEncodeInitAlloc or ippsVLCEncodeInit beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsVLCInputDataErr	Indicates an error if incorrect input is used.

VLCEncodeOne

Performs VLC encoding of a single element.

Syntax

```
ippsVLCEncodeOne_16s1u (Ipp16s src, Ipp8u** ppDst, int* pDstBitsOffset, const
    IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

src	Source value.
ppDst	Double pointer to the destination bitstream.
pDstBitsOffset	Pointer to the input/output bit position in the *pDst*.
pVLCSpec	Pointer to the VLC encoder specification structure.

Description

The function ippsVLCEncodeOne is declared in the ippdc.h file. This function encodes a single element *src* and stores the result in the destination buffer *pDst*. After encoding the function shifts the *pDst* pointer by the number of successfully written bytes. The function updates *pDstBitsOffset* setting it to the actual bit position in the destination buffer.

The ippsVLCEncodeBlock function uses the VLC table from the encoder specification structure *pVLCSpec* that must be initialized by the functions ippsVLCEncodeInitAlloc or ippsVLCEncodeInit beforehand.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsVLCInputDataErr	Indicates an error if incorrect input is used.

VLCCountBits

Computes a number of bits required to encode the source block.

Syntax

```
ippsVLCCountBits_16s32s (const Ipp16s* pSrc, int srcLen, Ipp32s* pCountBits,
    const IppsVLCEncodeSpec_32s* pVLCSpec);
```

Parameters

pSrc	Pointer to the source buffer.
srcLen	Number of elements in the *pSrc*.
pCounBits	Pointer to the computed length in bits required to encode *pSrc*.
pVLCSpec	Pointer to the VLC encoder specification structure.

Description

The function ippsVLCCountBits is declared in the ippdc.h file. This function computes a number of bits required to encode the source data *pSrc* using the VLC table from the encoder specification structure *pVLCSpec* that must be initialized by the functions ippsVLCEncodeInit or ippsVLCEncodeInitAlloc beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

VLCDecodeInitAlloc

Allocates memory and initializes the VLC decoder structure.

Syntax

```
ippsVLCDecodeInitAlloc_32s (const IppsVLCTable_32s* pInputTable,
    int inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables,
    IppsVLCDecodeSpec_32s** ppVLCSpec);
```

Parameters

pInputTable	Pointer to the VLC table.
inputTableSize	Size of the VLC table.
pSubTablesSizes	Pointer to the sizes of the subtables.
numSubTables	Number of the subtables.
ppVLCSpec	Double pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the VLC decoder structure *ppVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

The VLC decoder specification structure is used by the functions ippsVLCDecodeBlock and ippsVLCDecodeOne.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsMemAllocErr	Indicates an error if memory allocation fails.
ippStsVLCUsrTblCodeLengthErr	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in

	pSubTablesSizes is less than 0; or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeFree

Frees memory allocated for the VLC decoder structure.

Syntax

```
void ippsVLCDecodeFree_32s (IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

pVLCSpec	Pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeFree` is declared in the `ippdc.h` file. This function frees memory allocated for the VLC decoder structure by the function `ippsVLCDecodeInitAlloc`.

VLCDecodeInit

Initializes the VLC decoder structure.

Syntax

```
ippsVLCDecodeInit_32s (const IppsVLCTable_32s* pInputTable, int inputTableSize,
    Ipp32s* pSubTablesSizes, int numSubTables, IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

pInputTable	Pointer to the VLC table.
inputTableSize	Size of the VLC table.
pSubTablesSizes	Pointer to the sizes of the subtables.
numSubTables	Number of the subtables.
pVLCSpec	Pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the VLC decoder structure *pVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.
The size of the external buffer should be computed previously by calling the function `ippsVLCDecodeGetSize`. Alternatively, the VLC decoder structure can be initialized by the function `ippsVLCDecodeInitAlloc`.

The VLC decoder specification structure is used by the functions `ippsVLCDecodeBlock` and `ippsVLCDecodeOne`.

NOTE. The parameters *pInputTable* and *inputTableSize* of the `ippsVLCDecodeInit` and `ippsVLCDecodeGetSize` functions must have equal respective values to avoid an error in subsequent decoding functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsVLCUsrTblCodeLengthErr`	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in *pSubTablesSizes* is less than 1; or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeGetSize

Computes the size of the VLC decoder structure.

Syntax

```
ippsVLCDecodeGetSize_32s (const IppsVLCTable_32s* pInputTable, int
    inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables, Ipp32s* pSize);
```

Parameters

pInputTable	Pointer to the input table.
inputTableSize	Size of the input table.
pSubTablesSizes	Pointer to the sizes of the subtables.
numSubTables	Number of the subtables.
pVLCSpec	Pointer to the size of the VLC decoder specification structure.

Description

The function `ippsVLCDecodeGetSize` is declared in the `ippdc.h` file. This function computes the size of the VLC decoder structure *ppVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.
This function should be called prior to the function `ippsVLCDecodeInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsVLCUsrTblCodeLengthErr`	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in *pSubTablesSizes* is less than 0; or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeBlock

Decodes a block of VLC encoded data.

Syntax

```
ippsVLCDecodeBlock_1u16s (const Ipp8u** ppSrc, int* pSrcBitsOffset, Ipp16s*
    pDst, int dstLen, const IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

ppSrc	Double pointer to the source bitstream.
pSrcBitsOffset	Pointer to the input/output bit position.
pDst	Pointer to the destination buffer.
dstLen	Number of elements in the *pDst*.
pVLCSpec	Pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeBlock` is declared in the `ippdc.h` file. This function decodes *dstLen* VLC encoded elements in the source bitstream starting from the *pSrcBitsOffset* bit position. The function stores the result in the destination buffer *pDst*. After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates *pSrcBitsOffset* setting it to the actual bit position in the source buffer.

The function `ippsVLCDecodeBlock` uses the VLC table from the decoder specification structure *pVLCSpec* that must be initialized by the functions `ippsVLCDecodeInitAlloc` or `ippsVLCDecodeInit` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsVLCInputDataErr`	Indicates an error if incorrect input is used. It can indicate that bitstream contains code that is not specified in the table.
`ippStsBitOffsetErr`	Indicates an error if value of the *pSrcBitsOffset* is less than 0 or greater than 7.

ippStsContextMatchErr	Indicates an error if the *pVLCSpec* structure is not initialized by the function ippsVLCDecodeInitAlloc or ippsVLCDecodeInit.

VLCDecodeOne

Decodes a single VLC encoded element.

Syntax

```
ippsVLCDecodeOne_1u16s (const Ipp8u** ppSrc, int* pSrcBitsOffset, Ipp16s* pDst,
    const IppsVLCDecodeSpec_32s* pVLCSpec);
```

Parameters

ppSrc	Double pointer to the source bitstream.
pSrcBitsOffset	Pointer to the input/output bit position.
pDst	Pointer to the decoded value.
pVLCSpec	Pointer to the VLC decoder specification structure.

Description

The function ippsVLCDecodeOne is declared in the ippdc.h file. This function decodes a single VLC encoded element in the *ppSrc* on position *pSrcBitsOffset*. The decoded element is stored in the destination buffer *pDst*.

After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates *pSrcBitsOffset* setting it to the actual bit position in the source buffer.

The function ippsVLCDecodeOne uses the VLC table from the decoder specification structure *pVLCSpec* that must be initialized by the functions ippsVLCDecodeInitAlloc or ippsVLCDecodeInit beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

ippStsVLCInputDataErr	Indicates an error if incorrect input is used. For decoding functions it can indicate that bitstream contains code that is not specified in the table.
ippStsBitOffsetErr	Indicates an error if value of the *pSrcBitsOffset* is less than 0 or greater than 7.
ippStsContextMatchErr	Indicates an error if the *pVLCSpec* structure is not initialized by the function ippsVLCDecodeInitAlloc or ippsVLCDecodeInit.

Example of Using VLC Coding Functions

The functions ippsVLCCountBits and ippsVLCEncodeBlock can be simply used in the wide range of different encoders (for example, for audio data). For example, if you need to quantize some vectors and encode them using no more bits than specified, these functions can be used as follows:

- previously the table *inputTable* of the size *inputTableSize* should be created; this table is of the following format:

```
static IppsVLCTable_32s inputTable[]=
{
{value0, code0, length0};
{value1, code1, length1};
...
{valueN, codeN, lengthN};
}
```

- initialization step:

```
ippsVLCEncodeGetSize_32s(InputTable, inputTableSize, &Size);
pVLCSpec = ippsMalloc_8u(Size); /* memory allocation */
ippsVLCEncodeInit_32s(pInputTable, inputTableSize, pVLCSpec);
...
```

- counting bits step:

```
Do {
//A some quantization algorithm should be here. Let pSrc is output of
this algorithm.
...
ippsVLCCountBits_16s32s(pSrc, srcLen, &bitNumber, pVLCSpec);
} while (bitNumber > allowedBitNumber);
```

- encoding bit stream step:

```
ippsVLCEncodeBlock_16s1u (pSrc, srcLen, ppBitSteam,
pBitStreamBitsOffset, pVLCSpec);
```

VLCDecodeUTupleInitAlloc

Allocates memory and initializes the VLCDecodeUTuple structure based on the input parameters.

Syntax

```
IppStatus
ippsVLCDecodeUTupleInitAlloc_32s(const IppsVLCTable_32s* pInputTable,
    int inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables, int
    numElements, int numValueBit, IppsVLCDecodeUTupleSpec_32s**
    ppVLCSpec);
```

Parameters

pInputTable	Pointer to the input VLC table.
inputTableSize	Size of the input VLC table.
pSubTablesSizes	Pointer to the sizes of the subtables.
numSubTables	Number of the subtables.
numElements	Number of the elements in the tuple.
numValueBit	Size of one element in bits.
ppVLCSpec	Double pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeUTupleInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the VLC decoder structure *ppVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*, number of elements in tuple *numElements*, and size of one element *numValueBit*.

Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

The VLC decoder specification structure is used by the functions `ippsVLCDecodeUTupleBlock` and `ippsVLCDecodeUTupleOne`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsMemAllocErr	Indicates an error if memory allocation fails.
ippStsVLCUsrTblCodeLengthErr	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in *pSubTablesSizes* is less than 0; or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeUTupleFree

Frees memory allocated for the VLC decoder structure.

Syntax

```
void ippsVLCDecodeUTupleFree_32s(IppsVLCDecodeUTupleSpec_32s* pVLCSpec);
```

Parameters

pVLCSpec	Pointer to the VLC decoder specification structure.

Description

The function ippsVLCDecodeUTupleFree is declared in the ippdc.h file. This function frees memory allocated for the VLC decoder structure by the function ippsVLCDecodeUTupleInitAlloc.

VLCDecodeUTupleInit

Initializes the VLCDecodeUTuple structure.

Syntax

```
IppStatus ippsVLCDecodeUTupleInit_32s(const IppsVLCTable_32s*
    pInputTable, int inputTableSize, Ipp32s* pSubTablesSizes, int
    numSubTables, int numElements, int numValueBit,
    IppsVLCDecodeUTupleSpec_32s* pVLCSpec);
```

Parameters

pInputTable	Pointer to the input VLC table.
inputTableSize	Size of the input VLC table.
pSubTablesSizes	Pointer to the sizes of the subtables.
numSubTables	Number of the subtables.
numElements	Number of the elements in the tuple.
numValueBit	Size of one element in bits.
pVLCSpec	Pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeUTupleInit` is declared in the `ippdc.h` file. This function initializes the VLC decoder structure with U-tuple *pVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*, number of elements in tuple *numElements*, and size of one element *numValueBit*.

Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

The VLC decoder specification structure is used by the functions `ippsVLCDecodeUTupleBlock` and `ippsVLCDecodeUTupleOne`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.

`ippStsVLCUsrTblCodeLengthErr`	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in *pSubTablesSizes* is less than 0; or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeUTupleGetSize

Computes the size of the VLCDecodUTuple structure.

Syntax

```
IppStatus ippsVLCDecodeUTupleGetSize_32s(const IppsVLCTable_32s*
    pInputTable, int inputTableSize, Ipp32s* pSubTablesSizes, int
    numSubTables, int numElements, int numValueBit, Ipp32s* pSize);
```

Parameters

pInputTable	Pointer to the input VLC table.
inputTableSize	Size of the input VLC table.
pSubTablesSizes	Pointer to the sizes of the subtables.
numSubTables	Number of the subtables.
numElements	Number of the elements in the tuple.
numValueBit	Size of one element in bits.
pSize	Pointer to the size of VLC decoder specification structure.

Description

The function `ippsVLCDecodeUTupleGetSize` is declared in the `ippdc.h` file. This function computes the size of the VLC decoder structure with U-tuple that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*, number of elements in tuple *numElements*, and size of one element *numValueBit*.

Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

This function should be called prior to the function `ippsVLCDecodeUTupleInit`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsVLCUsrTblCodeLengthErr	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in *pSubTablesSizes* is less than 0; or if the sum of these values is less than the maximum code length in the input table.

VLCDecodeUTupleBlock

Decodes a block of VLC encoded data.

Syntax

```
IppStatus ippsVLCDecodeUTupleBlock_1u16s(Ipp8u** ppSrc, int*
    pSrcBitsOffset, Ipp16s* pDst, int dstLen, const
    IppsVLCDecodeUTupleSpec_32s* pVLCSpec);
```

Parameters

ppSrc	Double pointer to the source bitstream.
pSrcBitsOffset	Pointer to the input/output bit position.
pDst	Pointer to the destination buffer.
dstLen	Number of elements in the *pDst*.
pVLCSpec	Pointer to the VLC decoder specification structure.

Description

The function ippsVLCDecodeUTupleBlock is declared in the ippdc.h file. This function decodes *dstLen* VLC encoded elements in the source bitstream starting from the *pSrcBitsOffset* bit position. The function stores the result in the destination buffer *pDst*. After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates *pSrcBitsOffset* setting it to the actual bit position in the source buffer.

The function `ippsVLCDecodeUTupleBlock` uses the VLC table from the decoder specification structure *pVLCSpec* that must be initialized by the functions `ippsVLCDecodeUTupleInitAlloc` or `ippsVLCDecodeUTupleInit` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsVLCInputDataErr`	Indicates an error if incorrect input is used. It can indicate that bitstream contains code that is not specified in the table.
`ippStsBitOffsetErr`	Indicates an error if value of the *pSrcBitsOffset* is less than 0 or greater than 7.
`ippStsContextMatchErr`	Indicates an error if the *pVLCSpec* structure is not initialized by the function `ippsVLCDecodeUTupleInitAlloc` or `ippsVLCDecodeUTupleInit`.

VLCDecodeUTupleOne

Decodes a single VLC encoded tuple.

Syntax

```
IppStatus ippsVLCDecodeUTupleOne_1u16s(Ipp8u** ppSrc, int*
    pSrcBitsOffset, Ipp16s* pDst, const IppsVLCDecodeUTupleSpec_32s*
    pVLCSpec);
```

Parameters

ppSrc	Double pointer to the source bitstream.
pSrcBitsOffset	Pointer to the input/output bit position.
pDst	Pointer to the decoded value.
pVLCSpec	Pointer to the VLC decoder specification structure.

Description

The function `ippsVLCDecodeUTupleOne` is declared in the `ippdc.h` file. This function decodes a single VLC encoded element (cotaining *numElements* samples) in the *ppSrc* on position *pSrcBitsOffset*. The decoded element is stored in the destination buffer *pDst*.

After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates *pSrcBitsOffset* setting it to the actual bit position in the source buffer.

The function `ippsVLCDecodeUTupleOne` uses the VLC table from the decoder specification structure *pVLCSpec* that must be initialized by the functions `ippsVLCDecodeUTupleInit` or `ippsVLCDecodeUTupleInitAlloc` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsVLCInputDataErr`	Indicates an error if incorrect input is used. For decoding functions it can indicate that bitstream contains code that is not specified in the table.
`ippStsBitOffsetErr`	Indicates an error if value of the *pSrcBitsOffset* is less than 0 or greater than 7.
`ippStsContextMatchErr`	Indicates an error if the *pVLCSpec* structure is not initialized by the function `ippsVLCDecodeUTupleInit` or `ippsVLCDecodeUTupleInitAlloc`.

Huffman Coding

This section describes functions implementing the Huffman coding algorithm. This algorithm performs so called entropy encoding. The main idea of this method is to substitute the code words (Huffman codes) instead of symbols. The least length codes correspond to the most often occurring symbols allowing to perform compression. For example, the sequence “aaaaaaaa” contains only one symbol ‘a’ that corresponds to the Huffman code with size 1 bit; after encoding the output sequence will have size 1 byte - thus compression ratio will be 8.

Limitations and conventions.

Intel IPP functions implement the Huffman static encoding. This means that encoding is a two-passes operation. In the first pass the probability structure of the source data is determined, then in the second pass the encoding itself is performed. The table of Huffman Code lengths must be passed to decoder (this table can be effectively compressed).

Maximum length of the Huffman codes in the Intel IPP functions for Huffman coding is 32 bits.

The functions for Huffman encoding/decoding initialize a correspondent specification structure and perform the encoding and decoding operations.

To use the functions for **encoding**, follow this general scheme:

- Call the function `ippsEncodeHuffInitAlloc` to allocate memory and initialize the Huffman structure for encoding with specified frequency table.
  Or call the function `ippsEncodeHuffInit` to initialize the Huffman structure for encoding in the previously created external buffer. The size of this buffer can be computed by calling the function `ippsHuffGetSize`.
- If encoding is performed **by blocks**, call the function `ippsHuffGetDstBuffSize` to compute the size of the destination buffer.
- Call `ippsEncodeHuffOne` to encode a single element, or call `ippsEncodeHuff` to encode the source sequence (vector). (Alternatively the sequence of N element can be encoded by calling `ippsEncodeHuffOne` N times.) Then call `ippsEncodeHuffFinal` to finalize the encoding procedure. Note that the size of the input sequence (vector) should be passed to the decoding functions.
- Call `ippsHuffGetLenCodeTable` to received the table containing the lengths of Huffman codes. This table should be passed to the decoding functions.To increase the effectiveness of encoding it may be compressed by calling the function `ippsHuffLenCodeTablePack`.

- Call ippsHuffFree to free dynamic memory allocated with the Huffman structure for encoding by the function ippsEncodeHuffInitAlloc.

To use these functions for **decoding**, follow this general scheme:

- Call the function ippsDecodeHuffInitAlloc to allocate memory and initialize the Huffman structure for decoding with specified table containing lengths of Huffman codes.
  Or call the function ippsDecodeHuffInit to initialize the Huffman structure for encoding in the previously created external buffer. The size of this buffer can be computed by calling the functionippsHuffGetSize.
- If decoding is performed **by blocks**, call the function ippsHuffGetDstBuffSize to compute the size of the destination buffer.
- Call ippsDecodeHuffOne to encode a single element, or call ippsDecodeHuff to encode the source sequence (vector). The size of the source non-encoded sequence (vector) should be passed from the encoding functions as well as the table containing the lengths of Huffman codes. If this table is in the packed form, it may be restored by calling the function ippsHuffLenCodeTableUnpack.
- Call ippsHuffFree to free dynamic memory allocated with the Huffman structure for decoding by the function ippsDecodeHuffInitAlloc.

EncodeHuffInitAlloc

Allocates memory and initializes the structure for Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuffInitAlloc_8u(const int freqTable[256],
    IppHuffState_8u** ppHuffState);
```

Parameters

freqTable	Table of frequencies of the symbols.
ppHuffState	Double pointer to the Huffman encoding structure.

Description

The function `ippsEncodeHuffInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the structure for Huffman encoding *`pHuffState`*. This structure is specifying by the table of symbol's frequencies *`freqTable`*. The Huffman encoding structure is used by the functions `ippsEncodeHuffOne` and `ippsEncodeHuff`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *`pHuffState`* pointer is `NULL`.
`ippStsFreqTableErr`	Indicates an error if *`freqTable`* is invalid.
`ippStsMaxLenHuffCodeErr`	Indicates an error if the length of the Huffman code is more than expected one.
`ippMemAllocErr`	Indicates an error if memory allocation fails.

HuffFree

Frees memory allocated for the Huffman encoding and decoding structures.

Syntax

```
void ippsHuffFree_8u(IppHuffState_8u* pHuffState);
```

Parameters

`pHuffState`	Pointer to the Huffman coding structure.

Description

The function `ippsHuffFree` is declared in the `ippdc.h` file. This function frees memory allocated for the Huffman encoding and decoding structures by the function `ippsEncodeHuffInitAlloc` and `ippsDecodeHuffInit` respectively.

EncodeHuffInit

Initializes the Huffman encoding structure.

Syntax

```
IppStatus ippsEncodeHuffInit_8u (const int freqTable[256], IppHuffState_8u*
    pHuffState);
```

Parameters

freqTable	Table of frequencies of the symbols.
pHuffState	Pointer to the Huffman encoding structure.

Description

The function `ippsEncodeHuffInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the structure *pHuffState* that is required for Huffman encoding. This structure is specifying by the table of symbol's frequencies *freqTable*. The size of the external buffer should be computed previously by calling the function `ippsHuffGetSize`. Alternatively, the Huffman encoder structure can be initialized by the function `ippsEncodeHuffInitAlloc`.

The Huffman encoding structure is used by the functions `ippsEncodeHuffOne` and `ippsEncodeHuff`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pHuffState* pointer is `NULL`.
`ippStsFreqTableErr`	Indicates an error if *freqTable* is invalid.
`ippStsMaxLenHuffCodeErr`	Indicates an error if the length of the Huffman code is more than expected one.

HuffGetSize

Computes size of the external buffer for the Huffman encoding structure.

Syntax

```
IppStatus ippsHuffGetSize_8u(int* pHuffStateSize);
```

Parameters

pHuffStateSize	Pointer to the computed size of the Huffman coding structure.

Description

The function `ippsHuffGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) for the external buffer required for the Huffman encoding structure. This function should be called prior to the function `ippsEncodeHuffInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pHuffStateSize* pointer is `NULL`.

EncodeHuffOne

Performs Huffman encoding of a single element.

Syntax

```
IppStatus ippsEncodeHuffOne_8u(Ipp8u src, Ipp8u* pDst, int dstOffsetBits,
    IppHuffState_8u* pHuffState);
```

Parameters

src	Source single element.
pDst	Pointer to the destination buffer.
dstOffsetBits	Offset (in bits) of the position of code word in the destination buffer, starting from the high bit.
pHuffState	Pointer to the Huffman encoding structure.

Description

The function ippsEncodeHuffOne is declared in the ippdc.h file. This function encodes the single source element *src* using the Huffman encoding structure *pHuffState*. The result is placed in the destination buffer *pDst* in position specified by the value (in bits) of offset from the high bit *dstOffsetBits*. The Huffman encoding structure should be initialized by calling the functions ippsEncodeHuffInit or ippsEncodeHuffInitAlloc beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

EncodeHuff

Performs Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuff_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int*
    pDstLen, IppHuffState_8u* pHuffState);
```

Parameters

pSrc	Pointer to the source buffer.
srcLen	Number of elements in the source buffer.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the number of elements in the destination vector after encoding.
pHuffState	Pointer to the Huffman encoding structure.

Description

The function ippsEncodeHuff is declared in the ippdc.h file. This function encodes data in the source buffer *pSrc* using the Huffman encoding structure *pHuffState*. The encoded data are placed in the destination buffer *pDst*. The actual number of elements in the destination buffer is written in *pDstLen*. The Huffman encoding structure should be initialized by calling the functions ippsEncodeHuffInit or ippsEncodeHuffInitAlloc beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *srcLen* is less than or equal to 0.

EncodeHuffFinal

Performs Huffman encoding of the remainder.

Syntax

```
IppStatus ippsEncodeHuffFinal_8u(Ipp8u* pDst, int* pDstLen, IppHuffState_8u*
    pHuffState);
```

Parameters

pDst	Pointer to the destination vector.
pDstLen	Pointer to the number of elements in the destination vector after encoding.
pHuffState	Pointer to the the Huffman coding structure.

Description

The function ippsEncodeHuffFinal is declared in the ippdc.h file. This function performs the final stage of the Huffman encoding - flushes the remainder part of the encoded data to the destination vector *pDst* with *pDstLen* elements using the encoding structure *pHuffState*. The function should be run just after ippsEncodeHuff, while the Huffman encoding structure *pHuffState* is still initialized.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

HuffGetLenCodeTable

Returns the table containing lengths of Huffman codes.

Syntax

```
IppStatus ippsHuffGetLenCodeTable_8u(int codeLenTable[256], IppHuffState_8u*
    pHuffState);
```

Parameters

codeLenTable	Destination table containing lengths of Huffman codes.
pHuffState	Pointer to the pointer to the Huffman decoding structure.

Description

The function `ippsHuffGetLenCodeTable` is declared in the `ippdc.h` file. This function creates the table *codeLenTable* containing the lengths of the used Huffman codes. Their value are taken from the Huffman encoding structure *pHuffState*, which must be initialized beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pHuffState* pointer is `NULL`.

DecodeHuffInitAlloc

Allocates memory and initializes the Huffman decoding structure.

Syntax

```
IppStatus ippsDecodeHuffInitAlloc_8u(const int codeLenTable[256],
    IppHuffState_8u** ppHuffState);
```

Parameters

codeLenTable	Table containing lengths of Huffman codes.

ppHuffState	Double pointer to the Huffman decoding structure.

Description

The function ippsDecodeHuffInitAlloc is declared in the ippdc.h file. This function allocates memory and initializes the Huffman decoding structure *pHuffState*. The structure requires the table of lengths of Huffman codes *codeLenTable* that was used in the encoding operation. This table can be obtained by calling the function ippsHuffGetLenCodeTable or ippsHuffLenCodeTableUnpack. The Huffman decoding structure is used by the functions ippsDecodeHuffOne and ippsDecodeHuff.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pHuffState* pointer is NULL.
ippMemAllocErr	Indicates an error if memory allocation fails.

DecodeHuffInit

Initialized the Huffman decoding structure.

Syntax

```
IppStatus ippsDecodeHuffInit_8u(const int codeLenTable[256], IppHuffState_8u*
    pHuffState);
```

Parameters

codeLenTable	Table containing lengths of Huffman codes.
pHuffState	Pointer to the Huffman decoding structure.

Description

The function ippsDecodeHuffInit is declared in the ippdc.h file. This function initializes in the external buffer the Huffman decoding structure *pHuffState*. The structure requires the table of lengths of Huffman codes *codeLenTable* that was used in the encoding operation. This table can be obtained by calling the function ippsHuffGetLenCodeTable or ippsHuffLenCodeTableUnpack. The size of the external buffer should be computed previously by calling the function ippsHuffGetSize. Alternatively, the Huffman decoding structure can be initialized by the function ippsDecodeHuffInitAlloc.

The Huffman decoding structure is used by the functions ippsDecodeHuffOne and ippsDecodeHuff.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pHuffState* pointer is NULL.

DecodeHuffOne

Decodes a single code word.

Syntax

```
IppStatus ippsDecodeHuffOne_8u(const Ipp8u* pSrc, int srcOffsetBits, Ipp8u* pDst,
    IppHuffState_8u* pHuffState);
```

Parameters

pSrc	Pointer to the source buffer.
srcOffsetBits	Offset (bits) in the source buffer.
pDst	Pointer to the destination buffer.
pHuffState	Pointer to the Huffman decoding structure.

Description

The function ippsDecodeHuffOne is declared in the ippdc.h file. This function performs Huffman decoding of the single code word in the source buffer *pSrc*. Its position is specified by the value (in bits) of offset from the high bit *srcOffsetBits*. The decoded element is stored in the destination buffer *pDst*. The function uses the Huffman decoding structure *pHuffState* that should be initialized by calling the functions ippsDecodeHuffInitAlloc or ippsDecodeHuffInit beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

DecodeHuff

Performs Huffman decoding.

Syntax

```
IppStatus ippsDecodeHuff_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int*
    pDstLen, IppHuffState_8u* pHuffState);
```

Parameters

pSrc	Pointer to the source buffer.
srcLen	Number of elements in the source buffer.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the expected number of elements in the destination buffer; after decoding to the actual number of elements in the destination buffer.
pHuffState	Pointer to the Huffman coding structure.

Description

The function `ippsDecodeHuff` is declared in the `ippdc.h` file. This function decodes the data in the source buffer *pSrc* containing *srcLen* code words. The *pDstLen* decoded elements are stored in the destination vector *pDst*. The function uses the Huffman decoding structure *pHuffState* that must be initialized by calling the functions `ippsDecodeHuffInitAlloc` or `ippsDecodeHuffInit` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *srcLen* is less than or equal to 0.

HuffGetDstBuffSize

Computes the size of a destination buffer for Huffman encoding/decoding by blocks.

Syntax

```
IppStatus ippsHuffGetDstBuffSize_8u(const int codeLenTable[256], int srcLen,
    int* pEncDstBuffSize, int* pDecDstBuffSize);
```

Parameters

codeLenTable	Table with lengths of Huffman codes.
srcLen	Number of elements in the source buffer.
pEncDstBuffSize	Pointer to the computed size of the destination buffer for Huffman encoding.
pDecDstBuffSize	Pointer to the computed size of the destination buffer for Huffman decoding.

Description

The function `ippsHuffGetDstBuffSize` is declared in the `ippdc.h` file. This function computes the size (in bytes) of destination buffer for Huffman encoding and decoding by blocks. It should be called prior to the encoding/decoding operations.

The function computes the size of destination buffers for both operations. To exclude one of them the correspondent pointer should be set to `NULL`.

Application Notes

The following formula can be used to estimate maximum sizes of the destination buffer for the Huffman encoding and decoding by blocks:

```
encDstBuffSize = srcLen*4,
decDstBuffSize = srcLen*8
```

You can use these values without calling the function `ippsHuffGetDstBuffSize`. Nevertheless the function `ippsHuffGetDstBuffSize` allows to minimize the sizes of the destination buffers.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error if the *codeLenTable* pointer is NULL.

HuffLenCodeTablePack

Packs the table containing lengths of Huffman codes.

Syntax

```
IppStatus ippsHuffLenCodeTablePack_8u(const int codeLenTable[256], Ipp8u* pDst,
    int* pDstLen);
```

Parameters

codeLenTable	Table containing lengths of Huffman codes.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the number of elements in the destination buffer; after operation to the actual number of elements in the destination buffer.

Description

The function ippsHuffLenCodeTablePack is declared in the ippdc.h file. This function compresses the table *codeLenTable* containing lengths of Huffman codes and that is required for decoding. The table may be retrieved by calling the function ippsHuffGetLenCodeTable. To restore the table from the compressed form, call the function ippsHuffLenCodeTableUnpack.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *pDstLen* is less than or equal to 0; or if *pDstLen* is less than actual size of the output.

HuffLenCodeTableUnpack

Unpacks the table containing lengths of Huffman codes.

Syntax

```
IppStatus ippsHuffLenCodeTableUnpack_8u(const Ipp8u* pSrc, int* pSrcLen, int
    codeLenTable[256]);
```

Parameters

pSrc	Pointer to the source buffer.
pSrcLen	Pointer to the number of elements in the source buffer; after operation pointer to the actual number of elements in the source buffer.
codeLenTable	Table containg lengths of Huffman codes.

Description

The function `ippsHuffLenCodeTableUnpack` is declared in the `ippdc.h` file. This function restores the table *codeLenTable* containing the lengths of Huffman codes from the packed form *pSrc* created by the function `ippsHuffLenCodeTablePack`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *pSrcLen* is less than or equal to 0; or if *pSrcLen* is less than actual size.

Dictionary-Based Compression Functions

This section describes the Intel IPP functions that use different dictionary-based compression methods.

The full list of these functions is given in the Table 13-3

Table 13-3 Intel IPP Dictionary-Based Data Compression Functions

Function Short Name	Description
LZSS Coding	
EncodeLZSSInitAlloc	Allocates memory and initializes the LZSS state structure.
LZSSFree	Frees memory allocated for the LZSS state structure.
EncodeLZSSInit	Initializes the LZSS state structure.
LZSSGetSize	Computes the size of the LZSS state structure.
EncodeLZSS	Performs LZSS encoding.
EncodeLZSSFlush	Flushes the last few bits in the bitstream and aligns the output data on a byte boundary.
DecodeLZSSInitAlloc	Allocates memory and initializes the LZSS state structure.
DecodeLZSSInit	Initializes the LZSS state structure.
DecodeLZSS	Performs LZSS decoding
ZLIB Coding	
EncodeLZ77Init	Initializes the encoding LZ77 state structure.
EncodeLZ77GetSize	Computes the size of the encoding LZ77 state structure.
EncodeLZ77InitAlloc	Allocates memory and initializes the encoding LZ77 state structure.
LZ77Free	Frees memory allocated for the encoding or decoding LZ77 state structures.
EncodeLZ77	Performs LZ77 encoding.
EncodeLZ77SelectHuffMode	Determines the optimal encoding mode.
EncodeLZ77FixedHuff	Performs encoding with the fixed Huffman codes.
EncodeLZ77DynamicHuff	Performs encoding with the dynamic Huffman codes.
EncodeLZ77StoredBlock	Stores the data block without compression.
EncodeLZ77Flush	Writes the checksum and total length of the input data to the end of the output stream.
EncodeLZ77GetPairs	Retrieves pair data from the encoding state structure.
EncodeLZ77SetPairs	Sets pair data to the encoding state structure.

Table 13-3 Intel IPP Dictionary-Based Data Compression Functions (continued)

Function Short Name	Description
EncodeLZ77GetStatus	Reads the deflate status value from the encoding state structure.
EncodeLZ77SetStatus	Writes the deflate status value to the encoding state structure.
EncodeLZ77Reset	Resets the internal encoding state structure to the initial state.
DecodeLZ77Init	Initializes the LZ77 decoding structure.
DecodeLZ77GetSize	Computes the size of the decoding LZ77 state structure.
DecodeLZ77InitAlloc	Allocates memory and initializes the decoding LZ77 state structure.
DecodeLZ77	Performs LZ77 decoding.
DecodeLZ77GetBlockType	Determines the type of encoded data block.
DecodeLZ77FixedHuff	Performs fixed Huffman decoding.
DecodeLZ77DynamicHuff	Performs dynamic Huffman decoding.
DecodeLZ77StoredBlock	Performs decoding of the block stored without compression.
DecodeLZ77GetPairs	Retrieves pair data from the decoding state structure.
DecodeLZ77SetPairs	Sets pair data to the decoding state structure.
DecodeLZ77GetStatus	Reads the inflate status value from the LZ77 decoding state structure.
DecodeLZ77SetStatus	Sets the inflate status to the desired value in the LZ77 decoding state structure
DecodeLZ77Reset	Resets the internal decoding state structure to the initial state.
Adler32	Computes theAdler32 checksum for the source data buffer.
CRC32	Computes the CRC32 checksum for the source data buffer.

LZSS Compression Functions

This section describes Intel IPP data compression functions that implement the LZSS (Lempel-Ziv-Storer-Szymanski) compression algorithm [Storer82]. These functions perform LZSS coding with a vocabulary size of 32KB and 256-byte maximum match string length.

EncodeLZSSInitAlloc

Allocates memory and initializes the LZSS encoder state structure.

Syntax

```
IppStatus ippsEncodeLZSSInitAlloc_8u (IppLZSSState_8u** ppLZSSState);
```

Parameters

ppLZSSState	Double pointer to the LZSS encoder state structure.

Description

The function `ippsEncodeLZSSInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZSS encoder state structure *ppLZSSState* required for the encoder functions `ippsEncodeLZSS` and `ippsEncodeLZSSFlush`.

Return Values

`ippStsNoErr`	Indicates no errors.
`ippStsNullPtrErr`	Indicates an error if the *ppLZSSState* pointer is `NULL`.
`StsMemAllocErr`	Indicates an error if memory allocation fails.

LZSSFree

Frees memory allocated for the LZSS state structure.

Syntax

```
void ippsLZSSFree_8u (IppLZSSState_8u* pLZSSState);
```

Parameters

pLZSSState	Pointer to the LZSS encoder state structure.

Description

The function ippsEncodeLZSSFree is declared in the ippdc.h file. This function frees memory allocated for the LZSS state structure for encoding and decoding by the function ippsEncodeLZSSInitAlloc or ippsDecodeLZSSInitAlloc respectively.

EncodeLZSSInit

Initializes the LZSS encoder state structure.

Syntax

```
IppStatus ippsEncodeLZSSInit_8u (IppLZSSState_8u* pLZSSState);
```

Parameters

pLZSSState	Pointer to the LZSS encoder state structure.

Description

The function ippsEncodeLZSSInit is declared in the ippdc.h file. This function initializes the LZSS state structure *pLZSSState* in the external buffer. Its size should be computed previously by calling the function ippsLZSSGetSize. Alternatively, this state structure can be initialized by the function ippsEncodeLZSSInitAlloc.

The LZSS encoder state structure is required for the encoder functions ippsEncodeLZSS and ippsEncodeLZSSFlush.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pLZSSState* is NULL.

LZSSGetSize

Computes the size of the LZSS state structure.

Syntax

```
IppStatus ippsLZSSGetSize_8u (int* pLZSSStateSize);
```

Parameters

pLZSSStateSize	Pointer to the size of the LZSS state structure.

Description

The function `ippsLZSSGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the LZSS state structure for encoding and decoding and stores it to an integer pointed to by *pLZSSStateSize*. The function should be called prior to the function `ippsEncodeLZSSInit` or `ippsDecodeLZSSInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the pointer *pLZSSStateSize* is `NULL`.

EncodeLZSS

Performs LZSS encoding.

Syntax

```
IppStatus ippsEncodeLZSS_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u** ppDst,
    int* pDstLen, IppLZSSState_8u* pLZSSState);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the number of elements in the source buffer; it is updated after encoding.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pLZSSState	Pointer to the LZSS encoding state structure.

Description

The function `ippsEncodeLZSS` is declared in the `ippdc.h` file. This function performs LZSS encoding of data in the source buffer *ppSrc* with *pSrcLen* length and stores the result in the destination vector *pDst*. The length of the destination vector is stored in *pDstLen*. The LZSS encoder state structure *pLZSSState* must be initialized by the functions `ippsEncodeLZSSInitAlloc` or `ippsEncodeLZ77Init` beforehand.

After encoding the function returns the pointers to source and destination buffers shifted by the number of successfully read and encoded bytes, respectively. The function updates *pSrcLen* so it is equal to the actual number of elements in the source buffer.

Example 13-1 shows how to use the function `ippsEncodeLZSS_8u` and supporting functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *srcLen* is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

EncodeLZSSFlush

Encodes the last few bits in the bitstream and aligns the output data on the byte boundary.

Syntax

```
IppStatus ippsEncodeLZSSFlush_8u (Ipp8u** ppDst, int* pDstLen, IppLZSSState_8u*
    pLZSSState);
```

Parameters

ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of destination buffer.
pLZSSState	Pointer to the LZSS encoder state structure.

Description

The function `ippsEncodeLZSSFlush` is declared in the `ippdc.h` file. This function encodes the last few bits (remainder) in the bitstream, writes them to *ppDst*, and aligns the output data on a byte boundary.

Example 13-1 shows how to use the function `ippsEncodeLZSSFlush_8u` and supporting functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if *pDstLen* is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

Example 13-1 Using the Function ippsEncodeLZSS_8u and Supporting Functions

```
#define INBLOCKSIZE 65536
#define OUTBLOCKSIZE 16384
.......................
#include <stdio.h>
#include "ippdc.h"
.......................
FILE                    *inFile, *outFile;
Ipp8u                  *pSrc = NULL, *pDst = NULL, *startpSrc, *startpDst;
int                       srcLen, dstLen = 0;
int                       commonDstLen;
IppLZSSState_8u         *pLZSSState;
IppStatus                 status;
.......................

/*************************************************************/
/* Memory allocation for input data and output code buffers, */
/* and for the internal encoding state structure:            */
/*************************************************************/
startpSrc = ippsMalloc_8u( (INBLOCKSIZE) * sizeof(char) );
startpDst = ippsMalloc_8u( (OUTBLOCKSIZE) * sizeof(char) );
ippsEncodeLZSSInitAlloc_8u(&pLZSSState);

/*************************************************************/
/* Initializing the arguments, reading srcLen bytes of       */
/* input data from input data file to startpSrc:             */
/*************************************************************/
commonDstLen = (OUTBLOCKSIZE);
pDst = startpDst;
dstLen = commonDstLen;
srcLen = fread( startpSrc, sizeof(Ipp8u), INBLOCKSIZE, inFile );
if(srcLen <= 0)
  return -1;
pSrc = startpSrc;
/************************************************************* */
/* The main loop. In every iteration program calls the        */
/* ippsEncodeLZSS_8u. ippsEncodeLZSS_8u changes the values    */
/* of pSrc, srcLen, pDst and dstLen. ippsEncodeLZSS_8u        */
/* returns ippStsDstSizeLessExpected if there is no           */
/* room in the destination buffer pDst to continue encoding.  */
/* In this case dstLen == 0. Such case is handled by          */
/* flushing the pDst to the output file. If ippsEncodeLZSS_8u */
```

Example 13-1 Using the Function ippsEncodeLZSS_8u and Supporting Functions

```
/* returns ippStsNoErr, then the program flushes the current  */
/* pDst of length of (commonDstLen - dstLen) bytes to         */
/* the output file and reads next srcLen bytes of input data  */
/* to startpSrc.                                              */
/************************************************************ */
for( ; ; )
{
  status = ippsEncodeLZSS_8u(&pSrc, &srcLen, &pDst, &dstLen, pLZSSState);
  if( status == ippStsDstSizeLessExpected )
  {
    fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
    dstLen = commonDstLen;
    pDst = startpDst;
  }
  else if( status == ippStsNoErr )
  {
    fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
    pDst = startpDst;
    dstLen = commonDstLen;
     srcLen = fread( startpSrc, sizeof(Ipp8u), INBLOCKSIZE, inFile );
     if( srcLen <= 0 )
       break;
     pSrc = startpSrc;
   }
   else
 return (-1);
} /* for */

/************************************************************/
/*  The call of ippsEncodeLZSSFlush_8u flushes the last few   */
/*  bits of code to the destination buffer:                   */
/************************************************************/
if( ippsEncodeLZSSFlush_8u(&pDst, &dstLen, pLZSSState) == ippStsNoErr )
{
  fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
  ippsLZSSFree_8u(pLZSSState);
  ippsFree( startpSrc );
  ippsFree( startpDst );
  _fcloseall();
}
else
  return (-1);
....................
```

DecodeLZSSInitAlloc

Allocates memory and initializes the LZSS decoder state structure.

Syntax

```
IppStatus ippsDecodeLZSSInitAlloc_8u (IppLZSSState_8u** ppLZSSState)
```

Parameters

ppLZSSState	Double pointer to the LZSS decoder state structure.

Description

The function `ippsDecodeLZSSInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZSS decoder state structure. That structure is used by the function `ippsDecodeLZSS`.

Return Values

`ippStsNoErr`	Indicates no errors.
`ippStsNullPtrErr`	Indicates an error if the pointer *pLZSSState* is `NULL`.
`ippStsMemAllocErr`	Indicates an error if memory allocation fails.

DecodeLZSSInit

Initializes the LZSS decoder state structure.

Syntax

```
IppStatus ippsDecodeLZSSInit_8u (IppLZSSState_8u* pLZSSState);
```

Parameters

pLZSSState	Pointer to the LZSS decoder state structure.

Description

The function ippsDecodeLZSSInit is declared in the ippdc.h file. This function initializes the LZSS decoder state structure in the external buffer, the size of which should be computed previously by calling the function ippsLZSSGetSize. Alternatively, this state structure can be initialized by the function ippsDecodeLZSSInitAlloc.

The LZSS decoder state structure is required for the function ippsDecodeLZSS.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pLZSSState* is NULL.

DecodeLZSS

Performs LZSS decoding.

Syntax

```
IppStatus ippsDecodeLZSS_8u (Ipp8u* ppSrc, int* pSrcLen, Ipp8u*8 ppDst,
    int* pDstLen, IppLZSSState_8u* pLZSSState);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pLZSSState	Pointer to the LZSS decoding state structure.

Description

The function ippsDecodeLZSS is declared in the ippdc.h file. This function performs LZSS decoding of the *pSrcLen* elements of the *ppSrc* source buffer and stores the result in the *pDst* destination vector. The length of the destination vector is stored in *pDstLen*. The LZSS decoder state structure *pLZSSState* must be initialized by the functions ippsDecodeLZSSInitAlloc or ippsDecodeLZSSInit beforehand.

After decoding the function returns the pointers to source and destination buffers shifted by the number of successfully read and decoded bytes respectively. The function updates *pSrcLen* so it is equal to the actual number of elements in the source buffer.

Example 13-2 shows how to use the `ippsDecodeLZSS_8u` function and supporting functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *pSrcLen* or *pDstLen* is negative.
`ippStsDstSizeLessExpected`	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

Example 13-2 Using the Function `ippsDecodeLZSS` and Supporting functions

```
#define INBLOCKSIZE 65536
#define OUTBLOCKSIZE 16384
..................
#include <stdio.h>
#include "ippdc.h"
..................
FILE                    *inFile, *outFile;
Ipp8u                   *pSrc = NULL, *pDst = NULL, *startpSrc, *startpDst;
int                      srcLen, dstLen = 0;
IppLZSSState_8u         *pLZSSState;
IppStatus                status;
..................
/*************************************************************/
/*  Memory allocation for input code and output restored data */
/*  buffers, and for the internal decoding state structure:   */
/*************************************************************/
startpSrc = ippsMalloc_8u( OUTBLOCKSIZE * sizeof(Ipp8u) );
startpDst = ippsMalloc_8u( INBLOCKSIZE * sizeof(Ipp8u) );
ippsDecodeLZSSInitAlloc_8u( &pLZSSState );
pSrc = startpSrc;
pDst = startpDst;
/*********************************************************************/
/* The main loop. In every iteration program calls the ippsDecodeLZSS_8u. */
/* ippsDecodeLZSS_8u changes the values of pSrc, srcLen, pDst and dstLen. */
/* ippsDecodeLZSS_8u* returns ippStsDstSizeLessExpected if there is no    */
/* room in the destination buffer pDst to continue decoding. In this case */
/* dstLen == 0. Such case is handled by flushing the pDst to the output   */
/* file. If ippsDecodeLZSS_8u returns ippStsNoErr, then program flushes   */
/* the current pDst of length of (INBLOCKSIZE - dstLen) bytes to the      */
/* output file and reads next srcLen bytes of input code to startSrc      */
/*********************************************************************/
for( ; ; )
{
   status = ippsDecodeLZSS_8u( &pSrc, &srcLen, &pDst, &dstLen, pLZSSState);
    if( status == ippStsNoErr )
    {
      fwrite(startpDst, sizeof(Ipp8u), (INBLOCKSIZE - dstLen), outFile);
```

Example 13-2 Using the Function `ippsDecodeLZSS` and Supporting functions

```
        pDst = startpDst;
        dstLen = INBLOCKSIZE;
        srcLen = fread( startpSrc, sizeof(Ipp8u), OUTBLOCKSIZE, inFile );
        pSrc = startpSrc;
        if(srcLen == 0)
          break;
      }
      else if( status == ippStsDstSizeLessExpected )
      {
         fwrite(startpDst, sizeof(Ipp8u), (INBLOCKSIZE - dstLen), outFile);
         pDst = startpDst;
         dstLen = INBLOCKSIZE;
      }
  } /* for */
  /*************************************************************/
  /* Free the memory allocated for input code and output       */
  /* restored data buffers and for the internal decoding state */
  /* structure:                                                */
  /*************************************************************/
  ippsLZSSFree_8u( pLZSSState );
  ippsFree( startpSrc );
  ippsFree( startpDst );
  _fcloseall();
  .............
```

ZLIB Coding Functions

This section describes Intel IPP data compression functions that implement compression methods and data formats defined by the following specifications: [RFC1950], [RFC1951], and [RFC1952]. These formats are also known as ZLIB, DEFLATE, and GZIP, respectively.

A basic algorithm for these data compression methods is based on the Lempel-Ziv (LZ77) [Ziv77] dictionary-based compression.

The structure of ZLIB data is schematically shown on the Figure 13-1.

Figures Figure 13-2 and Figure 13-3 shows the schemes of applying the Intel IPP ZLIB functions for encoding and decoding, respectively. The functions which names are bolded are the mandatory functions for the specified stage.

The full version of the `zlib` library based on the Intel IPP is presented as the Intel IPP Sample downloadable form http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm.

Special Parameters

The ZLIB coding functions require you to specify several special parameters.

The *comprLevel* parameter specifies the level of compression rate and compression ratio. Table 13-4 lists the possible values of the *comprLevel* parameter and their meanings.

Table 13-4 comprLevel Parameter for ZLIB Functions

Value	Descriptions
IppLZ77FastCompr	Fast compression, maximum compression rate, and below average compression ratio
IppLZ77AverageCompr	Average compression rate, average compression ratio
IppLZ77BestCompr	Slow compression, maximum compression ratio

The *checksum* parameter specifies what algorithm is used to compute checksum for input data. Table 13-5 lists the possible values of the *checksum* parameter and their meanings.

Table 13-5 Checksum Parameter for ZLIB Functions

Value	Description
IppLZ77NoChcksm	Checksum is not calculated.
IppLZ77Adler32	Checksum is calculated using Adler32 algorithm.
IppLZ77CRC32	Checksum is calculated using the CRC32 algorithm.

Figure 13-1 ZLIB Data Structure

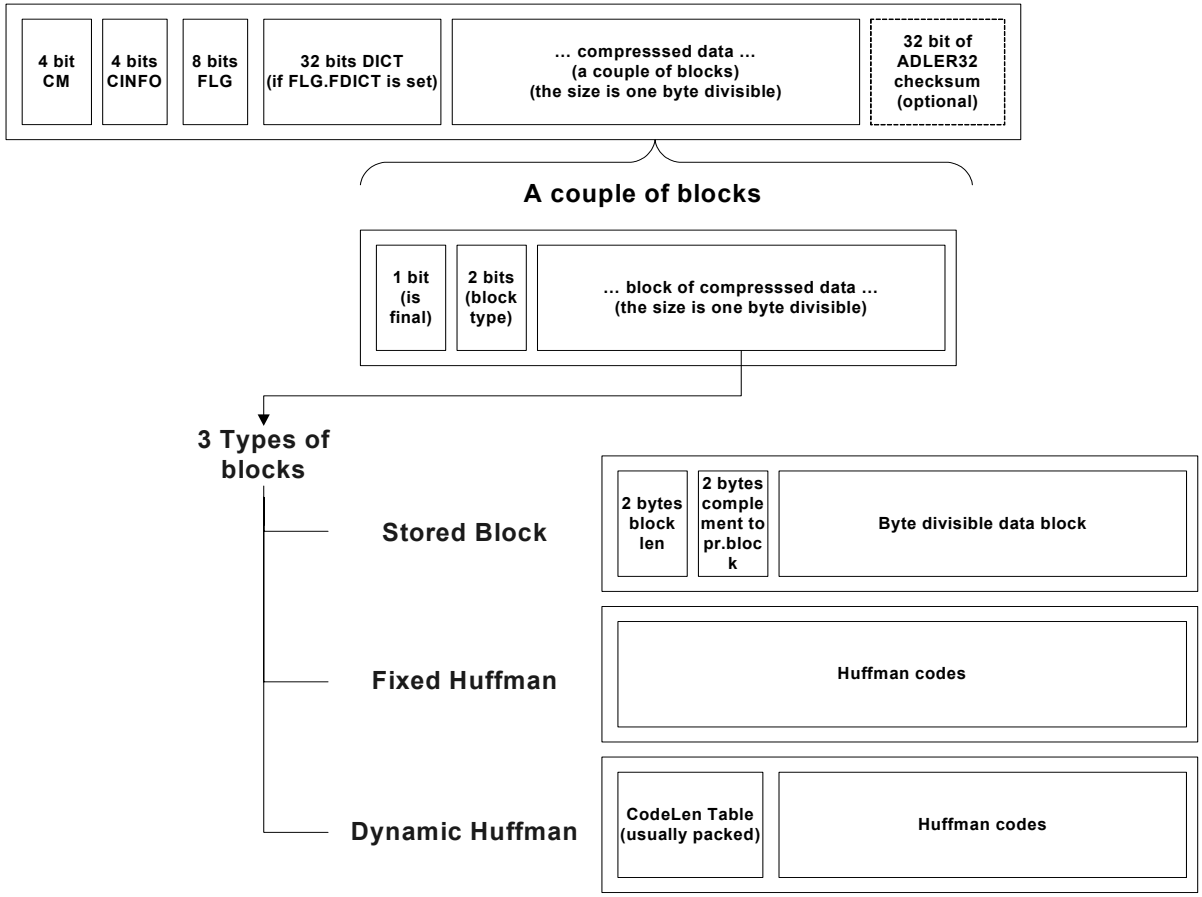


Figure 13-2 Encoding Data Block with Intel IPP ZLIB Functions

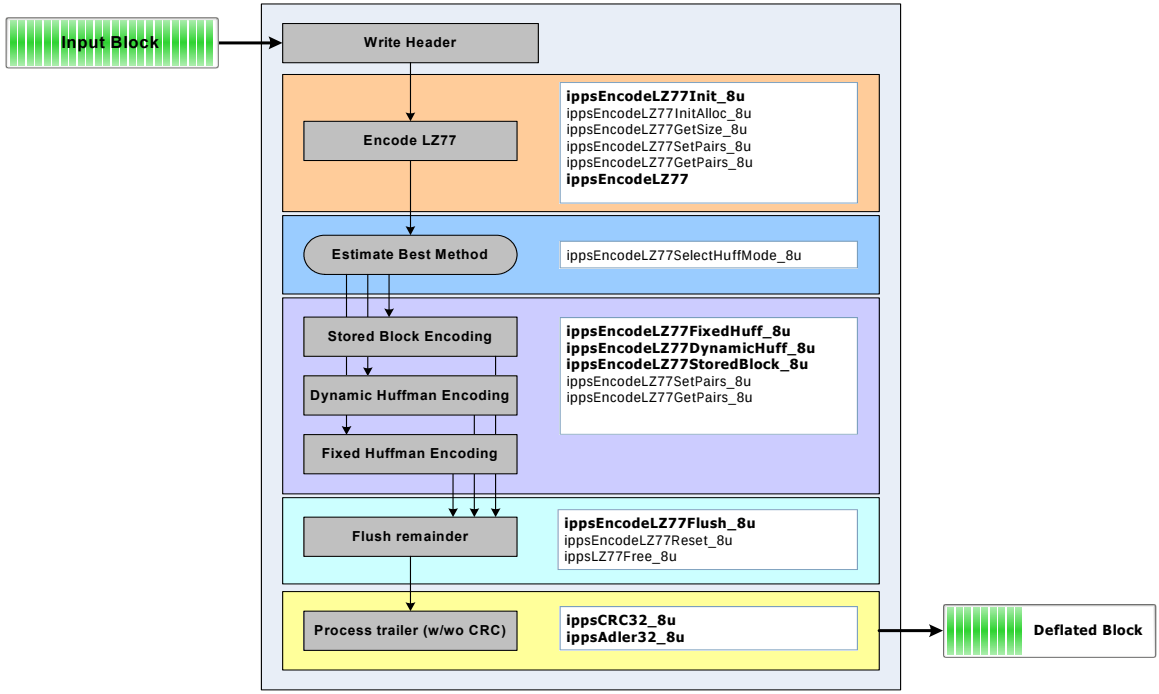
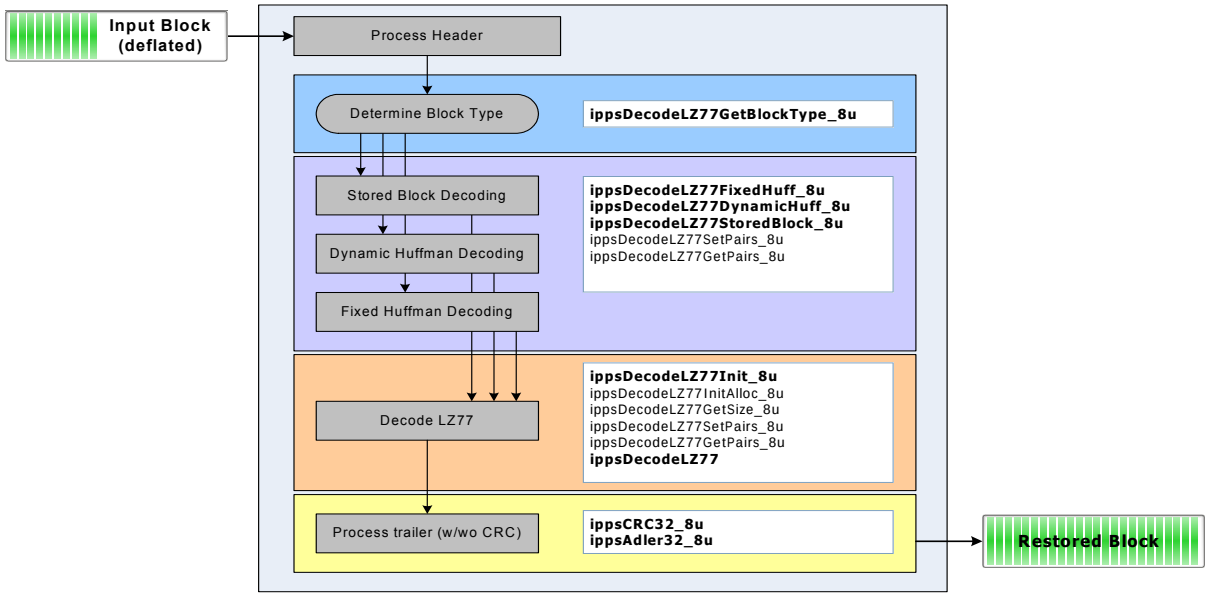


Figure 13-3 Decoding Data Block with Intel IPP ZLIB Functions



The *flush* parameter specifies the encoding mode for data block encoding. Table 13-6 lists the possible values of the *flush* parameter and their meanings.

Table 13-6 flush Parameter for ZLIB Functions

Value	Descriptions
IppLZ77NoFlush	The end of the block is aligned to a byte boundary.
IppLZ77SyncFlush	The end of the block is aligned to a byte boundary, and 4-byte marker is written to pDst.
IppLZ77FullFlush	The end of the block is aligned to a byte boundary, 4-byte marker is written to pDst, sliding dictionary is zeroed.
IppLZ77FinishFlush	The end of the block is aligned to a byte boundary and the function returns the ippStsStreamEnd status.

The *deflateStatus* parameter specifies the encoding status to ensure the compatibility with the RFC1951specification. This parameter is used by Intel IPP ZLIB encoding functions. Table 13-7 lists the possible values of the *deflateStatus* parameter and their meanings.

Table 13-7 deflateStatus Parameter for ZLIB Encoding Functions

Value	Descriptions
IppLZ77StatusInit	Specified the deflate implementation of encoding functions, must be used before stream encoding.
IppLZ77StatusLZ77Process	Call the deflate implementation of encoding function.
IppLZ77StatusHuffProcess	Call the deflate implementation of the encoding function with the fixed Huffman codes.
IppLZ77StatusFinal	Specified the last block in the stream.

The *inflateStatus* parameter specifies the decoding status to ensure the compatibility with the RFC1951specification. This parameter is used by Intel IPP ZLIB decoding functions. Table 13-8 lists the possible values of the *inflateStatus* parameter and their meanings.

Table 13-8 inflateStatus Parameter for ZLIB Decoding Functions

Value	Descriptions
IppLZ77inflateStatusInit	Specified the deflate implementation of encoding functions, must be used before stream encoding.
IppLZ77InflateStatusHuffProcess	Call the deflate implementation of the encoding function with the fixed Huffman codes.
IppLZ77InflateStatusLZ77Process	Call the deflate implementation of encoding function.
IppLZ77InflateStatusFinal	Specified the last block in the stream.

EncodeLZ77Init

Initializes the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77Init_8u (IppLZ77comprLevel comprLevel, ippLZ77chcksm
    checksum, IppLZ77_8u* pLZ77State);
```

Parameters

comprLevel	Specifies desired compression level (see Table 13-4).
checksum	Specifies an algorithm to compute the checksum for input data (see Table 13-5).
ppLZ77State	Pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77Init` is declared in the `ippdc.h` file. This function initializes the LZ77 encoding state structure in the external buffer. Its size should be previously computed by calling the function `ippsEncodeLZ77GetSize`. The function implements different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum* and different compression levels that are specified by the parameter *comprLevel*. The encoding state structure is used by different ZLIB encoder functions.

Alternatively, this state structure can be initialized by the function `ippsEncodeLZ77InitAlloc`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pLZ77State* pointer is NULL.
ippStsBadArgErr	Indicates an error if the *checksum* or *comprLevel* parameter has an illegal value.

EncodeLZ77GetSize

Computes the size of the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77GetSize_8u (int* pLZ77VLCStateSize);
```

Parameters

pLZ77StateSize	Pointer to the size of the LZ77 encoding state structure.

Description

The function ippsEncodeLZ77GetSize is declared in the ippdc.h file. This function computes the size in bytes of the LZ77 encoding state structure. The function should be called prior to the function ippsEncodeLZ77Init.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pLZ77StateSize* is NULL.

EncodeLZ77InitAlloc

Allocates memory and initializes the encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77InitAlloc_8u (IppLZ77comprLevel comprLevel,
    IppLZ77chcksm checksum, IppLZ77_8u** ppLZ77State);
```

Parameters

comprLevel	Specifies desired compression level (see Table 13-4).
checksum	Specifies an algorithm to compute the checksum for input data (see Table 13-5).
ppLZ77State	Double pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77InitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZ77 encoding state structure *ppLZ77State*. The function implements different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum* and different compression levels that are specified by the parameter *comprLevel*. The encoding state structure is used by different ZLIB encoder functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *ppLZ77State* pointer is `NULL`.
`ippStsBadArgErr`	Indicates an error if the *checksum* or *comprLevel* parameter has an illegal value.
`ippStsMemAlloc`	Inidicates an error if memory allocation fails.

LZ77Free

Frees memory allocated for the LZ77 encoding and decoding structures.

Syntax

```
void ippsLZ77Free_8u (IppLZ77State_8u* pLZ77State);
```

Parameters

pLZ77State	Pointer to the LZ77 encoding or decoding state structure.

Description

The function `ippsLZ77Free` is declared in the `ippdc.h` file. This function frees memory allocated for the LZ77 encoding or decoding state structures.

EncodeLZ77

Performs LZ77 encoding.

Syntax

```
IppStatus ippsEncodeLZ77_8u (Ipp8u** ppSrc, int* pSrcLen, IppLZ77Pair** ppDst,
    int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
flush	Specifies the encoding mode for data blocks (see Table 13-6).
pLZ77State	Pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77` is declared in the `ippdc.h` file. This function performs ZLIB encoding [Ziv77] using the sliding window technique in accordance with the RFC1951 specification.

The function uses the LZ77 encoding state structure *pLZ77State* that must be previously initialized by the function `ippsEncodeLZ77InitAlloc` or `ippsEncodeLZ77Init`.

The output data are of `IppLZ77Pair` type that is defined as follows:

```
typedef struct IppLZ77Pairs {
    Ipp16u length;
    Ipp16u offset;
} IppLZ77Pair;
```

where `length` is a match length, and `offset` is a distance between the far-left position of the look-ahead buffer and the first literal of the string match in the sliding window. If the match is not found, the offset field contains 0 and the length field contains a single-literal code.

Example 13-3 and Example 13-4 show how to use the `ippsEncodeLZ77` function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *dstLen* is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

EncodeLZ77SelectHuffMode

Determines the optimal encoding mode.

Syntax

```
IppStatus ippsEncodeLZ77SelectHuffMode_8u(IppLZ77Pair* pSrc, int srcLen,
    IppLZ77HuffMode* pHuffMode, IppLZ77State_8u* pLZ77State);
```

Parameters

pSrc	Pointer to the source buffer.
srcLen	Length of the source buffer.
pHuffMode	Pointer to the value of the parameter that specified the optimal encoding mode. Possible values are: IppLZ77UseFixed Encoding with fixed Huffman codes. IppLZ77UseDynamic Encoding with dynamic Huffman codes.
pLZ77State	Pointer to the LZ77 encoding state structure.

Description

The function ippsEncodeLZ77SelectHuffMode is declared in the ippdc.h file. This function analyzes the statistics of the source data stream *pSrc* of the IppLZ77Pair type and determines the optimal encoding mode - with dynamic Huffman codes or with fixed Huffman codes. The function returns the corresponding value of the parameter *pHuffMode*. Source data are produced by the function ippsEncodeLZ77.

This optimal mode is recommended, but you can use your own methods to encode the source data.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pLZ77State* or *pPairs* pointer is NULL.

EncodeLZ77FixedHuff

Performs fixed Huffman encoding.

Syntax

```
IppStatus ippsEncodeLZ77FixedHuff_8u (IppLZ77Pair** ppSrc, int* pSrcLen,
    Ipp8u** ppDst, int* pDstLen, IppLZ77flush flush, IppLZ77_8u* pLZ77State);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
flush	Specifies the encoding mode for data blocks (see Table 13-6).
pLZ77State	Pointer to the LZ77 encoding state structure.

Description

The function ippsEncodeLZ77FixedHuff is declared in the ippdc.h file. This function performs encoding with the fixed Huffman codes (see RFC1951specification) of the input data stream of the IppLZ77Pair type that is produced by the function ippsEncodeLZ77.

The function ippsEncodeLZ77FixedHuff writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter *flush*.

If the function returns one of the ippStsNoErr, ippStsDstSizeLessExpected, or ippStsStreamEnd statuses, the values of the pointers to the source and destination data buffers and to their lengths are changed and their new values are returned.

The function uses the LZ77 encoding state structure *pLZ77State* that must be previously initialized by the function ippsEncodeLZ77InitAlloc or ippsEncodeLZ77Init.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *DstLen* is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.
ippStsStreamEnd	Indicates a warning that the end of the stream occurs. The function returns this status if the parameter *flush* is set to IppLZ77FinishFlush.

EncodeLZ77DynamicHuff

Performs dynamic Huffman encoding.

Syntax

```
IppStatus ippsEncodeLZ77DynamicHuff_8u(IppLZ77Pair** ppSrc, int* pSrcLen,
    Ipp8u** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u*
    pLZ77State);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
flush	Specifies the encoding mode for data blocks (see Table 13-6).
pLZ77State	Pointer to the LZ77 encoding state structure.

Description

The function ippsEncodeLZ77DynamicHuff is declared in the ippdc.h file. This function performs dynamic Huffman encoding (see RFC1951specification) of the input data stream of the IppLZ77Pair type that is produced by the function ippsEncodeLZ77.

The function ippsEncodeLZ77DynamicHuff writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter *flush*.

If the function returns one of the ippStsNoErr, ippStsDstSizeLessExpected, or ippStsStreamEnd statuses, the values of the pointers to the source and destination data buffers and to their lengths are changed and their new values are returned.

The function uses the LZ77 encoding state structure *pLZ77State* that must be previously initialized by the function ippsEncodeLZ77InitAlloc or ippsEncodeLZ77Init.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *DstLen* is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.
ippStsStreamEnd	Indicates a warning that the end of the stream occurs. The function returns this status if the parameter *flush* is set to IppLZ77FinishFlush.

EncodeLZ77StoredBlock

Stores the data block without compression.

Syntax

```
IppStatus ippsEncodeLZ77StoredBlock_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u**
    ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
flush	Specifies the encoding mode for data blocks (see Table 13-6).
pLZ77State	Pointer to the LZ77 encoding state structure.

Description

The `ippsEncodeLZ77StoredBlock` function is declared in the `ippdc.h` file. This function copies source data *ppSrc* data to the destination buffer *ppDst* without Huffman encoding.

The function `ippsEncodeLZ77StoredBlock` writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter *flush*.

The function uses the LZ77 encoding state structure *pLZ77State* that must be previously initialized by the function `ippsEncodeLZ77InitAlloc` or `ippsEncodeLZ77Init`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *DstLen* is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

EncodeLZ77Flush

Writes the checksum and total length of the input data to the end of the stream.

Syntax

```
IppStatus ippsEncodeLZ77Flush_8u (Ipp8u** ppDst, int* pDstLen, IppLZ77State_8u*
    pLZ77State);
```

Parameters

ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of destination buffer.
pLZ77State	Pointer to the initialized LZ77 encoding state structure.

Description

The `ippsEncodeLZ77Flush` function is declared in the `ippdc.h` file. This function writes the computed checksum and total length of the input data stream to the end of the output stream that is encoded by the function `ippsEncodeLZ77SelectHuffMode`. This function is used to ensure compatibility with the RFC1950, RFC1951, and RFC1952 specifications.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *`DstLen`* is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

EncodeLZ77GetPairs

Retrieves pair data from the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77GetPairs_8u (IppLZ77Pair** ppPairs, int* pPairsInd,
    int* pPairsLen, IppLZ77State_8u* pLZ77State);
```

Parameters

`ppPairs`	Double pointer to the pair buffer.
`pPairsInd`	Pointer to the current index in the pair buffer.
`pPairsLen`	Pointer to the length of the pair buffer.
`pLZ77State`	Pointer to the initialized LZ77 encoding state structure.

Description

The `ippsEncodeLZ77GetPairs` function is declared in the `ippdc.h` file. This function returns the following pointers: *`ppPairs`* to the pair buffer, *`pPairsLen`* to the length of the pair buffer, and *`pPairsInd`* to the current index. These pointers are retrieved from the initialized LZ77 encoding state structure *`pLZ77State`*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pLZ77State* or *pPairs* pointer is NULL.

EncodeLZ77SetPairs

Sets pair data in the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77SetPairs_8u (IppLZ77Pair* pPairs, int pairsInd, int
    pairsLen, IppLZ77State_8u* pLZ77State);
```

Parameters

pPairs	Pointer to the pair buffer.
pairsInd	Current index in the pair buffer.
pairsLen	Length of the pair buffer.
pLZ77State	Pointer to the initialized LZ77 encoding state structure.

Description

The ippsEncodeLZ77SetPairs function is declared in the ippdc.h file. This function sets the pointer to the pair buffer *pPairs*, the length of the pair buffer *pairsLen*, and the current index *pairsInd* in the initialized LZ77 encoding state structure *pLZ77State*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pLZ77State* or *pPairs* pointer is NULL.

EncodeLZ77GetStatus

Reads the deflate status value from the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77GetStatus_8u (IppLZ77VLCDeflateStatus* pDeflateStatus,
    IppLZ77_8u* pLZ77State);
```

Parameters

pDeflateStatus	Pointer to the deflate status parameter (see Table 13-7).
pLZ77State	Pointer to the initialized LZ77 encoding state structure.

Description

The `ippsEncodeLZ77GetStatus` function is declared in the `ippdc.h` file. This function reads the current encoding status value *pDeflateStatus* from the initialized LZ77 state structure *pLZ77State*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pLZ77State* or *pDeflateStatus* pointer is `NULL`.

EncodeLZ77SetStatus

Sets the deflate status to the desired value in the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77SetStatus_8u (IppLZ77DeflateStatus deflateStatus,
    IppLZ77_8u* pLZ77Stat);
```

Parameters

pLZ77State	Pointer to the initialized LZ77 encoding state structure.

deflateStatus	Specified deflate status value (see Table 13-7).

Description

The `ippsEncodeLZ77SetStatus` function is declared in the `ippdc.h` file. This function sets the parameter *deflateStatus* to the specified value in the initialized LZ77 encoding state structure *pLZ77State*.

This parameter should be specified only if the compatibility with the RFC1950, RFC1951, and RFC1952 specifications is required.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pLZ77State* pointer is `NULL`.
`ippStsBadArgErr`	Indicates an error if the *deflateStatus* parameter has an illegal value.

EncodeLZ77Reset

Resets the LZ77 encoding state structure.

Syntax

```
IppStatus ippsEncodeLZ77Reset_8u (IppLZ77State_8u* pLZ77State);
```

Parameters

pLZ77State	Pointer to the LZ77 encoding state structure.

Description

The function `ippsEncodeLZ77Reset` is declared in the `ippdc.h` file. This function resets the LZ77 encoding state structure to the initial state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pLZ77State* pointer is `NULL`.

DecodeLZ77Init

Initializes the LZ77 decoding structure.

Syntax

```
IppStatus ippsDecodeLZ77Init_8u(IppLZ77Chcksm checksum, IppLZ77State_8u*
    pLZ77State);
```

Parameters

checksum	Specifies an algorithm to compute the checksum for output data (see Table 13-5)..
pLZ77State	Pointer to the initialized LZ77 decoding structure.

Description

The function `ippsDecodeLZ77Init` is declared in the `ippdc.h` file. This function initializes the LZ77 decoding structure in the external buffer whose size should be previously computed by calling the function `ippsDecodeLZ77GetSize`. The function uses different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum*. The decoding state structure is used by the function `ippsDecodeLZ77`.

Alternatively, this structure can be initialized by the function `ippsDecodeLZ77InitAlloc`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pLZ77State* is NULL.
ippStsBadArgErr	Indicates an error if the *checksum* parameter has an illiegal value.

DecodeLZ77GetSize

Computes the size of the LZ77 decoding structure.

Syntax

```
IppStatus ippsDecodeLZ77GetSize_8u(int* pLZ77StateSize);
```

Parameters

pLZ77StateSize	Pointer to the size of the LZ77 decoding structure.

Description

The function `ippsDecodeLZ77GetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the LZ77 decoding state structure. The function should be called prior to the function `ippsDecodeLZ77Init`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the pointer *pLZ77StateSize* is `NULL`.

DecodeLZ77InitAlloc

Allocates memory and initializes the LZ77 decoding structure.

Syntax

```
IppStatus ippsDecodeLZ77InitAlloc_8u(IppLZ77Chcksm checksum, IppLZ77State_8u**
    ppLZ77State);
```

Parameters

checksum	Specifies an algorithm to compute the checksum for output data (see Table 13-5).
ppLZ77State	Double pointer to the LZ77 decoding structure.

Description

The function `ippsDecodeLZ77InitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the decoding state structure *ppLZ77State*. The function uses different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum*. The decoding state structure is used by the function `ippsDecodeLZ77`.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error if the pointer *ppLZ77State* is NULL.
ippStsBadArgErr	Indicates an error if the *checksum* parameter has an illiegal value.
ippStsMemAlloc	Indicates an error if memory allocation fails.

DecodeLZ77

Performs LZ77 decoding.

Syntax

```
IppStatus ippsDecodeLZ77_8u(IppLZ77Pair** ppSrc, int* pSrcLen, Ipp8u** ppDst,
    int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
flush	Specifies the encoding mode for data blocks.
pLZ77State	Pointer to the LZ77 decoding structure.

Description

The function ippsDecodeLZ77 is declared in the ippdc.h file. This function performs LZ77 decoding of a sequence of pairs of the IppLZ77Pair type. The sliding window technique implemented in this function ensures compatibility with the RFC1950, RFC1951, and RFC1952 specifications.

The function ippsDecodeLZ77 can be used for decoding data not only in the DEFLATE, ZLIB, GZIP format, but in any other format based on the LZ77 algorithm.

Example 13-3 shows how to use the ippsDecodeLZ77 function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *ppLZ77State* is NULL.
ippStsSizeErr	Indicates an error if *DstLen* is less than or equal to 0.

ippStsDstSizeLessExpected	Indicates a warning that the destination buffer is full.

DecodeLZ77GetBlockType

Determines the type of encoded data block.

Syntax

```
IppStatus ippsDecodeLZ77GetBlockType_8u(Ipp8u** ppSrc, int* pSrcLen,
    IppLZ77HuffMode* pHuffMode, IppLZ77State_8u* pLZ77State);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
pHuffMode	Pointer to the value of the parameter that indicates the encoding mode. Possible values are:
	IppLZ77UseFixed — Encoding with fixed Huffman codes.
	IppLZ77UseDynamic — Encoding with dynamic Huffman codes.
	IppLZ77UseStored — Stored block without Huffman coding.
pLZ77State	Pointer to the LZ77 decoding state structure.

Description

The function ippsDecodeLZ77GetBlockType is declared in the ippdc.h file. This function analyzes the source data *ppSrc* and returns the parameter *pHuffMode* which value indicates what Huffman decoding mode must be used for decoding the source bit stream. In accordance with this value the one of the functions ippsDecodeLZ77FixedHuff, ippsDecodeLZ77DynamicHuff, or ippsDecodeLZ77StoredBlock must be used.

NOTE. If you use only one specified encoding mode, then type of the input data for decoding is known, and this function is not necessary.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *ppLZ77State* is NULL.

ippStsSizeErr	Indicates an error if *SrcLen* is less than or equal to 0.
ippStsSrcSizeLessExpected	Indicates a warning that the source buffer is insufficient (internal bit stream and source buffer do not contain enought bits to decode the type of block).

DecodeLZ77FixedHuff

Performs fixed Huffman decoding.

Syntax

```
IppStatus ippsDecodeLZ77FixedHuff_8u(Ipp8u** ppSrc, int* pSrcLen,
  IppLZ77Pair** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u*
  pLZ77State);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source vector.
ppDst	Double pointer to the destination vector.
pDstLen	Pointer to the length of the destination vector.
flush	Specifies the decoding mode for data blocks.
pLZ77State	Pointer to the LZ77 decoding structure.

Description

The function `ippsDecodeLZ77FixedHuff` is declared in the `ippdc.h` file. This function performs decoding of the source data *ppSrc* with the fixed Huffman codes (see RFC1951specification). The source data must be compatible with the RFC1950, RFC1951, and RFC1952 specifications.

Exit from the `ippsDecodeLZ77FixedHuff` function occurs when:

1. the input stream has ended, but the end-of-block marker is not decoded and the function returns the `ippStsSrcSizeLessExpected` status.
2. the input stream has not ended, but the destination buffer is full and the function returns the `ippStsDstSizeLessExpected` status.
3. the input stream has ended, the end-of-stream marker is decoded and the function returns the `ippStsNoErr` status.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *DstLen* is less than or equal to 0.
ippStsSrcSizeLessExpected	Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).
ippStsDstSizeLessExpected	Indicates a warning that the destination buffer is full.
ippStsStreamEnd	Indicates a warning that the stream ends. This warning returns only if the *flush* parameter is set to *IppLZ77FinishFlush*.

DecodeLZ77DynamicHuff

Performs dynamic Huffman decoding.

Syntax

```
IppStatus ippsDecodeLZ77DynamicHuff_8u(Ipp8u** ppSrc, int* pSrcLen,
    IppLZ77Pair** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u*
    pLZ77State);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source vector.
ppDst	Double pointer to the destination vector.
pDstLen	Pointer to the length of the destination vector.
flush	Specifies the decoding mode for data blocks.
pLZ77State	Pointer to the LZ77 decoding structure.

Description

The function ippsDecodeLZ77DynamicHuff is declared in the ippdc.h file. This function performs decoding of the source data *ppSrc* with the dynamic Huffman codes (see RFC1951specification).

Exit from the ippsDecodeLZ77FixedHuff function occurs when:

1. the input stream has ended, but the end-of-block marker is not decoded and the function returns the `ippStsSrcSizeLessExpected` status.
2. the input stream has not ended, but the destination buffer is full and the function returns the `ippStsDstSizeLessExpected` status.
3. the input stream has ended, the end-of-stream marker is decoded and the function returns the `ippStsNoErr` status.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *`DstLen`* is less than or equal to 0.
`ippStsSrcSizeLessExpected`	Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).
`ippStsDstSizeLessExpected`	Indicates a warning that the destination buffer is full.
`ippStsStreamEnd`	Indicates a warning that the stream ends. This warning returns only if the *`flush`* parameter is set to *`IppLZ77FinishFlush`*.

DecodeLZ77StoredBlock

Performs decoding of the block stored without compression.

Syntax

```
IppStatus ippsDecodeLZ77StoredBlock_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u**
    ppDst, int* pDstLen, IppLZ77State_8u* pLZ77State);
```

Parameters

`ppSrc`	Double pointer to the source buffer.
`pSrcLen`	Pointer to the length of the source vector.
`ppDst`	Double pointer to the destination vector.
`pDstLen`	Pointer to the length of the destination vector.
`pLZ77State`	Pointer to the LZ77 decoding structure.

Description

The function `ippsDecodeLZ77StoredBlock` is declared in the `ippdc.h` file. This function performs decoding of the source data block *ppSrc* that is stored without Huffman coding.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *DstLen* is less than or equal to 0.
`ippStsWrongBlockType`	Indicates a warning that the type of the block is not the “stored without compression type”.
`ippStsSrcSizeLessExpected`	Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).
`ippStsDstSizeLessExpected`	Indicates a warning that the destination buffer is full.
`ippStsStreamEnd`	Indicates a warning that the stream ends. This warning returns only if the *flush* parameter is set to *IppLZ77FinishFlush*.

DecodeLZ77GetPairs

Retrieves pair data from the LZ77 decoding state structure.

Syntax

```
IppStatus ippsDecodeLZ77GetPairs_8u(IppLZ77Pair** ppPairs, int* pPairsInd, int*
    pPairsLen, IppLZ77State_8u* pLZ77State);
```

Parameters

ppPairs	Double pointer to the pair buffer.
pPairsInd	Pointer to the current index in the pair buffer.
pPairsLen	Pointer to the length of the pair buffer.
pLZ77State	Pointer to the initialized LZ77 decoding state structure.

Description

The `ippsDecodeLZ77GetPairs` function is declared in the `ippdc.h` file. This function returns the following pointers: *ppPairs* to the pair buffer, *pPairsLen* to the length of the pair buffer, and *pPairsInd* to the current index. These pointers are retrieved from the initialized LZ77 decoding state structure *pLZ77State*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pLZ77State* or *pPairs* pointer is `NULL`.

DecodeLZ77SetPairs

Sets pair data in the LZ77 decoding state structure.

Syntax

```
IppStatus ippsDecodeLZ77SetPairs_8u(IppLZ77Pair* pPairs, int pairsInd, int
    pairsLen, IppLZ77State_8u* pLZ77State);
```

Parameters

pPairs	Pointer to the pair buffer.
pairsInd	Current index in the pair buffer.
pairsLen	Length of the pair buffer.
pLZ77State	Pointer to the initialized LZ77 decoding state structure.

Description

The `ippsDecodeLZ77SetPairs` function is declared in the `ippdc.h` file. This function sets the pointer to the pair buffer *pPairs*, the length of the pair buffer *pairsLen*, and the current index *pairsInd* in the initialized LZ77 decoding state structure *pLZ77State*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pLZ77State* or *pPairs* pointer is `NULL`.

DecodeLZ77GetStatus

Reads the inflate status value from the LZ77 decoding state structure.

Syntax

```
IppStatus ippsDecodeLZ77GetStatus_8u(IppLZ77InflateStatus* pInflateStatus,
    IppLZ77State_8u* pLZ77State);
```

Parameters

pInflateStatus	Pointer to the inflate status parameter (see Table 13-7).
pLZ77State	Pointer to the LZ77 decoding structure.

Description

The `ippsDecodeLZ77GetStatus` function is declared in the `ippdc.h` file. This function reads the current decoding status value *pInflateStatus* from the initialized LZ77 state structure *pLZ77State*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.

DecodeLZ77SetStatus

Sets the inflate status to the desired value in the LZ77 decoding state structure.

Syntax

```
IppStatus ippsDecodeLZ77SetStatus_8u(IppLZ77InflateStatus inflateStatus,
    IppLZ77State_8u* pLZ77State);
```

Parameters

pLZ77State	Pointer to the initialized LZ77 encoding state structure.
inflateStatus	Specified deflate status value (see Table 13-7).

Description

The `ippsDecodeLZ77SetStatus` function is declared in the `ippdc.h` file. This function sets the parameter *`inflateStatus`* to the specified value in the initialized LZ77 decoding state structure *`pLZ77State`*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *`pLZ77State`* pointer is `NULL`.
`ippStsBadArgErr`	Indicates an error if the *`inflateStatus`* parameter has an illegal value.

DecodeLZ77Reset

Resets the LZ77 decoding state structure.

Syntax

```
IppStatus ippsDecodeLZ77Reset_8u(IppLZ77State_8u* pLZ77State);
```

Parameters

`pLZ77State`	Pointer to the LZ77 encoding state structure.

Description

The function `ippsDecodeLZ77Reset` is declared in the `ippdc.h` file. This function resets the LZ77 decoding state structure to the initial state.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *`pLZ77State`* pointer is `NULL`.

Adler32

Computes the Adler32 checksum for the source data buffer.

Syntax

```
IppStatus ippsAdler32_8u (const Ipp8u* pSrc, int srcLen, Ipp32u* pAdler32);
```

Parameters

pSrc	Pointer to the source data buffer.
srcLen	Number of elements in the source data buffer.
pAdler32	Pointer to the checksum value.

Description

The function `ippsAdler32` is declared in the `ippdc.h` file. This function computes the checksum for *srcLen* elements of the source data buffer *pSrc* and stores it in the *pAdler32*. The checksum is computed using the Adler32 algorithm that is a modified version of the Fletcher algorithm [Flet82], [ITU224], [RFC1950].

You can use this function to compute the accumulated value of the checksum for multiple buffers in the data stream by specifying as an input parameter the checksum value obtained in the preceding function call.

Parameters

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error if *srcLen* is less than or equal to 0.

CRC32

Computes the CRC32 checksum for the source data buffer.

Syntax

```
IppStatus ippsCRC32_8u (const Ipp8u* pSrc, int srcLen, Ipp32u* pCRC32);
```

Parameters

pSrc	Pointer to the source data buffer.
srcLen	Number of elements in the source data buffer.
pCRC32	Pointer to the checksum value.

Description

The function `ippsCRC32` is declared in the `ippdc.h` file. This function computes the checksum for *srcLen* elements of the source data buffer *pSrc* and stores it in the *pCRC32*. The checksum is computed using the CRC32 algorithm [Grif87], [Nel92].

You can use this function to compute the accumulated value of the checksum for multiple buffers in the data stream by specifying as an input parameter the checksum value obtained in the preceding function call.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error if the length of the source vector is less than or equal to 0.

Example of Using Intel IPP ZLIB Coding Functions

The code Example 13-3 demonstrates how the model of a compressor can be implemented using the Intel IPP ZLIB functions `ippsEncodeLZ77_8u` and `ippsDecodeLZ77_8u` and an arbitrary bit encoding/decoding methods for the pairs.

Example 13-3 Using the `ippsEncodeLZ77` and `ippsDecodeLZ77` functions

```
#include <stdio.h>
#include <stdlib.h>
#include "ippdc.h"
#define BUFLEN 1000
...............
int
main(int argc, char* argv[])
{
    int len;
    int lenr;
    Ipp8u* pSrc;
    int srcLen;
    IppLZ77Pair* pDst;
    int dstLen;
```

Example 13-3 Using the `ippsEncodeLZ77` and `ippsDecodeLZ77` functions

```
    IppLZ77Pair* pSrcr;
    int srcLenr;
    Ipp8u* pDstr;
    int dstLenr;
    IppStatus status;
    IppLZ77State_8u* pLZ77State;
    ....................
    /* LZ77 encoding */
    status = ippsEncodeLZ77InitAlloc_8u(IppLZ77FastCompr, IppLZ77CRC32,
                                        &pLZ77State);
   /* Reading the srcLen bytes of input data from the input data file to pSrc */
    ....................
    len = dstLen;
    status = ippsEncodeLZ77_8u(&pSrc, &srcLen, &pDst, &dstLen,
                               IppLZ77FinishFlush, pLZ77State);

    /* Here we have the array of Pairs in pDst of (len - dstLen)  */
    /* In this point the arbitrary bit coding method can be used, */
    /* for example LZSS algorithm.                                */
    ....................
    ....................
    /* Writing the compressed data to the output file             */
    ippsLZ77Free_8u(pLZ77State);
    /* LZ77 decoding */
    status = ippsDecodeLZ77InitAlloc_8u(IppLZ77CRC32, &pLZ77State);

    /* Reading the compressed data from the file                  */
    ....................
    /* On this point the bit decoding method, corresponding to the above   */
    /* selected bit encoding method (LZSS for example) must be used        */
    ....................
    ....................
    /* After bit decoding the pairs restored placed to the pSrcr of srcLenr */
    /*  for further performing of LZ77 decoding                             */
    lenr = dstLenr;
    status = ippsDecodeLZ77_8u(&pSrcr, &srcLenr, &pDstr, &dstLenr,
                               IppLZ77FinishFlush, pLZ77State);
    /* Writing the (lenr - dstLenr) bytes of decompressed data to the output
file */
    ippsLZ77Free_8u(pLZ77State);

    return 0;
} /* main */
```

IBWT-Based Compression Functions

This section describes the Intel IPP functions that support composed algorithms based on the Burrows-Wheeler transform (BWT). The full list of these functions is given in the Table 13-9.

Table 13-9 Intel IPP BWT-Based Compression Functions

Function Short Name	Description
BWT Functions	
BWTFwdGetSize	Computes the size of the external buffer for the forward BWT transform.
BWTFwd	Performs the forward BWT transform.
BWTInvGetSize	Computes the size of the external buffer for the inverse BWT transform.
BWTInv	Performs the inverse BWT transform.
BWTGetSize_SmalBlock	Computes the size of the external buffer for the BWT transforms for small data block.
BWTFwd_SmallBlock	Performs the forward BWT transform for small data block.
BWTInv_SmallBlock	Performs the inverse BWT transform for small data block.
GIT Functions	
EncodeGITInitAlloc	Allocates memory and initializes the GIT encoding state structure.
GITFree	Frees memory allocated by the function `EncodeGITInitAlloc` or `DecodeGITInitAlloc`.
EncodeGITInit	Initializes the GIT encoding state structure.
EncodeGITGetSize	Computes the size of the GIT encoding state structure.
EncodeGIT	Performs GIT encoding.
DecodeGITInitAlloc	Allocates memory and initializes the GIT decoding state structure.
DecodeGITInit	Initializes the GIT decoding state structure.
DecodeGITGetSize	Computes the size of the GIT decoding state structure.
DecodeGIT	Performs GIT decoding.
MTF Functions	
MTFInitAlloc	Allocates memory and initializes the MTF structure.
MTFFree	Frees memory allocated for the MTF structure.
MTFInit	Initializes the MTF structure.

Table 13-9 Intel IPP BWT-Based Compression Functions (continued)

Function Short Name	Description
MTFGetSize	Computes the size of the MTF structure.
MTFFwd	Performs the forward MTF transform.
MTFInv	Performs the inverse MTF transform.
RLE Functions	
EncodeRLE	Performs RLE encoding.
DecodeRLE	Performs RLE decoding.
bzip2 Compatible Functions	
EncodeRLEInitAlloc_BZ2	Allocates memory and initializes the bzip2-specific RLE structure.
RLEFree_BZ2	Frees memory allocated for the bzip2-specific RLE structure.
EncodeRLEInit_BZ2	Initializes the bzip2-specific RLE structure.
RLEGetSize_BZ2	Compute the size of the state structure for the bzip2-specific RLE.
EncodeRLE_BZ2	Performs the bzip2-specific RLE.
EncodeRLEFlush_BZ2	Flushes the remaining data after RLE.
RLEGetInUseTable	Gets the pointer to the inUse vector from the RLE state structure.
DecodeRLE_BZ2	Performs the bzip2-specific RLE.
EncodeZ1Z2_BZ2	Performs the bzip2-specific Z1Z2 encoding.
DecodeZ1Z2_BZ2	Performs the bzip2-specific Z1Z2 decoding.
ReduceDictionary	Performs the dictionary reducing.
ExpandDictionary	Performs the dictionary expanding.
CRC32_BZ2	Computes the CRC32 checksum for the source data buffer.
EncodeHuffInitAlloc_BZ2	Allocated memory an initializes the elements of the bzip2-specific internal state for Huffman encoding.
EncodeHuffFree_BZ2	Frees memory allocated for the bzip2-specific Huffman encoding structure.
EncodeHuffInit_BZ2	Initializes the elements of the bzip2-specific internal state for Huffman encoding.
EncodeHuffGetSize_BZ2	Computes the size of the internal state for bzip2-specific Huffman encoding.
PackHuffContext_BZ2	Performs the bzip2-specific encoding of Huffman context.

Table 13-9 Intel IPP BWT-Based Compression Functions (continued)

Function Short Name	Description
EncodeHuff_BZ2	Performs the bzip2-specific Huffman encoding.
DecodeHuffInitAlloc_BZ2	Allocated memory an initializes the elements of the bzip2-specific internal state for Huffman decoding.
DecodeHuffFree_BZ2	Frees memory allocated for the bzip2-specific Huffman decoding structure.
DecodeHuffInit_BZ2	Initializes the elements of the bzip2-specific internal state for Huffman decoding.
DecodeHuffGetSize_BZ2	Computes the size of the internal state for bzip2-specific Huffman decoding.
UnpackHuffContext_BZ2	Performs the bzip2-specific decoding of Huffman context.
DecodeHuff_BZ2	Performs the bzip2-specific Huffman decoding.

Burrows-Wheeler Transform

This section describes data compression functions that perform Burrows-Wheeler Transform (BWT).This method does not compress data, but it simplifies the structure of input data and makes more effective further compression. One of the distinctive feature of this method is operation on the block of data (as a rule of size 512kB - 2 mB). The main idea of this method is block sorting which groups symbols with a similar context. Let us consider how BWT works on the input data block ‘abracadabra’. The first step is to create a matrix containing all its possible cyclic permutations. The first row is input string, the second is created by shifting it to the left by one symbol and so on:

```
abracadabra
bracadabraa
racadabraab
acadabraabr
cadabraabra
adabraabrac
dabraabraca
abraabracad
braabracada
raabracadab
aabracadabr
```

Then all rows are sorted in accordance with the lexicographic order:

```
 0 aabracadabr
 1 abraabracad
 2 abracadabra
 3 acadabraabr
 4 adabraabrac
 5 braabracada
 6 bracadabraa
 7 cadabraabra
 8 dabraabraca
 9 raabracadab
10 racadabraab
```

The last step is to write out the last column and the index of the input string: `rdarcaaaabb`, 2 - this is a result of the forward BWT transform.

Inverse BTW is performed as follows:

elements of the input string are numbered in ascending order

```
 0  r
 1  d
 2  a
 3  r
 4  c
 5  a
 6  a
 7  a
 8  a
 9  b
10  b
```

and sorted in accordance with the lexicographic order:

```
 2  a
 5  a
 6  a
 7  a
 8  a
 9  b
10  b
 4  c
 1  d
 0  r
 3  r
```

This index array is a vector of the inverse transform (`Inv`), the further reconstruction of the string is performed in the following manner:

```
src[] = "rdarcaaaabb";
Inv[] = {2,5,6,7,8,9,10,4,1,0,3};
index = 2; // index of the initial string is known from the forward BWT
for( i = 0; i < len; i++ ) {
  index = Inv[index];
  dst[i] = src[index];
}
```

BWTFwdGetSize

Computes the size of the external buffer for the forward BWT transform.

Syntax

```
IppStatus ippsBWTFwdGetSize_8u(int wndSize, int* pBWTFwdBuffSize);
```

Parameters

wndSize	Window size for BWT transform.
pBWTFwdStateSize	Pointer to the computed size of the additional buffer.

Description

The function `ippsBWTFwdGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required by the function`ippsBWTFwd` for the forward BWT transform.

Example 13-4 shows how to use the function `ippsBWTFwdGetSize_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pBWTFwdBuffSize* pointer is `NULL`.

BWTFwd

Performs the forward BWT transform.

Syntax

```
IppStatus ippsBWTFwd_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
    int* pIndex, Ipp8u* pBWTFwdBuff);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the source and destination vectors.
pIndex	Pointer to the index of first position for the forward BWT transform.
pBWTFwdBuff	Pointer to the additional buffer.

Description

The function `ippsBWTFwd` is declared in the `ippdc.h` file. This function performs the forward BWT transform of *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTFwdBuff*. The size of this buffer should be computed by calling the function `ippsBWTFwdGetSize` beforehand.

Example 13-4 shows how to use the function `ippsBWTFwd_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *len* is less than or equal to 0.

BWTInvGetSize

Computes the size of the external buffer for the inverse BWT transform.

Syntax

```
IppStatus ippsBWTInvGetSize_8u(int wndSize, int* pBWTInvBuffSize);
```

Parameters

wndSize	Window size for BWT transform.
pBWTInvBuffSize	Pointer to the computed size of the additional buffer.

Description

The function `ippsBWTInvGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required by the function `ippsBWTInv` for the inverse BWT transform.

Example 13-4 shows how to use the function `ippsBWTInvGetSize_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pBWTInvBuffSize* pointer is `NULL`.

BWTInv

Performs the inverse BWT transform.

Syntax

```
IppStatus ippsBWTInv_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int index,
    Ipp8u* pBWTInvBuff);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.

len	Number of elements in the source and destination vectors.
index	Iindex of first position for the inverse BWT transform.
pBWTInvBuff	Pointer to the additional buffer.

Description

The function `ippsBWTInv` is declared in the `ippdc.h` file. This function performs the inverse BWT transform of *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTInvBuff*. The size of this buffer should be computed by calling the function `ippsBWTInvGetSize` beforehand.

Example 13-4 shows how to use the function `ippsBWTInv_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *len* is less than or equal to 0.

Example 13-4 Performing the Burrows-Wheeler Transform

```
void func_BWT()
{
    int wndSize = 8;
    int pBWTFwdBuffSize;
    int pBWTInvBuffSize;

    Ipp8u pSrc[] = "baadeffg";
    int len = 8;
    int pIndex;

    Ipp8u* pDst = ippsMalloc_8u(len);
    Ipp8u* pDstInv = ippsMalloc_8u(len);

    ippsBWTFwdGetSize_8u(wndSize, &pBWTFwdBuffSize);
    Ipp8u* pBWTFwdBuff = ippsMalloc_8u(pBWTFwdBuffSize);
    ippsBWTFwd_8u(pSrc, pDst, len, &pIndex, pBWTFwdBuff);

    ippsBWTInvGetSize_8u( wndSize, &pBWTInvBuffSize);
    Ipp8u* pBWTInvBuff = ippsMalloc_8u(pBWTInvBuffSize);
    ippsBWTInv_8u(pDst, pDstInv, len, pIndex, pBWTInvBuff);
}
Result:
    pDst ->     "bagadeff"
    pDstInv ->  "baadeffg"
```

BWTGetSize_SmalBlock

Computes the size of the external buffer for the BWT transforms for small data block.

Syntax

```
IppStatus ippsBWTGetSize_SmallBlock_8u(int wndSize, int* pBuffSize);
```

Parameters

wndSize	Window size for BWT transform.
pBuffSize	Pointer to the computed size of the additional buffer.

Description

The function `ippsBWTGetSize_SmallBlock` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required for the forward and inverse BWT transforms of the small data block by the functions `ippsBWTFwd_SmallBlock` and `ippsBWTInv_SmallBlock` respectively. The size of the data block must be in the range (0, 32768].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pBuffSize* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error if *wndSize* is less than or equal to 0, or is greater than 32768.

BWTFwd_SmallBlock

Performs the forward BWT transform for small data block.

Syntax

```
IppStatus ippsBWTFwd_SmallBlock_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
    int* pIndex, Ipp8u* pBWTBuff);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the source and destination vectors.
pIndex	Pointer to the index of first position for the forward BWT transform.
pBWTBuff	Pointer to the additional buffer.

Description

The function `ippsBWTFwd_SmallBlock` is declared in the `ippdc.h` file. This function performs the forward BWT transform od small block of data, that the number of elements must be in the range (0, 32768]. The function processes *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTBuff*. The size of this buffer must be computed by calling the function `ippsBWTGetSize_SmalBlock` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *len* is less than or equal to 0, or is greater than 32768.

BWTInv_SmallBlock

Performs the inverse BWT transform for small data block.

Syntax

```
IppStatus ippsBWTInv_SmallBlock_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int
    index, Ipp8u* pBWTBuff);
```

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the source and destination vectors.

index	Index of first position for the inverse BWT transform.
pBWTBuff	Pointer to the additional buffer.

Description

The function ippsBWTInv_SmallBlock is declared in the ippdc.h file. This function performs the inverse BWT transform od small block of data, that the number of elements must be in the range (0, 32768]. The function processes *len* elements starting from *index* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTBuff*. The size of this buffer must be computed by calling the function ippsBWTGetSize_SmalBlock beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *len* is less than or equal to 0, or is greater than 32768; or if*index*x is greater than or equal to .

Generalized Interval Transformation Coding

This section describes Intel IPP data compression functions that perform Generalized Interval Transformation (GIT) coding. The coding procedure is of a statistical entropy nature. The GIT algorithm is based on Interval Transformations also known as Distance coding or Inversion frequencies [Arn97] and Rice-Golomb coding [Gal75]. This algorithm compresses blocks of data at ratios that are close to Huffman encoding compression ratios. However, with low entropy source data, the GIT algorithm typically provides coding at higher compression ratios as compared with the Huffman algorithm.

Special Arguments

The GIT coding functions give you possibility to adjust the compression algorithm specifying the *strategyHint* argument. The *strategyHint* argument suggests using different type of reordering the symbols in the alphabet in dependence on the input data. In some cases it can improve encoding. Table 13-10 lists the possible values of the *strategyHint* argument and their meanings.

Table 13-10 strategyHint Argument for GIT Encoding Functions

Value	Descriptions
ippGITNoStrategy	No strategy.

Table 13-10 strategyHint Argument for GIT Encoding Functions

Value	Descriptions
ippGITLeftReorder	Alphabet is reordered in such a way that more frequent symbols have less numbers (moved to the left).
ippGITRightReorder	Alphabet is reordered in such a way that more frequent symbols have greater numbers (move to right)
ippGITFixedOrder	Fixed alphabet order, no reordering

EncodeGITInitAlloc

Allocates memory and initializes the GIT encoding state structure.

Syntax

```
IppStatus ippsEncodeGITInitAlloc_8u (int maxSrcLen, int maxDstLen,
    IppGITState_8u** ppGITState);
```

Parameters

maxSrcLen	Maximum length of the source buffer.
maxDstLen	Maximum length of the destination buffer.
ppGITState	Double pointer to the GIT encoding state structure.

Description

The function `ippsEncodeGITInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the GIT encoding state structure *ppGITState*. This structure is used by the function `ippsEncodeGIT`.

The size of the GIT encoding state structure depends on the values of the *maxSrcLen* and *maxDstLen* parameters that set the upper bound on the sizes of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

The *maxSrcLen* parameter sets the size of a block for input data processing. Due to the adaptive coding used by the GIT algorithm, the compression ratio is very likely to improve as the block size increases.

Return Values

IppStsNoErr	Indicates no errors.
IppStsSizeErr	Indicates an error if *maxSrcLen* or *maxDstLen* is less than or equal to 0.

GITFree

Frees memory allocated for the GIT coding structures.

Syntax

```
void ippsGITFree_8u (IppGITState_8u* pGITState);
```

Parameters

pGITState	Pointer to the GIT encoding or decoding state structure.

Description

The function ippsGITFree is declared in the ippdc.h file. This function frees memory allocated for the GIT encoding or decoding state structure by the functions ippsEncodeGITInitAlloc or ippsDecodeGITInitAlloc, respectively.

EncodeGITInit

Initializes the GIT encoding state structure.

Syntax

```
IppStatus ippsEncodeGITInit_8u (int maxSrcLen, int maxDstLen,
    IppGITState_8u* pGITState);
```

Parameters

maxSrcLen	Maximum length of the source buffer.
maxDstLen	Maximum length of the destination buffer.
pGITState	Pointer to the GIT encoding state structure.

Description

The function `ippsEncodeGITInit` is declared in the `ippdc.h` file. This function initializes the GIT encoding state structure in the external buffer whose size should be computed previously by the calling the function `ippsEncodeGITGetSize`. Alternatively this structure can be initialized by the function `ippsEncodeGITInitAlloc`.

The size of the GIT encoding state structure depends on the values of the *maxSrcLen* and *maxDstLen* parameters, which set the upper bound on the sizes of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

The *maxSrcLen* parameter sets the size of a block for input data processing. Due to adaptive coding used by the GIT algorithm, the compression ratio is very likely to improve as the block size increases.

NOTE. The *maxSrcLen* and *maxDstLen* parameters of the `ippsEncodeGITInit` and `ippsEncodeGITGetSize` functions must have equal respective values to avoid errors in the `ippsEncodeGIT` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the pointer *pGITState* is `NULL`.
`ippStsSizeErr`	Indicates an error if *maxSrcLen* or *maxDstLen* is less than or equal to 0.

EncodeGITGetSize

Computes the size of the GIT encoding state structure.

Syntax

```
IppStatus ippsEncodeGITGetSize_8u (int maxSrcLen, int maxDstLen, int*
    pGITStateSize);
```

Parameters

maxSrcLen	Maximum length of the source buffer.
maxDstLen	Maximum length of the destination buffer.
pGITStateSize	Pointer to the size of the GIT encoding state structure.

Description

The function `ippsEncodeGITGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the GIT encoding state structure. Its size depends on the *maxSrcLen* and *maxDstLen* parameters, which are the maximum lengths of the input and output data buffers, respectively.

The function should be called prior to the function `ippsEncodeGITInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the pointer *pGITStateSize* is `NULL`.
`ippStsSizeErr`	Indicates an error if *maxSrcLen* or *maxDstLen* is less than or equal to 0.

EncodeGIT

Performs GIT encoding.

Syntax

```
IppStatus ippsEncodeGIT_8u (const Ipp8u* pSrc, int srcLen, Ipp8u* pDst,
    int* pDstLen, IppGITStrategyHint strategyHint, IppGITState_8u* pGITState);
```

Parameters

pSrc	Pointer to the source buffer.
srcLen	Length of the source buffer.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
strategyHint	Suggests using of the code implemented specific strategy for lexicorgaphical reordering (see Table 13-10); default value is `ippGITNoStrategyHint`.

pGITState	Pointer to the GIT encoding state structure.

Description

The function ippsEncodeGIT is declared in the ippdc.h file. This function performs GIT encoding. It processes the *srcLen* bytes of the input data *pSrc* and writes the results to the *pDst* buffer. The function uses the GIT encoding state structure *pGITState* that contains pointers to different data structures required for operation. The structure must be initialized by the functions ippsEncodeGITInitAlloc or ippsEncodeGITInit beforehand.

It is not recommended to use source data buffer less than 8Kb.

WARNING. Value of *srcLen* must not exceed that of *maxSrcLen* specified in the function ippsEncodeGITInitAlloc or ippsEncodeGITInit.

Nevertheless, the larger files can be encoded by subsequently calling the ippsEncodeGIT function several times (see Example 13-5 below).

Encoding quality depends on the input data and in some cases can be improved by specifying the *strategyHint* parameter (see Table 13-10).

The default value for the *strategyHint* parameter is ippGITNoStrategyHint.

The code Example 13-5 shows how to use the ippsEncodeGIT_8u and supporting functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if *srcLen* is greater than the value of the *maxSrcLen* parameter passed to ippsEncodeGITGetSize or ippsEncodeGITInitAlloc.

Example 13-5 Using the Function EncodeGIT and Supporting Functions

```
#define BLOCKSIZE 65536
................
FILE*                in, out;
Ipp8u*               data, code;
IppGITState_8u*      pGITState;
int                  GITStateSize, size, codeLen;
```

Example 13-5 Using the Function EncodeGIT and Supporting Functions

```
IppGITStrategyHint  strategyHint;
................

/*************************************************************/
/* Opening the file containing input data and the file       */
/* for the compressed data:                                  */
/*************************************************************/
in = fopen("datafile.txt", "rb");
out = fopen("codefile", "wb");

/*************************************************************/
/* Memory allocation for input data and output code buffers, */
/* and for the internal encoding state structure:            */
/*************************************************************/
data = (Ipp8u*) malloc(BLOCKSIZE * sizeof(Ipp8u));
code = (Ipp8u*) malloc((BLOCKSIZE << 1) * sizeof(Ipp8u));
ippsEncodeGITGetSize_8u(BLOCKSIZE, (BLOCKSIZE << 1), &GITStateSize);
pGITState = (IppGITState_8u*) malloc(GITStateSize * sizeof(Ipp8u));

/*************************************************************/
/* Initializing the memory, allocated for internal encoding  */
/* state structure and setting the encoding strategy:        */
/*************************************************************/
ippsEncodeGITInit_8u(BLOCKSIZE, (BLOCKSIZE << 1), pGITState);
strategyHint = ippGITLeftReorder;
/*************************************************************/
/* The main loop. In every iteration program reads the size   */
/* bytes from the input data file to data array and          */
/* compresses by calling the ippsEncodeGIT_8u function.      */
/* ippsEncodeGIT_8u places the compressed data to the code   */
/* array and returns the length of compressed data in bytes  */
/* trough the &codeLen argument. After that program writes   */
/* the value of codeLen and codeLen bytes of compressed data */
/* to the output data file.                                  */
/*************************************************************/
for( ; ; )
{
    size = fread(&data[0], sizeof(Ipp8u), BLOCKSIZE, in);
    if(size <= 0)
      break;
    if(ippsEncodeGIT_8u(data, size, code, &codeLen, strategyHint,
pGITState) != ippStsNoErr)
      return (-1);
    fwrite(&codeLen, sizeof(int), 1, out);
    fwrite(code, sizeof(Ipp8u), codeLen, out);
} /* for */

/*************************************************************/
/* Free the memory allocated for input data and output       */
/* code buffers and for the internal encoding state          */
/* structure:                                                */
```

Example 13-5 Using the Function EncodeGIT and Supporting Functions

```
/*************************************************************/
free(pGITState);
free(data);
free(code);
fclose(in);
fclose(out);
....................
```

DecodeGITInitAlloc

Allocates memory and initializes the GIT decoding state structure.

Syntax

```
IppStatus ippsDecodeGITInitAlloc_8u (int maxSrcLen, int maxDstLen,
    IppGITState_8u** ppGITState);
```

Parameters

maxSrcLen	Maximum length of the source buffer.
maxDstLen	Maximum length of the destination buffer.
ppGITState	Double pointer to the GIT decoding state structure.

Description

The function ippsDecodeGITInitAlloc is declared in the ippdc.h file. This function allocates memory and initializes the GIT decoding state structure *ppGITState*. This structure is used by the function ippsDecodeGIT.

The size of the GIT decoding state structure depends on the value of the *maxDstLen* parameter, which sets the upper bound on the lengths of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *ppGITState* is NULL.
ippStsSizeErr	Indicates an error if *maxSrcLen* or *maxDstLen* is less than or equal to 0.

DecodeGITInit

Initializes the GIT decoding state structure.

Syntax

```
IppStatus ippsDecodeGITInit_8u (int maxDstLen, IppGITState_8u* pGITState);
```

Parameters

maxDstLen	Maximum length of the destination buffer.
pGITState	Pointer to memory allocated for the GIT decoding state structure.

Description

The function `ippsDecodeGITInit` is declared in the `ippdc.h` file. This function initializes the GIT decoding state structure *pGITState* in the external buffer. The size of this buffer should be determined by the function `ippsEncodeGITGetSize`. Alternatively this structure can be initialized by the function `ippsDecodeGITInitAlloc`.

The size of the GIT decoding state structure depends on the value of the *maxDstLen* parameter, which sets the upper bound on the lengths of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

NOTE. The *maxDstLen* parameter of the `ippsDecodeGITInit` and `ippsDecodeGITGetSize` functions must be the same to avoid an error in the function `ippsDecodeGIT`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the pointer *pGITState* is `NULL`.
`ippStsSizeErr`	Indicates an error if *maxDstLen* is less than or equal to 0.

DecodeGITGetSize

Computes the size of the GIT decoding state structure.

Syntax

```
IppStatus ippsDecodeGITGetSize_8u (int maxSrcLen, int* pGITStateSize);
```

Parameters

maxSrcLen	Maximum length of the source buffer.
pGITStateSize	Pointer to the size of the GIT decoding state structure.

Description

The function `ippsDecodeGITGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes *pGITStateSize* of the GIT decoding state structure. Its value depends on the *maxSrcLen* parameter, which is the maximum length of the input data buffer.

The function should be called prior to the function `ippsDecodeGITInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the pointer *pGITStateSize* is `NULL`.
`ippStsSizeErr`	Indicates an error if *maxSrcLen* is less than or equal to 0.

DecodeGIT

Performs GIT decoding.

Syntax

```
IppStatus ippsDecodeGIT_8u (const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int*
    pDstLen, IppGITStrategyHint strategyHint, IppGITState_8u* pGITState);
```

Parameters

pSrc	Pointer to the source buffer.

srcLen	Length of the source buffer.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
strategyHint	Suggests using of the code implemented specific strategy for lexicorgaphical reordering (see Table 13-10); default value is `ippGITNoStrategyHint`.
pGITState	Pointer to the GIT decoding state structure.

Description

The function `ippsDecodeGIT` is declared in the `ippdc.h` file. This function performs GIT decoding - it decodes data encoded by the `ippsEncodeGIT` function.

The `ippsDecodeGIT` uses the GIT state structure *pGITState* that contains pointers to different data structures required for GIT decoding. This structure must be initialized by the functions `ippsDecodeGITInitAlloc` or `ippsDecodeGITInit` beforehand.
The function `ippsDecodeGIT` assumes that there is enough memory for decoding. It obtains the source data from the *pSrc* source vector with the *srcLen* length and writes the decoded data to the *pDst* destination vector. The size of destination buffer must be equal to or greater than *maxDstLen* (see `ippsDecodeGITInitAlloc` and `ippsDecodeGITInit`). *pDstLen* is an output parameter, to which the function writes the actual size of the decoded data.

The code Example 13-6 shows how to use the `ippsDecodeGIT_8u` and supporting functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if there is not enough memory allocated for the destination buffer.

Example 13-6 Using the Function `DecodeGIT` and Supporting Functions

```
#define BLOCKSIZE 65536
.....................
FILE* in, out;
Ipp8u* rdata, code;
int GITStateSize, size, codeLen, rdataLen;
IppGITState_8u pGITState;
IppGITStrategyHint strategyHint;
```

Example 13-6 Using the Function DecodeGIT and Supporting Functions

```
.....................
/*************************************************************/
/* Opening the file containing input compressed data and the */
/* file  for the restored (uncompressed) data:               */
/*************************************************************/
in = fopen("codefile", "rb");
out = fopen("datafile_restored.txt", "wb");

/*************************************************************/
/* Memory allocation for input code and output restored data */
/* buffers, and for the internal decoding state structure:   */
/*************************************************************/
rdata = (Ipp8u*) malloc((BLOCKSIZE << 1) * sizeof(Ipp8u));
code = (Ipp8u*) malloc(BLOCKSIZE * sizeof(Ipp8u));
ippsDecodeGITGetSize_8u((BLOCKSIZE << 1), &GITStateSize);
pGITState = (IppGITState_8u*) malloc(GITStateSize * sizeof(Ipp8u));

/*************************************************************/
/* Initializing the memory, allocated for internal decoding  */
/* state structure and setting the decoding strategy         */
/* (should be the same as the strategy for encoding) :       */
/*************************************************************/
ippsDecodeGITInit_8u(BLOCKSIZE, pGITState);
strategyHint = ippGITLeftReorder;

/*************************************************************/
/* The main loop. On every iteration program reads the size   */
/* of the compressed block (codeLen) from the input code     */
/* file. After that program reads the codeLen bytes of code  */
/* from the input code file to the code array. The call of   */
/* ippsDecodeGIT_8u performs the decoding of codeLen bytes   */
/* of code to the rdata error of rdataLen. At the end        */
/* program writes the rdataLen bytes of restored data(rdata) */
/* to the output file.                                       */
/*************************************************************/
for( ; ; )
{
   size = fread(&codeLen, sizeof(int), 1, in);
   if(size <= 0)
     break;
   fread(&code[0], sizeof(Ipp8u), codeLen, in);
   ippsDecodeGIT_8u(code, codeLen, rdata, &rdataLen, strategyHint,
pGITState);
   fwrite(rdata, sizeof(Ipp8u), rdataLen, out);
} /* for */

/*************************************************************/
/* Free the memory allocated for input code and output       */
/* restored data buffers and for the internal decoding state */
/* structure:                                                */
/*************************************************************/
```

Example 13-6 Using the Function `DecodeGIT` and Supporting Functions

```
free(pGITState);
free(code);
free(rdata);
fclose(in);
fclose(out);
............
```

Move To Front Functions

This section describes the functions that performs Move To Front (MTF) data transform method. The basic idea is to represent the symbols of the source sequence as the current indexes of that symbols in the modified alphabet. This alphabet is a list where frequently used symbols are placed in the upper lines. When the given symbols occurs it is replaced by its index in the list, then this symbol is moved in the first position in the list, and all indexes are updated. For example, the sequence "`baabbfffacczzdd`" contains symbols that form the ordered 'alphabet'`{'a', 'b', 'c', 'd', 'f', 'z'}`. The function will operate in the following manner:

Table 13-11

source	destination	alphabet
		0, 1, 2, 3, 4, 5, 6
		'a', 'b', 'c', 'd', 'f', 'z'
b	1	'b', 'a', 'c', 'd', 'f', 'z'
a	1	'a', 'b', 'c', 'd', 'f', 'z'
a	0	'a', 'b', 'c', 'd', 'f', 'z'
b	1	'b', 'a', 'c', 'd', 'f', 'z'
f	4	'f', 'b', 'a', 'c', 'd', 'z'
f	0	'f', 'b', 'a', 'c', 'd', 'z'
f	0	'f', 'b', 'a', 'c', 'd', 'z'
a	2	'a', 'f', 'b', 'c', 'd', 'z'
c	3	'c', 'a', 'f', 'b','d', 'z'
c	0	'c', 'a', 'f', 'b', 'd', 'z'
z	5	'z', 'c', 'a', 'f', 'b', 'd'
z	0	'z', 'c', 'a', 'f', 'b', 'd'

Table 13-11

source	destination	alphabet
d	5	'd', 'z', 'c', 'a', 'f', 'b'
d	0	'd', 'z', 'c', 'a', 'f', 'b'

Finally, the function returns the destination sequence: 11014002305050.

These transformed data can be used for the following effective compression. This method is often used after Burrows-Wheeler transform.

MTFInitAlloc

Allocates memory and initializes the MTF structure.

Syntax

```
IppStatus ippsMTFInitAlloc_8u(IppMTFState_8u** ppMTFState);
```

Parameters

ppMTFState	Double pointer to the MTF transform structure.

Description

The function `ippsMTFInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the MTF structure that contains required parameters for the MTF transform. This structure is used by the functions `ippsMTFFwd` and `ippsMTFInv`.

Example 13-7 shows how to use the function `ippsMTFInitAlloc_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *ppMTFState* pointer is `NULL`.
`ippMemAllocErr`	Indicates an error if no memory is allocated.

MTFFree

Frees memory allocated for the MTF structure.

Syntax

```
void ippsMTFFree_8u(IppMTFState_8u* pMTFState);
```

Parameters

pMTFState	Pointer to the MTF structure.

Description

The function `ippsMTFFree` is declared in the `ippdc.h` file. This function frees memory allocated for the MTF structure by the function `ippsMTFInitAlloc`.

MTFInit

Initializes the MTF structure.

Syntax

```
IppStatus ippsMTFInit_8u(IppMTFState_8u* pMTFState);
```

Parameters

pMTFState	Pointer to the MTF structure.

Description

The function `ippsMTFInit` is declared in the `ippdc.h` file. This function initializes the MTF structure that contains parameters for the MTF transform in the external buffer. This structure is used by the functions `ippsMTFFwd` and `ippsMTFInv`. The size of this buffer should be computed previously by calling the function `ippsMTFGetSize`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pMTFState* pointer is `NULL`.

MTFGetSize

Computes the size of the MTF structure.

Syntax

```
IppStatus ippsMTFGetSize_8u(int* pMTFStateSize);
```

Parameters

pMTFStateSize	Pointer to the computed MTF structure size.

Description

The function `ippsMTFGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) that is required for the MTF structure. Call this function prior to the function `ippsMTFInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pMTFStateSize* pointer is `NULL`.

MTFFwd

Performs the forward MTF transform.

Syntax

```
IppStatus ippsMTFFwd_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, IppMTFState_8u*
    pMTFState);
```

Parameters

pSrc	Pointer to the source buffer.
pDst	Pointer to the destination buffer.
srcLen	Number of elements in the source and destination buffers.
pMTFState	Pointer to the MTF structure.

Description

The function `ippsMTFFwd` is declared in the `ippdc.h` file. This function performs the forward MTF transform of *len* elements of the data in the source buffer *pSrc* and stores result in the buffer *pDst*. The parameters of the MTF transform are specified in the MTF structure *pMTFState* that should be initialized by the functions ippsMTFInitAlloc or ippsMTFInit beforehand.

Example 13-7 shows how to use the function `ippsMTFFwd_8u`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *len* is less than or equal to 0.

Example 13-7 Using the Function ippsMTFFwd

```
void func_MTF()
{
    IppMTFState_8u* ppMTFState;
    Ipp8u pSrc[] = "adadasdasd";
    Ipp8u  pDst[10];
    int len = 10;

    ippsMTFInitAlloc_8u(&ppMTFState);
    ippsMTFFwd_8u( pSrc, pDst, len, ppMTFState);
}
Result:  pDst -> 97 100 1 1 1 115 2 2 2 2
```

MTFInv

Performs the inverse MTF transform.

Syntax

```
IppStatus ippsMTFInv_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
    IppMTFState_8u* pMTFState);
```

Parameters

pSrc	Pointer to the source buffer.
pDst	Pointer to the destination buffer.

len	Number of elements in the source and destination buffers.
pMTFState	Pointer to the MTF structure.

Description

The function `ippsMTFInv` is declared in the `ippdc.h` file. This function performs the inverse MTF transform of *len* elements of data in the source buffer *pSrc* and stores result in the buffer *pDst*. The parameters of the MTF transform are specified in the MTF structure *pMTFState* that should be initialized by the functions ippsMTFInitAlloc or ippsMTFInit beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *len* is less than or equal to 0.

Run Length Encoding Functions

This section describes functions that performs data compression using Run Length Encoding (RLE) method. The main idea of this method is to replace the sequence of the same symbol by a number of its occurrences. For example, the string "`aaabbbbaaaaabbabbbbbb`" will be transformed to "`aa1bb2aa3bb0abb4`". The important parameters of the RLE method is a threshold - that is a minimum number of the repeated symbols exceeding of which starts up the encoding procedure. In the Intel IPP data compression functions this threshold is set to 2. Upper bound of the number of repetitions is 255 - to use one byte for repetition counter.

EncodeRLE

Performs RLE encoding.

Syntax

```
IppStatus ippsEncodeRLE_8u(const Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int*
    pDstLen);
```

Parameters

ppSrc	Double pointer to the source data buffer.

pSrcLen	Pointer to the number of elements in the source buffer after encoding point to the number of remaining elements.
pDst	Pointer to the destination data buffer.
pDstLen	Pointer to the number of elements in the destination buffer, after encoding to the actual number of elements in the destination buffer.

Description

The function `ippsEncodeRLE` is declared in the `ippdc.h` file. This function performs RLE encoding *pSrcLen* elements of the source data buffer *ppSrc* and stores the result in the buffer *pDst*. After encoding the function updates the *pSrcLen* thus it points to the number of remaining elements of the source buffer, and *pDstLen* points to the actual number of elements stored in the destination buffer.

Sometimes the number of output elements can exceed the specified size of destination buffer. In this case the function returns the warning message and pointer to the number of not-encoded source elements. The user can call the function `ippsEncodeRLE` again to complete encoding.

Example 13-8 shows how to use the function `ippsEncodeRLE_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specicifed pointers is null.
`ippStsSizeErr`	Indicates an error if *pSrcLen* or *pDstLen* is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

DecodeRLE

Performs RLE decoding.

Syntax

```
IppStatus ippsDecodeRLE_8u(const Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen);
```

Parameters

ppSrc	Double pointer to the source data buffer.

pSrcLen	Pointer to the number of elements in the source buffer, after decoding pointer to the number of remaining elements.
pDst	Pointer to the destination data buffer.
pDstLen	Pointer to the number of elements in the destination buffer, after encoding to the actual number of elements in the destination buffer.

Description

The function `ippsDecodeRLE` is declared in the `ippdc.h` file. This function performs RLE decoding *srcLen* elements of the source data buffer *ppSrc* and stores the result in the destination buffer *pDst*. After decoding the function updates the *pSrcLen* thus it points to the number of remaining elements of the source buffer, and *pDstLen* points to the actual number of elements stored in the destination buffer.

Sometimes the number of output elements can exceed the specified size of destination buffer. In this case the function returns the warning message and pointer to the number of not-decoded source elements. The user can call the function `ippsDecodeRLE` again to complete decoding.

Example 13-8 shows how to use the function `ippsDecodeRLE_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specicifed pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if *pSrcLen* or *pDstLen* is less than or equal to 0.
`ippStsSrcDataErr`	Indicates an error if *ppSrc* contains unsupported data.
`ippStsDstSizeLessExpected`	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

Example 13-8 Using RLE Encoding and Decoding Functions

```
void func_RLE()
{
    Ipp8u* pSrc = (Ipp8u*)"aaaddddddssadsaaddd";
    int pSrcLen = 18;
    int pDstLen = 18;
    int pDstDLen = 18;

    Ipp8u* pDst = ippsMalloc_8u(pDstLen);
    Ipp8u* pDstD = ippsMalloc_8u(pDstDLen);
    ippsEncodeRLE_8u(&pSrc, &pSrcLen, pDst, &pDstLen);
```

Example 13-8 Using RLE Encoding and Decoding Functions

```
    ippsDecodeRLE_8u(&pDst, &pDstLen, pDstD, &pDstDLen);
}

Result:   pDst:  aa1dd3ss0adsaa0dd1
          pDstD: aaaddddddssadsaaddd
```

bzip2 Coding Functions

EncodeRLEInitAlloc_BZ2

Allocates memory and initializes the bzip2-specific RLE structure.

Syntax

```
IppStatus ippsEncodeRLEInitAlloc_BZ2_8u(IppRLEState_BZ2** ppRLEState);
```

Parameters

ppRLEState	Double pointer to the bzip2-specific RLE structure.

Description

The function `ippsEncodeRLEInitAlloc_BZ2` is declared in the `ippdc.h` file. This function allocates memory and initializes the bzip2-specific RLE structure that contains required parameters for the RLE. This structure is used by the function `ippsEncodeRLE_BZ2`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *ppRLEState* pointer is `NULL`.

RLEFree_BZ2

Frees memory allocated for the bzip2-specific RLE structure.

Syntax

```
void ippsRLEFree_BZ2_8u(IppRLEState_BZ2* pRLEState);
```

Parameters

pRLEState	Pointer to the bzip2-specific RLE structure.

Description

The function `ippsRLEFree_BZ2` is declared in the `ippdc.h` file. This function frees memory allocated for bzip2-specific RLE structure by the function `ippsEncodeRLEInitAlloc_BZ2`.

EncodeRLEInit_BZ2

Initializes the bzip2-specific RLE structure.

Syntax

```
IppStatus ippsEncodeRLEInit_BZ2_8u(IppRLEState_BZ2* pRLEState);
```

Parameters

pRLEState	Pointer to the bzip2-specific RLE structure.

Description

The function `ippsEncodeRLEInit_BZ2` is declared in the `ippdc.h` file. This function initializes the bzip2-specific RLE structure that contains parameters for the RLE in the external buffer. This structure is used by the function `ippsEncodeRLE_BZ2`. The size of this buffer should be computed previously by calling the function `ippsRLEGetSize_BZ2`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pRLEState* pointer is `NULL`.

RLEGetSize_BZ2

Compute the size of the state structure for the bzip2-specific RLE.

Syntax

```
IppStatus ippsRLEGetSize_BZ2_8u(int* pRLEStateSize);
```

Parameters

pRLEStateSize	Pointer to the size of the state structure for bzip2-specific RLE.

Description

The function `ippsRLEGetSize_BZ2` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the internal state structure for the bzip2-specific RLE.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pRLEStateSize* pointer is `NULL`.

EncodeRLE_BZ2

Performs the bzip2-specific RLE.

Syntax

```
IppStatus ippsEncodeRLE_BZ2_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int*
    pDstLen, IppRLEState_BZ2* pRLEState);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pRLEState	Pointer to the bzip2-specific RLE state structure.

Description

The function `ippsEncodeRLE_BZ2` is declared in the `ippdc.h` file. This function performs RLE encoding with thresholding equal to 4. It processes the input data *ppSrc* and writes the results to the *pDst* buffer. The function uses the bzip2-specific RLE state structure *pRLEState*. This structure must be initialized by the functions `ippsEncodeRLEInitAlloc_BZ2` or `ippsEncodeRLEInit_BZ2` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.

ippStsSizeErr	Indicates an error if length of the source or destination buffer is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

EncodeRLEFlush_BZ2

Flushes the remaining data after RLE.

Syntax

```
IppStatus ippsEncodeRLEFlush_BZ2_8u(Ipp8u* pDst, int* pDstLen, IppRLEState_BZ2*
    pRLEState);
```

Parameters

pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pRLEState	Pointer to the bzip2-specific RLE state structure.

Description

The function `ippsEncodeRLEFlush_BZ2` is declared in the `ippdc.h` file. This function flushes the remaining data after RLE encoding with thresholding equal to 4. The function uses the initialized bzip2-specific RLE state structure *pRLEState*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the destination buffer is less than or equal to 0.

RLEGetInUseTable

Gets the pointer to the `inUse` vector from the RLE state structure.

Syntax

```
IppStatus ippsRLEGetInUseTable_8u(Ipp8u inUse[256], IppRLEState_BZ2*
    pRLEState);
```

Parameters

inUse	Pointer to the `inUse` vector.
pRLEState	Pointer to the bzip2-specific RLE state structure.

Description

The function `ippsRLEGetInUseTable` is declared in the `ippdc.h` file. This function gets the pointer to the *inUse* vector (table) from the initialized bzip2-specific RLE state structure *pRLEState*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.

DecodeRLE_BZ2

Performs the bzip2-specific RLE.

Syntax

```
IppStatus ippsDecodeRLE_BZ2_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int*
    pDstLen);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer, after decoding - pointer to the size of the remaining data.

pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer, after decoding - pointer to the resulting size of the destination biffer.

Description

The function `ippsDecodeRLE_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific RLE decoding with thresholding equal to 4.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the source or destination buffer is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

EncodeZ1Z2_BZ2

Performs the bzip2-specific Z1Z2 encoding.

Syntax

```
IppStatus ippsEncodeZ1Z2_BZ2_8u16u(Ipp8u** ppSrc, int* pSrcLen, Ipp16u* pDst,
    int* pDstLen, int freqTable[258]);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer, after decoding - pointer to the size of the remaining data.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer, after decoding - pointer to the resulting size of the destination buffer.
freqTable	Table of frequencies collected for the alphabet symbols.

Description

The function `ippsEncodeZ1Z2_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific Z1Z2 encoding.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the source or destination buffer is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

DecodeZ1Z2_BZ2

Performs the bzip2-specific Z1Z2 decoding.

Syntax

```
IppStatus ippsDecodeZ1Z2_BZ2_16u8u(Ipp16u** ppSrc, int* pSrcLen, Ipp8u* pDst,
    int* pDstLen);
```

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer, after decoding - pointer to the size of the remaining data.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer, after decoding - pointer to the resulting size of the destination buffer.

Description

The function `ippsDecodeZ1Z2_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific Z1Z2 decoding.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.

ippStsSizeErr	Indicates an error if length of the source or destination buffer is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

ReduceDictionary

Performs the dictionary reducing.

Syntax

```
IppStatus ippsReduceDictionary_8u_I(const Ipp8u inUse[256], Ipp8u* pSrcDst, int
    srcDstLen, int* pSizeDictionary);
```

Parameters

inUse	Table of 256 values of the Ipp8u type.
pSrcDst	Pointer to the source and destination buffer.
srcDstLen	Length of the source and destination buffer.
pSizeDictionary	Pointer to the size of the dictionary on entry, and to the size of reduced dictionary after operation.

Description

The function `ippsReduceDictionary` is declared in the `ippdc.h` file. This function performs the dictionary reducing.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is `NULL`.
ippStsSizeErr	Indicates an error if length of the source and destination buffer is less than or equal to 0.

ExpandDictionary

Performs the dictionary expanding.

Syntax

```
IppStatus ippsExpandDictionary_8u_I(const Ipp8u inUse[256], Ipp8u* pSrcDst, int
    srcDstLen, int sizeDictionary);
```

Parameters

inUse	Table of 256 values of the Ipp8u type.
pSrcDst	Pointer to the source and destination buffer.
srcDstLen	Length of the source and destination buffer.
sizeDictionary	Size of the dictionary on entry, and to the size of expanded dictionary after operation.

Description

The function `ippsExpandDictionary` is declared in the `ippdc.h` file. This function performs the dictionary expanding.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the source and destination buffer is less than or equal to 0.

CRC32_BZ2

Computes the CRC32 checksum for the source data buffer.

Syntax

```
IppStatus ippsCRC32_BZ2_8u(const Ipp8u* pSrc, int srcLen, Ipp32u* pCRC32);
```

Parameters

pSrc	Pointer to the source data buffer.
srcLen	Number of elements in the source data buffer.
pCRC32	Pointer to the accumulated checksum value.

Description

The function `ippsCRC32_BZ2` is declared in the `ippdc.h` file. This function computes the checksum for *srcLen* elements of the source data buffer *pSrc* and stores it in the *pCRC32*. The checksum is computed using the CRC32 direct algorithm that is specific for the bzip2 coding.

You can use this function to compute the accumulated value of the checksum for multiple buffers by specifying as an input parameter the checksum value obtained in the preceding function call.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error if the length of the source vector is less than or equal to 0.

EncodeHuffGetSize_BZ2

Computes the size of the internal state for bzip2-specific Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuffGetSize_BZ2_16u8u(int wndSize, int*
    pEncodeHuffStateSize);
```

Parameters

wndSize	Size of the block to be processed.
pEncodeHuffStateSize	Pointer to the size of the internal state for bzip2-specific Huffman coding.

Description

The function ippsEncodeHuffGetSize_BZ2 is declared in the ippdc.h file. This function computes the size of the internal state structure for bzip2-specific Huffman encoding in dependence of the size of the block to be encoded.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pEncodeHuffStateSize* is NULL.
ippStsSizeErr	Indicates an error if *wndSize* is less than or equal to 0.

EncodeHuffInit_BZ2

Initializes the elements of the bzip2-specific internal state for Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuffInit_BZ2_16u8u(int sizeDictionary, const int
    freqTable[258], const Ipp16u* pSrc, int srcLen, IppEncodeHuffState_BZ2*
    pEncodeHuffState);
```

Parameters

sizeDictionary	Size of the dictionary.
freqTable	Table of frequencies of symbols.
pSrc	Pointer to the source vector.
srcLen	Length of the source vector.
pEncodeHuffState	Pointer to internal state structure for bzip2 specific Huffman coding.

Description

The function ippsEncodeHuffInit_BZ2 is declared in the ippdc.h file. This function initializes the elements of the bzip2-specific internal state for Huffman encoding. This structure is used by the function ippsEncodeHuff_BZ2. The size of this buffer should be computed previously by calling the function ippsEncodeHuffGetSize_BZ2.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the source buffer is less than or equal to 0.

EncodeHuffInitAlloc_BZ2

Allocated memory an initializes the elements of the bzip2-specific internal state for Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuffInitAlloc_BZ2_16u8u(int wndSize, int sizeDictionary,
    const int freqTable[258], const Ipp16u* pSrc, int srcLen,
    IppEncodeHuffState_BZ2** ppEncodeHuffState);
```

Parameters

wndSize	Size in bytes of the block to be processed.
sizeDictionary	Size of the dictionary.
freqTable	Table of frequencies of symbols.
pSrc	Pointer to the source vector.
srcLen	Length of the source vector.
ppEncodeHuffState	Double pointer to the state structure for bzip2 specific Huffman coding.

Description

The function ippsEncodeHuffInit_BZ2 is declared in the ippdc.h file. This function allocates memory and initializes the elements of the bzip2-specific internal state *pEncodeHuffState* for Huffman encoding. This structure is used by the function ippsEncodeHuff_BZ2.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.

ippStsSizeErr	Indicates an error if length of the source buffer is less than or equal to 0.

EncodeHuffFree_BZ2

Frees memory allocated for the bzip2-specific Huffman encoding structure.

Syntax

```
void ippsEncodeHuffFree_BZ2_16u8u(IppEncodeHuffState_BZ2* pEncodeHuffState);
```

Parameters

pEncodeHuffState	Pointer to the bzip2-specific Huffman coding structure.

Description

The function ippsEncodeHuffFree_BZ2 is declared in the ippdc.h file. This function frees memory allocated for bzip2-specific Huffman encoding structure by the function ippsEncodeHuffInitAlloc_BZ2.

PackHuffContext_BZ2

Performs the bzip2-specific encoding of Huffman context.

Syntax

```
IppStatus ippsPackHuffContext_BZ2_16u8u(Ipp32u* pCode, int* pCodeLenBits,
    Ipp8u* pDst, int* pDstLen, IppEncodeHuffState_BZ2* pEncodeHuffState);
```

Parameters

pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
pDst	Pointer to the destination vector.
pDstLen	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.

pEncodeHuffState	Pointer to internal state structure for bzip2 specific Huffman encoding.

Description

The function `ippsPackHuffContext_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific encoding of the `Huffman context`. The function uses the bzip2-specific Huffman encoding state structure *pEncodeHuffState*. This structure must be initialized by the functions `ippsEncodeHuffInitAlloc_BZ2` or `ippsEncodeHuffInit_BZ2` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the destination buffer is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

EncodeHuff_BZ2

Performs the bzip2-specific Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuff_BZ2_16u8u(Ipp32u* pCode, int* pCodeLenBits, Ipp16u**
    ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen, IppEncodeHuffState_BZ2*
    pEncodeHuffState);
```

Parameters

pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the length of source vector.
pDst	Pointer to the destination vector.
pDstLen	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.

pEncodeHuffState	Pointer to internal state structure for bzip2 specific Huffman encoding.

Description

The function `ippsEncodeHuff_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific Huffman encoding. The function uses the bzip2-specific Huffman encoding state structure *pEncodeHuffState*. This structure must be initialized by the functions `ippsEncodeHuffInitAlloc_BZ2` or `ippsEncodeHuffInit_BZ2` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the source or destination buffer is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

DecodeHuffGetSize_BZ2

Computes the size of the internal state for bzip2-specific Huffman decoding.

Syntax

```
IppStatus ippsDecodeHuffGetSize_BZ2_8u16u(int wndSize, int*
    pDecodeHuffStateSize);
```

Parameters

wndSize	Size of the block to be processed.
pDecodeHuffStateSize	Pointer to the size of the internal state for bzip2-specific Huffman coding.

Description

The function `ippsDecodeHuffGetSize_BZ2` is declared in the `ippdc.h` file. This function computes the size of the internal state structure for bzip2-specific Huffman decoding.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pDecodeHuffStateSize* is NULL.
ippStsSizeErr	Indicates an error if *wndSize* is less than or equal to 0.

DecodeHuffInit_BZ2

Initializes the elements of the bzip2-specific internal state for Huffman decoding.

Syntax

```
IppStatus ippsDecodeHuffInit_BZ2_8u16u(int sizeDictionary,
    IppDecodeHuffState_BZ2* pDecodeHuffState);
```

Parameters

sizeDictionary	Size of the dictionary.
pDecodeHuffState	Pointer to internal state structure for bzip2 specific Huffman coding.

Description

The function ippsDecodeHuffInit_BZ2 is declared in the ippdc.h file. This function initializes the elements of the bzip2-specific internal state for Huffman decoding. This structure is used by the function ippsDecodeHuff_BZ2. The size of this buffer should be computed previously by calling the function ippsDecodeHuffGetSize_BZ2.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pDecodeHuffState* pointer is NULL.
ippStsSizeErr	Indicates an error if *sizeDictionary* is less than or equal to 0.

DecodeHuffInitAlloc_BZ2

Allocated memory an initializes the elements of the bzip2-specific internal state for Huffman decoding.

Syntax

```
IppStatus ippsDecodeHuffInitAlloc_BZ2_8u16u(int wndSize, int sizeDictionary,
    IppDecodeHuffState_BZ2** ppDecodeHuffState);
```

Parameters

wndSize	Size in bytes of the block to be processed.
sizeDictionary	Size of the dictionary.
pDecodeHuffState	Pointer to the state structure for bzip2 specific Huffman decoding.

Description

The function `ippsDecodeHuffInitAlloc_BZ2` is declared in the `ippdc.h` file. This function allocates memory and initializes the elements of the bzip2-specific internal state *pDecodeHuffState* for Huffman decoding. This structure is used by the function ippsDecodeHuff_BZ2.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pDecodeHuffState* pointer is NULL.
ippStsSizeErr	Indicates an error if *sizeDictionary* or *wndSize* is less than or equal to 0.

DecodeHuffFree_BZ2

Frees memory allocated for the bzip2-specific Huffman decoding structure.

Syntax

```
void ippsDecodeHuffFree_BZ2_8u16u(IppDecodeHuffState_BZ2* pDecodeHuffState);
```

Parameters

pDecodeHuffState	Pointer to the bzip2-specific Huffman decoding structure.

Description

The function `ippsDecodeHuffFree_BZ2` is declared in the `ippdc.h` file. This function frees memory allocated for bzip2-specific Huffman decoding structure by the function `ippsDecodeHuffInitAlloc_BZ2`.

UnpackHuffContext_BZ2

Performs the bzip2-specific decoding of Huffman context.

Syntax

```
IppStatus IppStatus ippsUnpackHuffContext_BZ2_8u16u(Ipp32u* pCode, int*
    pCodeLenBits, Ipp8u** ppSrc, int* pSrcLen, IppDecodeHuffState_BZ2*
    pDecodeHuffState);
```

Parameters

pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the size of source buffer on input, pointer to the resulting length of the source vector on output.
pDecodeHuffState	Pointer to internal state structure for bzip2 specific Huffman decoding.

Description

The function `ippsUnpackHuffContext_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific decoding of the `Huffman context`. The function uses the bzip2-specific Huffman decoding state structure *pDecodeHuffState*. This structure must be initialized by the functions `ippsDecodeHuffInitAlloc_BZ2` or `ippsDecodeHuffInit_BZ2` beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the destination buffer is less than or equal to 0.
ippStsSrcSizeLessExpected	Indicates a warning if size of the source buffer is insufficient to store all output elements.

DecodeHuff_BZ2

Performs the bzip2-specific Huffman decoding.

Syntax

```
IppStatus IppStatus ippsDecodeHuff_BZ2_8u16u(Ipp32u* pCode, int* pCodeLenBits,
    Ipp8u** ppSrc, int* pSrcLen, Ipp16u* pDst, int* pDstLen,
    IppDecodeHuffState_BZ2* pDecodeHuffState);
```

Parameters

pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the size of source buffer.
pDst	Pointer to the destination vector.
pDstLen	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.
pDecodeHuffState	Pointer to internal state structure for bzip2 specific Huffman decoding.

Description

The function ippsDecodeHuff_BZ2 is declared in the ippdc.h file. This function performs the bzip2-specific Huffman decoding. The function uses the bzip2-specific Huffman decoding state structure *pDecodeHuffState*. This structure must be initialized by the functions ippsDecodeHuffInitAlloc_BZ2 or ippsDecodeHuffInit_BZ2 beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the destination buffer is less than or equal to 0.
`ippStsSrcSizeLessExpected`	Indicates a warning if size of the source buffer is insufficient to store all output elements.

Handling of Special Cases

A

Some mathematical functions implemented in Intel IPP are not defined for all possible argument values. This appendix describes how the corresponding Intel IPP functions used in signal processing data-domain handle situations when their input arguments fall outside the range of function definition or may lead to ambiguously determined output results.

Table A-1 below summarizes these special cases for general vector functions described in chapter 5 and lists result values together with status codes returned by these functions.

Table A-1 Special Cases for Intel IPP Signal Processing Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsSqrt	16s	Sqrt (x <0)	0	ippStsSqrtNegArg
	32f	Sqrt (x <0)	NAN_32F	ippStsSqrtNegArg
	64s	Sqrt (x <0)	0	ippStsSqrtNegArg
	64f	Sqrt (x <0)	NAN_64F	ippStsSqrtNegArg
ippsDiv ippsDiv_Round	8u	Div (0/0)	0	ippStsDivByZero
		Div (x/0)	IPP_MAX_8U	ippStsDivByZero
	16s	Div (0/0)	0	ippStsDivByZero
		Div (x/0), x>0	IPP_MAX_16S	ippStsDivByZero
		Div (x/0), x<0	IPP_MIN_16S	ippStsDivByZero
	16sc	Div (0/0)	0	ippStsDivByZero
		Div (x/0)	0	ippStsDivByZero

Table A-1 Special Cases for Intel IPP Signal Processing Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsDiv	32f	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0), x>0	INF_32F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_32F	ippStsDivByZero
	32fc	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0)	NAN_32F	ippStsDivByZero
				ippStsDivByZero
	64f	Div (0/0)	NAN_64F	
		Div (x/0), x>0	INF_64F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_64F	ippStsDivByZero
	64fc	Div (0/0)	NAN_64F	ippStsDivByZero
		Div (x/0)	NAN_64F	ippStsDivByZero
ippsDivC	All	Div(x/0)	x	ippStsDivByZeroErr
ippsLn	16s	Ln (0)	IPP_MIN_16S	ippStsLnZeroArg
		Ln (x<0)	IPP_MIN_16S	ippStsLnNegArg
	32s	Ln (0)	IPP_MIN_32S	ippStsLnZeroArg
		Ln (x<0)	IPP_MIN_32S	ippStsLnNegArg
	32f	Ln (x<0)	NAN_32F	ippStsLnNegArg
		Ln(x<IPP_MINABS_32F)	INF_NEG_32F	ippStsLnZeroArg
	64f	Ln (x<0)	NAN_64F	ippStsLnNegArg
		Ln(x<IPP_MINABS_64F)	INF_NEG_64F	ippStsLnZeroArg
ippsExp	16s	overflow	IPP_MAX_16S	ippStsNoErr
	32s	overflow	IPP_MAX_32S	ippStsNoErr
	64s	overflow	IPP_MAX_64S	ippStsNoErr
	32f	overflow	INF_32F	ippStsNoErr
	64f	overflow	INF_64F	ippStsNoErr
ippsThreshold, ippsThresholdLT, ippsThresholdGT, ippsThresholdLTVal, ippsThresholdGTVal	16sc	level<0	x	ippStsThreshNegLevelErr
	32fc	level<0	x	ippStsThreshNegLevelErr
	64fc	level<0	x	ippStsThreshNegLevelErr

Table A-1 Special Cases for Intel IPP Signal Processing Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsThresholdLTInv	32f	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_32F	ippStsInvZero
	32fc	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_32F	ippStsInvZero
	64f	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_64F	ippStsInvZero
	64fc	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_64F	ippStsInvZero
ipps10Log10	32s	10Log10 (0)	IPP_MIN_32S	ippStsLnZeroArg
		10Log10 (x<0)	IPP_MIN_32S	ippStsLnNegArg

Here x denotes an input value. For the definition of the constants used, see "Data Ranges" in Chapter 2.

Note that flavors of the same math function operating on different data types may produce unlike results for the equal argument values. However, for a given function and a fixed data type, handling of special cases is the same for all function flavors that have different descriptors in their names. For example, logarithm function `ippsLn` operating on `16s` data treats zero argument values in the same way for all its flavors `ippsLn_16s_Sfs` and `ippsLn_16s_ISfs`.

Table A-2 below summarizes special cases for fixed accuracy arithmetic vector functions described in the Chapter 12 "Fixed-Accuracy Arithmetic Functions".

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsInv	32f	Inv (x =+0)	INF_32F	ippStsSingularity
		Inv (x =-0)	-INF_32F	ippStsSingularity
	64f	Inv (x =+0)	INF_64F	ippStsSingularity
		Inv (x =-0)	-INF_64F	ippStsSingularity

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsDiv	32f	Div (x>0, y=+0)	INF_32F	ippStsSingularity
		Div (x>0, y=-0)	-INF_32F	ippStsSingularity
		DIv (x<0, y=+0)	-INF_32F	ippStsSingularity
		Div (x<0, y=-0)	INF_32F	ippStsSingularity
		Div (x=0, y=0)	NAN_32F	ippStsSingularity
	64f	Div (x>0, y=+0)	INF_64F	ippStsSingularity
		Div (x>0, y=-0)	-INF_64F	ippStsSingularity
		DIv (x<0, y=+0)	-INF_64F	ippStsSingularity
		Div (x<0, y=-0)	INF_64F	ippStsSingularity
		Div (x=0, y=0)	NAN_64F	ippStsSingularity
ippsSqrt	32f	Sqrt (x<0)	NAN_32F	ippStsDomain
		Sqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	Sqrt (x<0)	NAN_64F	ippStsDomain
		Sqrt (x=-INF)	NAN_64F	ippStsDomain
ippsInvSqrt	32f	InvSqrt (x<0)	NAN_32F	ippStsDomain
		InvSqrt (x=+0)	INF_32F	ippStsSingularity
		InvSqrt (x=-0)	-INF_32F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	InvSqrt (x<0)	NAN_64F	ippStsDomain
		InvSqrt (x=+0)	INF_64F	ippStsSingularity
		InvSqrt (x=-0)	-INF_64F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_64F	ippStsDomain
ippsInvCbrt	32f	InvCbrt (x=+0)	INF_32F	ippStsSingularity
		InvCbrt (x=-0)	-INF_32F	ippStsSingularity
	64f	InvCbrt (x=+0)	INF_64F	ippStsSingularity
		InvCbrt (x=-0)	-INF_64F	ippStsSingularity
ippsPow	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsPowx	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity
ippsExp	32f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_32F	ippStsOverflow
	64f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_64F	ippStsOverflow
ippsLn	32f	Ln (x<0)	NAN_32F	ippStsDomain
		Ln (x=-INF)	NAN_32F	ippStsDomain
		Ln (x=0)	-INF_32F	ippStsSingularity
	64f	Ln (x<0)	NAN_64F	ippStsDomain
		Ln (x=-INF)	NAN_64F	ippStsDomain
		Ln (x=0)	-INF_64F	ippStsSingularity
ippsLog10	32f	Ln (x<0)	NAN_32F	ippStsDomain
		Ln (x=-INF)	NAN_32F	ippStsDomain
		Ln (x=0)	-INF_32F	ippStsSingularity
	64f	Ln (x<0)	NAN_64F	ippStsDomain
		Ln (x=-INF)	NAN_64F	ippStsDomain
		Ln (x=0)	-INF_64F	ippStsSingularity
ippsCos	32f	Cos (INF)	NAN_32F	ippStsDomain
	64f	Cos (INF)	NAN_64F	ippStsDomain
ippsSin	32f	Sin (INF)	NAN_32F	ippStsDomain
	64f	Sin (INF)	NAN_64F	ippStsDomain
ippsSinCos	32f	SinCos (INF)	NAN_32F, NAN_32F	ippStsDomain
	64f	SinCos (INF)	NAN_64F, NAN_64F	ippStsDomain
ippsTan	32f	Tan (INF)	NAN_32F	ippStsDomain
	64f	Tan (INF)	NAN_64F	ippStsDomain

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsAcos	32f	Acos (x), |x|>1	NAN_32F	ippStsDomain
		Acos (INF)	NAN_32F	ippStsDomain
	64f	Acos (x), |x|>1	NAN_64F	ippStsDomain
		Acos (INF)	NAN_64F	ippStsDomain
ippsAsin	32f	Acos (x), |x|>1	NAN_32F	ippStsDomain
		Acos (INF)	NAN_32F	ippStsDomain
	64f	Acos (x), |x|>1	NAN_64F	ippStsDomain
		Acos (INF)	NAN_64F	ippStsDomain
ippsCosh	32f	Cosh (x), |x|>overflow	INF_32F	ippStsOverflow
	64f	Cosh (x), |x|>overflow	INF_64F	ippStsOverflow
ippsSinh	32f	Sinh (x), |x|>overflow	INF_32F	ippStsOverflow
	64f	Sinh (x), |x|>overflow	INF_64F	ippStsOverflow
ippsAcosh	32f	Acosh (x<1)	NAN_32F	ippStsDomain
		Acosh (x=-INF)	NAN_32F	ippStsDomain
	64f	Acosh (x<1)	NAN_64F	ippStsDomain
		Acosh (x=-INF)	NAN_64F	ippStsDomain
ippsAtanh	32f	Atanh (x=1)	INF_32F	ippStsSingularity
		Atanh (x=-1)	-INF_32F	ippStsSingularity
		Atanh (x), |x|>1	NAN_32F	ippStsDomain
		Atanh (INF)	NAN_32F	ippStsDomain
	64f	Atanh (x=1)	INF_64F	ippStsSingularity
		Atanh (x=-1)	-INF_64F	ippStsSingularity
		Atanh (x), |x|>1	NAN_64F	ippStsDomain
		Atanh (INF)	NAN_64F	ippStsDomain
ippsErfc	32f	Erfc (x), x<underflow	0	ippStsUnderflow
	64f	Exp (x), x<underflow	0	ippStsUnderflow

Removed Functions

B

This appendix contains Table B-1 that lists the functions that have been removed from the Intel IPP version 5.0. If an application created with the previous versions calls function listed here, then the source code must be modified. The table specifies the corresponding Intel IPP 5.0 functions to substitute for removed functions.

Table 0-1 Removed Functions

Removed from 5.0	Substituation or Workaround	Comments
Audio Coding Fucntions		
ippsApplySF_32f_I	ippsScale_32f_I	
ippsBuildHDT_32s	ippsVLCDecodeInit_32s	Data Compression domain
ippsBuildHET_16s	ippsVLCEncodeInit_32s	Data Compression domain
ippsBuildHET_VLC_32s	ippsVLCEncodeInit_32s	Data Compression domain
ippsCdbkFree_VQ_32f	ippsVQCodeBookFree_32f	
ippsCdbkInitAlloc_VQ_32f	ippsVQCodeBookInitAlloc_32f	
ippsCountBits_1tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsCountBits_2tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsCountBits_4tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsDecodeVLC_32s	ippsVLCDecodeOne_1u16s	Data Compression domain
ippsDecodeVLC_AACESCBlock_32s	ippsVLCDecodeEscBlock_AAC_1u16s	
ippsDecodeVLC_Block_32s	ippsVLCDecodeBlock_1u16s	Data Compression domain
ippsDecodeVLC_MP3ESCBlock_32s	ippsVLCDecodeEscBlock_MP3_ 1u16s	
ippsEncodeBlock_1tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeBlock_2tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeBlock_4tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain

Table 0-1 Removed Functions

Removed from 5.0	Substituation or Workaround	Comments
ippsEncodeVLC_Block_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeVLC_MP3ESCBlock_16s	ippsVLCEncodeEscBlock_MP3_16s1u	
ippsFDPInv_32f	ippsFDPInv_32f_I	
ippsGetSizeHDT_32s	ippsVLCDecodeGetSize_32s	Data Compression domain
ippsGetSizeHET_16s	ippsVLCEncodeGetSize_32s	Data Compression domain
ippsGetSizeHET_VLC_32s	ippsVLCEncodeGetSize_32s	Data Compression domain
ippsHuffmanCountBits_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsIndexSelect_VQ_32f	ippsVQIndexSelect_32f	
ippsMainSelect_VQ_32f	ippsVQMainSelect_32f	
ippsPreSelect_VQ_32f	ippsVQPreliminarySelect_32f	
ippsResetFDP_32f	ippsFDPReset_32f	
ippsResetFDP_SFB_32f	ippsFDPResetSfb_32f	
ippsResetFDPGroup_32f	ippsFDPResetGroup_32f	
ippsVectorReconstruction_VQ_32f	ippsVQReconstruction_32f	
Common Fucntions		
ippCoreGetCpuClocks	ippGetCpuClocks	
ippCoreGetCpuType	ippGetCpuType	
ippCoreGetStatusString	ippGetStatusString	
ippCoreSetDenormAreZeros	ippSetDenormAreZeros	
ippCoreSetFlushToZero	ippSetFlushToZero	
ippStaticFree		useless, no memory allocation
ippStaticInitBest	ippStaticInit	
Speech Recognition Fucntions		
ippsAddMulCol_64f_D2L	ippsAddMulColumn_64f_D2L	
ippsAddNRow_32f_D2	ippsAddNRows_32f_D2	
ippsCoefUpdateAECNLMS_32fc_I		
ippsCoefUpdateAECNLMS_32sc_I		
ippsControllerGetSizeAEC_32f		
ippsControllerGetSizeAEC_32s		
ippsControllerInitAEC_32f		
ippsControllerInitAEC_32s		
ippsControllerUpdateAEC_32f		
ippsControllerUpdateAEC_32s		

Table 0-1 Removed Functions

Removed from 5.0	Substituation or Workaround	Comments
ippsCopyCol_32f_D2	ippsCopyColumn_32f_D2	
ippsDeltaDeltaW1_32f_D2	ippsDeltaDelta_Win1_32f_D2	
ippsDeltaDeltaW2_32f_D2	ippsDeltaDelta_Win2_32f_D2	
ippsDeltaW1_32f_D2	ippsDelta_Win1_32f_D2	
ippsDeltaW2_32f_D2	ippsDelta_Win2_32f_D2	
ippsDotProdCol_64f_D2L	ippsDotProdColumn_64f_D2L	
ippsFilterAECNLMS_32fc		
ippsFilterAECNLMS_32sc_Sfs		
ippsLMThreshold_32f	ippsScaleLM_32f	
ippsLogGauss_16s32f_D2	ippsLogGauss_Scaled_16s32f_D2	
ippsLogGauss_16s32f_D2L	ippsLogGauss_Scaled_16s32f_D2L	
ippsLogGauss_IdVar_16s32f_D2	ippsLogGauss_IdVarScaled_ 16s32f_D2	
ippsLogGauss_IdVar_16s32f_D2L	ippsLogGauss_IdVarScaled_ 16s32f_D2L	
ippsLogGauss_IdVarLow_16s32f_D2	ippsLogGauss_IdVarLowScaled_ 16s32f_D2	
ippsLogGauss_IdVarLow_16s32f_D2L	ippsLogGauss_IdVarLowScaled_ 16s32f_D2L	
ippsLogGauss_Low_16s32f_D2	ippsLogGauss_LowScaled_ 16s32f_D2	
ippsLogGauss_Low_16s32f_D2L	ippsLogGauss_LowScaled_ 16s32f_D2L	
ippsLogGauss1_32f_D2	ippsLogGauss_32f_D2	
ippsLogGauss1_32f_D2L	ippsLogGauss_32f_D2L	
ippsLogGauss1_64f_D2	ippsLogGauss_64f_D2	
ippsLogGauss1_64f_D2L	ippsLogGauss_64f_D2L	
ippsLogGauss2_32f_D2	ippsLogGaussAddMultiMix_32f_D2	
ippsLogGauss2_32f_D2L	ippsLogGaussAddMultiMix_32f_D2L	
ippsLogGauss2_64f_D2	ippsLogGaussAddMultiMix_64f_D2	
ippsLogGauss2_64f_D2L	ippsLogGaussAddMultiMix_64f_D2L	
ippsLogGaussAdd_16s32f_D2	ippsLogGaussAdd_Scaled_16s32f_D2	
ippsLogGaussAdd_16s32f_D2L	ippsLogGaussAdd_Scaled_ 16s32f_D2L	

Table 0-1 Removed Functions

Removed from 5.0	Substituation or Workaround	Comments
ippsLogGaussAdd_IdVar_16s32f_D2	ippsLogGaussAdd_IdVarScaled_ 16s32f_D2	
ippsLogGaussAdd_IdVar_16s32f_D2L	ippsLogGaussAdd_IdVarScaled_ 16s32f_D2L	
ippsLogGaussAdd1_32f_D2	ippsLogGaussAdd_32f_D2	
ippsLogGaussAdd2_32f_D2	ippsLogGaussAddMultiMix_32f_D2	
ippsLogGaussAddMultiMix_ 16s32f_D2	ippsLogGaussAddMultiMix_Scaled_ 16s32f_D2	
ippsLogGaussAddMultiMix_1 6s32f_D2L	ippsLogGaussAddMultiMix_Scaled_ 16s32f_D2L	
ippsLogGaussMax_16s32f_D2	ippsLogGaussMax_Scaled_ 16s32f_D2	
ippsLogGaussMax_16s32f_D2L	ippsLogGaussMax_Scaled_ 16s32f_D2L	
ippsLogGaussMax_IdVar_16s32f_D2	ippsLogGaussMax_IdVarScaled_ 16s32f_D2	
ippsLogGaussMax_IdVar_16s32f_D2L	ippsLogGaussMax_IdVarScaled_ 16s32f_D2L	
ippsLogGaussMax1_32f_D2	ippsLogGaussMax_32f_D2	
ippsLogGaussMaxMultiMix_ 16s32f_D2	ippsLogGaussMaxMultiMix_Scaled_ 16s32f_D2	
ippsLogGaussMaxMultiMix_ 16s32f_D2L	ippsLogGaussMaxMultiMix_Scaled_ 16s32f_D2L	
ippsLogGaussMultiMix_16s32f_D2	ippsLogGaussMultiMix_Scaled_ 16s32f_D2	
ippsLogGaussMultiMix_16s32f_D2L	ippsLogGaussMultiMix_Scaled_ 16s32f_D2L	
ippsLogGaussMultiMix_Low_ 16s32f_D2	ippsLogGaussMultiMix_LowScaled_ 16s32f_D2	
ippsLogGaussMultiMix_Low_ 16s32f_D2L	ppsLogGaussMultiMix_LowScaled_16s 32f_D2L	
ippsLogGaussSingle_16s32f	ippsLogGaussSingle_Scaled_16s32f	
ippsLogGaussSingle_BlockDVar_ 16s32f	ippsLogGaussSingle_BlockDVarScaled _16s32f	
ippsLogGaussSingle_DirectVar_ 16s32f	ippsLogGaussSingle_DirectVarScaled_ 16s32f	
ippsLogGaussSingle_IdVar_16s32f	ippsLogGaussSingle_IdVarScaled _16s32f	
ippsLogGaussSingle_IdVarLow_ 16s32f	ippsLogGaussSingle_IdVarLowScaled_ 16s32f	
ippsLogGaussSingle_Low_16s32f	ippsLogGaussSingle_LowScaled _16s32f	

Table 0-1 Removed Functions

Removed from 5.0	Substituation or Workaround	Comments
ippsLPToLSP_16s_D2Sfs	ippsLPToLSP_16s_Sfs	
ippsLPToLSP_32f_D2	ippsLPToLSP_32f	
ippsLSPToLP_16s_D2Sfs	ippsLSPToLP_16s_Sfs	
ippsLSPToLP_32f_D2	ippsLSPToLP_32f	
ippsMahDist1_32f_D2	ippsMahDist_32f_D2	
ippsMahDist1_32f_D2L	ippsMahDist_32f_D2L	
ippsMahDist1_64f_D2	ippsMahDist_64f_D2	
ippsMahDist1_64f_D2L	ippsMahDist_64f_D2L	
ippsMahDist2_32f_D2	ippsMahDistMultiMix_32f_D2	
ippsMahDist2_32f_D2L	ippsMahDistMultiMix_32f_D2L	
ippsMahDist2_64f_D2	ippsMahDistMultiMix_64f_D2	
ippsMahDist2_64f_D2L	ippsMahDistMultiMix_64f_D2L	
ippsMeanCol_32f_D2	ippsMeanColumn_32f_D2	
ippsMeanCol_32f_D2L	ippsMeanColumn_32f_D2L	
ippsMeanVarCol_32f_D2	ippsVarColumn_32f_D2	
ippsMeanVarCol_32f_D2L	ippsVarColumn_32f_D2L	
ippsMulCol_64f_D2L	ippsMulColumn_64f_D2L	
ippsNormalizeCol_32f_D2	ippsNormalizeColumn_32f_D2	
ippsQRTransCol_64f_D2L	ippsQRTransColumn_64f_D2L	
ippsRecSqrt_32f_Th	ippsRecSqrt_32f	
ippsStepSizeUpdateAECNLMS_32f		
ippsStepSizeUpdateAECNLMS_32s		
ippsSumAllRow_32f_D2	ippsSumRow_32f_D2	
ippsSumAllRow_32f_D2L	ippsSumRow_32f_D2L	
ippsSumCol_32f_D2	ippsSumColumn_32f_D2	
ippsSumCol_32f_D2L	ippsSumColumn_32f_D2L	
ippsSumCol_64f_D2	ippsSumColumn_64f_D2	
ippsSumCol_64f_D2L	ippsSumColumn_64f_D2L	
ippsSumColAbs_64f_D2L	ippsSumColumnAbs_64f_D2L	
ippsSumColSqr_64f_D2L	ippsSumColumnSqr_64f_D2L	
ippsVarCol_32f_D2	ippsVarColumn_32f_D2	
ippsVarCol_32f_D2L	ippsVarColumn_32f_D2L	

Bibliography

This bibliography provides a list of reference books and other sources of additional information that might be useful to the application programmer. This list is neither complete nor exhaustive, but serves as a starting point. Of all the references listed, [Mit93] will be the most useful to those readers who already have a basic understanding of signal processing. This reference collects the work of 27 experts in the field and has both great breadth and depth.

The books [Opp75], [Opp89], [Jac89], and [Zie83] are undergraduate signal processing texts. [Opp89] is a much revised edition of the classic [Opp75]; [Jac89] is more concise than the others; and [Zie83] also covers continuous-time systems.

[AMRWB+] 3GPP Technical Specification 26.290: “Extended Adaptive Multi-Rate - Wideband (AMR-WB+) codec; Transcoding functions”, v.6.3.0, rel.6, June-2005.

[Arn97] Z. Arnavut, S. Magliveras. *Block-Sorting and Compression*. IEEE Data Compression Conference, Snowbird, Utah, pp.181-190, March 1997.

[Ash94] M.R. Asharif and F. Amano. *Acoustic Echo Canceler Using the FBAF Algorithm*. IEEE Trans. Comm., Vol. 42, No. 12, pp. 3090-3094, Dec. 1994.

[Brislawn94] C. Brislawn, *Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks*. Los Alamos Report LA-UR-94-1747, 1994.

[Cap78] V. Cappellini, A. G. Constantinides, and P. Emilani. *Digital Filters and Their Applications*. Academic Press, London, 1978.

[Cap94] Cappé O., *Elimination of the musical noise phenomenon with the Ephraim and Malah noise suppressor*. IEEE Trans. Speech and Audio Processing, vol. 2(2), (1994).

[CCITT] CCITT, Recommendation G.711. *Pulse Code Modulation of Frequencies*, 1984.

[Coh02] I. Cohen and B. Berdugo. *Noise Estimation by Minima Controlled Recursive Averaging for Robust Speech Enhancemen*t. IEEE Signal Proc. Letters, Vol. 9, No. 1, Jan. 2002, pp. 12-15.

[Cro83] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets*. Springer Verlag, Pennsylvania, 1992.

[Eph84] Y. Ephraim and D. Malah. *Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator*. IEEE Trans. ASSP, Vol. 32, No. 6, Dec. 1984, pp. 1109-1121.

[ETSI02] ETSI TS 102 114:2002 (DTS Coherent Acoustics – Core and Extensions)(2002). http://www.etsi.org/services_products/freestandard/home.htm

[EC126] ETSI TS 126 404 V6.0.0. UMTS. *General audio codec audio processing functions; Enhanced aacPlus general audio codec; Encoder specification*; SBR PART - 3GPP TS 26.404 version 6.0.0 Release 6 (09/2004).

[ES201] ETSI ES 201 108 V1.1.2. ETSI Standard*. Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Front-end feature extraction algorithm; Compression algorithms*.

[ES202] ETSI ES 202 050 V1.1.1. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Advanced front-end feature extraction algorithm; Compression algorithms*.

[Feig92] E. Feig and S. Winograd. *Fast algorithms for DCT*. IEEE Transactions on Signal Processing, vol.40, No.9, 1992.

[Flet82] J.G.Fletcher. *An Arithmetic Checksum for Serial Transmissions*. IEEE Transactions on Communications, vol.COM-30, No.1, 1982, pp.247-252.

[Gal75] R. Gallager, D.Van Voorhis. *Optimal source codes for geometrically distributed integer alphabets*. IEEE Transactions on Information Theory, vol. IT-21, No. 3, pp.228-230, 1975.

[Griff87] G.Griffiths, G.C.Stones. *The tea-leaf reader algorithm: an efficient implementation of CRC-16 and CRC-32*. Communications of the ACM , vol.30, No.7, 1987, pp.617-620.

[Ham83] R.W. Hamming. *Digital Filters*, Prentice-Hall, New Jersey 1983.

[Har78] F. Harris. *On the Use of Windows*. Proceedings of the IEEE, vol. 66, No.1, IEEE, 1978.

[Hay91] S. Haykin. *Adaptive Filter Theory*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[ISO11172] ISO/IEC 11172-3 - Information technology. *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s.* Part 3: Audio (1993)

[ISO13818] ISO/IEC 13818-3 - Information technology. *Generic coding of moving pictures and associated audio information.* Part 3: Audio (1998).

[ISO14496] ISO/IEC 14496-3 - Information technology. *Coding of audio-visual objects.* Part 3: Audio (2001).

[ISO14496A] ISO/IEC 14496-3/Amd.1:2003 - Bandwidth Extension (2003).

[ITU224] ITU-T Recommendation X.224 Annex D. *Checksum Algorithms*, (11/93), pp144,145.

[ITU722] ITU-T Recommendation G.722.1 (09/99), *Coding at 24 and 32 8 kbit/s for hand-free operation in systems with low frame loss*.

[ITU723] ITU-T Recommendation G.723.1 . *Dual Rate speech coder for Multimedia Communications transmitting at 5.3 and 6.3 Kbit/s*, (03/96).

[ITU723A] ITU-T Recommendation G.723.1 Annex A. *Silence compression scheme*, (11/96).

[ITU729] ITU-T Recommendation G.729. *Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)*, (03/96).

[ITU729A] ITU-T Recommendation G.729 Annex A. *Reduced Complexity 8 kbit/s CS-ACELP speech codec*, (11/96).

[ITU729B] ITU-T Recommendation G.729 Annex B. *A silence compression scheme for G.729 optimized for terminals conforming to Recommendation V.70*, (10/96).

[ITU729B1] ITU-T Recommendation G.729 Annex B Corrigendum 1, (02/98).

[ITUV34] ITU-T Recommendation V.34. *A modem operating at data signalling rates of up to 33 600 bit/s for use on the general switched telephone network and on leased point-to-point 2-wire telephone-type circuit.* (02/98).

[Jack89] Leland B. Jackson. *Digital Filters and Signal Processing*. Kluwer Academic Publishers, second edition, 1989.

[Kab86] P. Kabal and P.Ramachandran. *The Computation of line Spectral Frequencies Using Chebyshev Polynomials*. IEEE transaction on acoustic, speech and signal processing, vol. ASSP-34, No.6, 1986.

[Lyn89] Paul A. Lynn. *Introductory Digital Signal Processing with Computer Applications*. John Wiley&Sons, Inc., New York, 1993.

[Mar01] R. Martin. *Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics*. IEEE Trans. Speech and Audio, Vol. 9, No. 5, July 2001, pp. 504-512.

[Med91] Y. Medan, E. Yair, D. Chazan. *Super Resolution Pitch Determination of Speech Signals*. IEEE Transactions on Signal Processing, vol 39, No.1, 1991.

[Mit93] Sanjit K. Mitra and James F. Kaiser editors. *Handbook for Digital Signal Processing*. John Wiley&Sons, Inc., New York, 1993.

[Mit98] S. K. Mitra. *Digital Signal Processing*. McGraw Hill, 1998.

[Nel92] M.R.Nelson. *File verification using CRC 32-bit cyclical redundancy check*. Dr.Dobb's Journal, vol. 17, Issue 5, 1992, p.64.

[NIC91] Nam Ik Cho and Sang Uk Lee. *Fast algorithm and implementation of 2D DCT*. IEEE Transactions on Circuits and Systems, vol. 31, No.3, 1991.

[Opp75] A.V. Oppenheim and R.W. Schafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Opp89] A.V. Oppenheim and R.W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Rab78] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Rao90] K.R. Rao and P. Yip. *Discrete Cosine Transform. Algorithms, Advantages and Applications*. Academic Press, San Diego, 1990.

[RFC1950] P.Deutsch, J-L.Gailly. *ZLIB Compressed Data Format Specification version 3.3*, May, 1996. http://www.ietf.org/rfc/rfc1950.txt

[RFC1951] P.Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*, May, 1996. http://www.ietf.org/rfc/rfc1951.txt

[RFC1952] P.Deutsch. *GZIP file format specification version 4.3, May, 1996.* http://www.ietf.org/rfc/rfc1952.txt

[Seg78] R. Sedgewick. *Implementing quicksort programs*. Communications of the ACM, Vol. 21, No. 10, pp. 847-857, Oct. 1978.

[Storer82] J. Storer, T.Szymanski. *Data Compression via Textual Substitution.* Journal of the Association for Computing Machinery (ACM), vol.19, No.4, pp.928-951, Oct.1982.

[Strang96] G. Strang and T. Nguyen. *Wavelet and Filter Banks.* Wellesley-Cambridge Press, 1996, pp. 153-157.

[Tuc92] R. Tucker. *Voice Activity Detection Using a Periodicity Measure*. IEEE Proceedings-I, vol. 139, No. 4, August 1992, pp. 377-380.

[Vai93] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, Englewood Cliffs, New Jersey.

[Wid85] B. Widrow and S.D. Stearns. *Adaptive Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Zie83] Rodger E. Ziemer, William H. Tranter and D. Ronald Fannin. *Signals and Systems: Continuous and Discrete*. Macmillan Publishing Co., New York, 1983.

[Ziv77] J.Ziv and A.Lempel. *A Universal Algorithm for Sequential Data Compression*. IEEE Transactions on Information Theory, vol.23, No.3, pp.337-343, May 1977.

Glossary

adaptive filter	An adaptive filter varies its filter coefficients (taps) over time. Typically, the filter's coefficients are varied to make its output match a prototype "desired" signal as closely as possible. Non-adaptive filters do not vary their filter coefficients over time.
arithmetic operation	An operation that adds, subtracts, multiplies, or squares the image pixel values.
BQ	One of the modes, which indicates that the IIR initialization function initializes a cascade of biquads.
CCS	See complex conjugate-symmetric.
companding functions	The functions that perform an operation of data compression by using a logarithmic encoder-decoder. Companding allows you to maintain the percentage error constant by logarithmically spacing the quantization levels.
complex conjugate-symmetric	A kind of symmetry that arises in the Fourier transform of real signals. A complex conjugate-symmetric signal has the property that x(-n) = x(n)*, where "*" denotes conjugation.
conjugate	The conjugate of a complex number a + bj is a - bj.
conjugate-symmetric	See complex conjugate-symmetric.
DCT	Acronym for the discrete cosine transform.
decimation	Filtering a signal followed by down-sampling. Filtering prevents aliasing distortion in the subsequent down-sampling. See down-sampling.
down-sampling	Down-sampling conceptually decreases a signal's sampling rate by removing samples from between neighboring samples of a signal. See decimation.

element-wise	An element-wise operation performs the same operation on each element of a vector, or uses the elements of the same position in multiple vectors as inputs to the operation. For example, the element-wise addition of the vectors $\{x_0, x_1, x_2\}$ and $\{y_0, y_1, y_2\}$ is performed as follows: $\{x_0, x_1, x_2\} + \{y_0, y_1, y_2\} = \{x_0 + y_0, x_1 + y_1, x_2 + y_2\}$.
FIR	Abbreviation for finite impulse response filter. Finite impulse response filters do not vary their filter coefficients (taps) over time.
FIR LMS	Abbreviation for least mean squares finite impulse response filter.
fixed-point data format	A format that assigns one bit for a sign and all other bits for fractional part. This format is used for optimized conversion operations with signed, purely fractional vectors. For example, S.31 format assumes a sign bit and 31 fractional bits; S15.16 assumes a sign bit, 15 integer bits, and 16 fractional bits.
IIR	Abbreviation for infinite impulse response filters.
in-place	A function that performs its operation in-place, takes its input from an array and returns its output to the same array. See not-in-place.
interpolation	Up-sampling a signal followed by filtering. The filtering gives the inserted samples a value close to the samples of their neighboring samples in the original signal. See up-sampling.
LMS	Abbreviation for least mean square, an algorithm frequently used as a measure of the difference between two signals. Also used as shorthand for an adaptive FIR filter employing the LMS algorithm for adaptation.
LTI	Abbreviation for linear time-invariant systems. In LTI systems, if an input consists of the sum of a number of signals, then the output is the sum of the system's responses to each signal considered separately [Lyn89].

MMX™ technology	An enhancement to Intel architecture aimed at better performance in multimedia and communications applications. The technology uses four additional data types, eight 64-bit MMX registers, and 57 additional instructions implementing the SIMD (single instruction, multiple data) technique.
MR	One of the modes, indicating the multi-rate variety of the function.
multi-rate	An operation or signal processing system involving signals with multiple sample rates. Decimation and interpolation are examples of multi-rate operations.
not-in-place	A function that performs its operation not-in-place takes its input from a source array and puts its output in a second, destination array.
polyphase	A computationally efficient method for multi-rate filtering. For example, interpolation or decimation.
CCS	A representation of a complex conjugate-symmetric sequence which is easier to use than the Pack or Perm formats.
Pack	A compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms (“natural” in the sense that bit-reversed order is natural for radix-2 complex FFTs).
Perm	A format for storing the values for the FFT algorithm. RCPerm format stores the values in the order in which the FFT algorithm uses them. That is, the real and imaginary parts of a given sample need not be adjacent.
saturation	Using saturation arithmetic, when a number exceeds the data-range limit for its data type, it saturates to the upper data-range limit. For example, a signed word greater than 7FFFh saturates to 7FFFh. When a number is less than the lower data-range limit, it saturates to the lower data-range. For example, a signed word less than 8000h saturates to 8000h.
sinusoid	See tone.

Streaming SIMD Extensions	The major enhancement to Intel architecture instruction set. Incorporates a group of general-purpose floating-point instructions operating on packed data, additional packed integer instructions, together with cacheability control and state management instructions. These instructions significantly improve performance of applications using compute-intensive processing of floating-point and integer data.
tone	A sinusoid of a given frequency, phase, and magnitude. Tones are used as test signals and as building blocks for more complex signals.
up-sampling	Up-sampling conceptually increases the signal sampling rate by inserting zero-valued samples between neighboring samples of a signal.
window	A mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

Index

Numerics

10Log10, 5-50

A

AAC
- ADIF header structure, 10-126
- ADTS frame header structure, 10-127
- channel information structure, 10-133
- channel pair element structure, 10-132
- individual channel side information structure, 10-128
- LTP structure, 10-132
- scalable extension element structure, 10-129
- scalable main element structure, 10-129
- TNS structure for one layer, 10-131

about this manual, 1-5
about this software, 1-1
Abs, 5-41
AccCovarianceMatrix, 8-72
accuracy, of arithmetic functions, 12-1
ACELPFixedCodebookSearch_G723, 9-99
Acos, 12-38
Acosh, 12-52
adaptive codebook, 9-64
adaptive filters, 10-34
AdaptiveCodebookContribution_G729, 9-62
AdaptiveCodebookDecode_AMRWB, 9-204
AdaptiveCodebookDecode_AMRWBE, 9-229
AdaptiveCodebookDecode_GSMAMR, 9-148
AdaptiveCodebookDecodeGetSize_AMRWB, 9-202
AdaptiveCodebookDecodeInit_AMRWB, 9-202
AdaptiveCodebookDecodeUpdate_AMRWB, 9-203
AdaptiveCodebookGain_G729, 9-63
AdaptiveCodebookGain_GSMAMR, 9-150
AdaptiveCodebookGainCoeff_AMRWB, 9-197
AdaptiveCodebookSearch_AMRWB, 9-199
AdaptiveCodebookSearch_AMRWBE, 9-228
AdaptiveCodebookSearch_G723, 9-101
AdaptiveCodebookSearch_G729, 9-46
AdaptiveCodebookSearch_GSMAMR, 9-145
AdaptiveCodebookSearch_RTA, 9-342
Add, 5-18
AddAllRowSum, 8-8
AddC, 5-16
AddMulColumn, 8-172
AddMulRow, 8-173
AddNRows, 8-92
AddProduct, 5-20
AddProductC, 5-21
Adler32, 13-80
ALawToLin, 9-329
ALawToMuLaw, 9-332
AlgebraicCodebookDecode_AMRWB, 9-196
AlgebraicCodebookSearch_AMRWB, 9-191
AlgebraicCodebookSearch_GSMAMR, 9-152
AlignPtr(). See ippAlignPtr
AltInitMCRA, 8-243
amplitude ratio, 9-107
AMR WB speech codec
- AdaptiveCodebookDecode_AMRWB, 9-204
- AdaptiveCodebookDecodeGetSize_AMRWB, 9-202
- AdaptiveCodebookDecodeInit_AMRWB, 9-202
- AdaptiveCodebookDecodeUpdate_AMRWB, 9-203

AdaptiveCodebookGainCoeff_AMRWB, 9-197
AdaptiveCodebookSearch_AMRWB, 9-199
AlgebraicCodebookDecode_AMRWB, 9-196
AlgebraicCodebookSearch_AMRWB, 9-191
DecDTXBuffer_AMRWB, 9-215
DecodeGain_AMRWB, 9-212
Deemphasize_AMRWB, 9-186
EncDTXBuffer_AMRWB, 9-213
GainQuant_AMRWB, 9-209
HighPassFilter_AMRWB, 9-182
HighPassFilterGetDlyLine_AMRWB, 9-184
HighPassFilterGetSize_AMRWB, 9-181
HighPassFilterInit_AMRWB, 9-181
HighPassFilterSetDlyLine_AMRWB, 9-185
ISFQuant_AMRWB, 9-205
ISFQuantDecode_AMRWB, 9-207
ISFQuantDecodeDTX_AMRWB, 9-209
ISFQuantDTX_AMRWB, 9-208
ISFToISP_AMRWB, 9-177
ISPToISF_Norm_AMRWB, 9-176
ISPToLPC_AMRWB, 9-174
LPCToISP_AMRWB, 9-173
OpenLoopPitchSearch_AMRWB, 9-178
Preemphasize_AMRWB, 9-185
ResidualFilter_AMRWB, 9-172
SynthesisFilter_AMRWB, 9-187
VAD_AMRWB, 9-189
VADGetSize_AMRWB, 9-188
VADInit_AMRWB, 9-189
AMR WB+ speech codec
AdaptiveCodebookSearch_AMRWBE, 9-228
BandJoin_AMRWBE, 9-233
BandJoinUpsample_AMRWBE, 9-236
BandSplit_AMRWBE, 9-232
BandSplitDownsample_AMRWBE, 9-234
DecodeDemux_AMRWBE, 9-248
Deemphasize_AMRWBE, 9-223, 9-224
Downsample_AMRWBE, 9-230
EncodeMux_AMRWBE, 9-247
FFFTInv_PermToR_AMRWBE, 9-227, 9-229
FFTFwd_RToPerm_AMRWBE, 9-226
GainDecodeTCX_AMRWBE, 9-246
GainQuant_AMRWBE, 9-242
GainQuantTCX_AMRWBE, 9-244
ISFQuantDecode_AMRWBE, 9-239
ISFQuantDecodeHighBand_AMRWBE, 9-240
ISFQuantHighBand_AMRWBE, 9-241
LPCToISP_AMRWBE, 9-221
OpenLoopPitchSearch_AMRWBE, 9-220
PostFilterLowBand_AMRWBE, 9-225
QuantTCX_AMRWBE, 9-243
ResamplePolyphase_AMRWBE, 9-237
SNR_AMRWBE, 9-219
SynthesisFilter_AMRWBE, 9-222
Upsample_AMRWBE, 9-231
AnalysisFilter_PQMF_MP3, 10-84
AnalysisFilter_SBR, 10-203
AnalysisFilterEnc_SBR, 10-182
AnalysisFilterEncFree_SBR, 10-184
AnalysisFilterEncGetSize_SBR, 10-181
AnalysisFilterEncInit_SBR, 10-181
AnalysisFilterEncInitAlloc_SBR, 10-183
AnalysisFilterFree_PQMF_MP3, 10-83
AnalysisFilterFree_SBR, 10-190
AnalysisFilterGetSize_PQMF_MP3, 10-82
AnalysisFilterGetSize_SBR, 10-193
AnalysisFilterInit_PQMF_MP3, 10-81
AnalysisFilterInit_SBR, 10-198
AnalysisFilterInitAlloc_PQMF_MP3, 10-82
AnalysisFilterInitAlloc_SBR, 10-187
AnalysisPQMF_MP3, 10-80
And, 5-7
AndC, 5-6
Arctan, 5-53
Arithmetic functions, 5-15 thru ??
10Log10, 5-50
Abs, 5-41
Add, 5-18
AddC, 5-16
AddProduct, 5-20
AddProductC, 5-21
Arctan, 5-53
Cubrt, 5-46
Div, 5-36
Div_Round, 5-39
DivC, 5-33
DivCRev, 5-35
Exp, 5-47
Ln, 5-49
Mul, 5-24
MulC, 5-22
Sqr, 5-42

Sqrt, 5-44
Sub, 5-31
SubC, 5-27
SubCRev, 5-29
SumLn, 5-52
Asin, 12-40
Asinh, 12-54
Atan, 12-42
Atan2, 12-44
Atanh, 12-56
audience for this manual, 1-7
audio coding, 10-1
Audio coding functions
CalcSF, 10-20
Deinterleave, 10-11
FDPFree, 10-40
FDPFwd, 10-44
FDPGetSize, 10-40
FDPInit, 10-39
FDPInitAlloc, 10-38
FDPInv, 10-45
FDPReset, 10-41
FDPResetGroup, 10-43
FDPResetSfb, 10-42
FIRBlockFree, 10-35
FIRBlockInitAlloc, 10-34
FIRBlockOne, 10-36
IndexSelect_VQ, 10-68
Interleave, 10-9
MakeFloat, 10-24
MDCTFwd, MDCTInv, 10-31
MDCTFwdFree, MDCTInvFree, 10-28
MDCTFwdGetBufSize, MDCTInvGetBufSize, 10-30
MDCTFwdGetSize, MDCTInvGetSize, 10-29
MDCTFwdInit, MDCTInvInit, 10-27
MDCTFwdInitAlloc, MDCTInvInitAlloc, 10-25
Pow34, 10-13
Pow43, 10-16
Pow43Scale, 10-17
PreSelect_VQ, 10-64
Scale, 10-22
VLCCountEscBits_AAC, 10-56
VLCCountEscBits_MP3, 10-55
VLCDecodeEscBlock_AAC, 10-50
VLCDecodeEscBlock_MP3, 10-47
VLCDecodeUTupleEscBlock_AAC, 10-53
VLCDecodeUTupleEscBlock_MP3, 10-52
VLCEncodeEscBlock_AAC, 10-58
VLCEncodeEscBlock_MP3, 10-57
VQCodeBookFree, 10-63
VQCodeBookGetSize, 10-64
VQCodeBookInit, 10-62
VQCodeBookInitAlloc, 10-61
VQMainSelect, 10-66
VQReconstruction, 10-71
audio encoder
See MP3 audio encoder
Aurora functions
BlindEqualization_Aurora, 8-223
EvalDelta_Aurora, 8-224
HighBandCoding_Aurora, 8-221
LowHighFilter_Aurora, 8-220
MelFBankInitAlloc_Aurora, 8-216
NoiseSpectrumUpdate_Aurora, 8-213
ResidualFilter_Aurora, 8-217
SmoothedPowerSpectrumAurora, 8-212
TabsCalculation_Aurora, 8-217
VADDecision_Aurora, 8-230
VADFlush_Aurora, 8-231
VADGetBufSize_Aurora, 8-229
VADInit_Aurora, 8-229
WaveProcessing_Aurora, 8-218
WienerFilterDesign_Aurora, 8-214
AutoCorr, 6-6, 9-14
AutoCorr_G723, 9-90
AutoCorr_G729, 9-27
AutoCorr_GSMAMR, 9-122
AutoCorr_NormE, 9-17
AutoCorr_NormE_G723, 9-91
autocorrelation, in speech codecs, 9-27
AutoCorrLagMax, 9-15
AutoScale, 9-12

B

BandJoin_AMRWBE, 9-233
BandJoinUpsample_AMRWBE, 9-236
BandPassFilter_RTA, 9-347
BandSplit_AMRWBE, 9-232
BandSplitDownsample_AMRWBE, 9-234
bandwidth expansion, 9-94

Basic arithmetic functions for speech recognition, 8-8 thru 8-17
AddAllRowSum, 8-8
BlockDMatrixFree, 8-17
BlockDMatrixInitAlloc, 8-16
CopyColumn_Indirect, 8-14
MatVecMul, 8-20
SubRow, 8-13
SumColumn, 8-10
SumRow, 8-11
VecMatMul, 8-18
BhatDist, 8-161
binomial window, 9-90
bit stream, 10-20
BitReservoirInit_MP3, 10-107
BlindEqualization_Aurora, 8-223
block filtering, 10-34
BlockDMatrixFree, 8-17
BlockDMatrixInitAlloc, 8-16
BuildSignTable, 8-130
BuildSymblTableDV4D, 5-104
Burrows-Wheeler transform, 13-85 thru 13-93
BWTFwd, 13-88
BWTFwd_SmallBlock, 13-91
BWTFwdGetSize, 13-87
BWTGetSize_SmallBlock, 13-91
BWTInv, 13-89
BWTInv_SmallBlock, 13-92
BWTInvGetSize, 13-89
BWT. See Burrows-Wheeler transform
BWTFwd, 13-88
BWTFwd_SmallBlock, 13-91
BWTFwdGetSize, 13-87
BWTGetSize_SmallBlock, 13-91
BWTInv, 13-89
BWTInv_SmallBlock, 13-92
BWTInvGetSize, 13-89
bzip2 Coding Functions, 13-114
bzip2 compatibility functions
CRC32_BZ2, 13-122
bzip2 compatible functions
DecodeHuffFree_BZ2, 13-130
EncodeHuffFree_BZ2, 13-126
RLEFree_BZ2, 13-114

C

CalcSF, 10-20
CalcStatesDV, 5-103
CartToPolar, complex, 5-94
Cbrt, 12-12
CCS format, 7-6 thru ??
CdbkFree, 8-191
CdbkGetSize, 8-187
CdbkInit, 8-188
CdbkInitAlloc, 8-190
Ceil, 12-65
CepstrumToLP, 8-32
Chebyshev polynomials, 9-31
codebook, 9-33
fixed, 9-58
indices, 9-37
search, 9-62
CodebookSearch_G728, 9-287
CombinedFilter_G728, 9-274
CombinedFilterGetStateSize_G728, 9-273
CombinedFilterInit_G728, 9-274
common functions
ippAlignPtr, 3-12
ippFree, 3-15
ippGetCpuClocks, 3-9
ippGetCpuFreqMhz, 3-9
ippGetCpuType, 3-7
ippGetMaxCacheSizeB, 3-10
ippGetNumThreads, 3-14
ippGetStatusString, 3-6
ippMalloc, 3-14
ippSetDenormAreZeros, 3-11
ippSetFlushToZero, 3-10
ippSetNumThreads, 3-13
Companding functions, 9-327 thru 9-333
ALawToLin, 9-329
bALawToMuLaw, 9-332
LinToALaw, 9-330
LinToMuLaw, 9-328
MuLawToALaw, 9-331
MuLawToLin, 9-327
Compare, 11-10
CompareIgnoreCase, 11-13
CompareIgnoreCaseLatin, 11-13
CompensateOffset, 8-23
Concat, 11-22

ConcatC, 11-23
concepts
    IPP structures, 2-7
    of IPP, 2-1
Conj, 5-67
ConjCcs, 7-11
ConjFlip, 5-68
ConjPack, 7-7
ConjPerm, 7-9
Conv, 6-2
ConvCyclic, 6-5
conventions
    font, 1-7
    naming, 1-8
    signal name, 1-8
Conversion Functions
    FindNearestOne, 5-100
Conversion functions, 5-55 thru 5-99
    CartToPolar, complex, 5-94
    Conj, 5-67
    ConjFlip, 5-68
    Convert, 5-60
    CplxToReal, 5-78
    FindNearest, 5-101
    Flip, 5-99
    Imag, 5-76
    Join, 5-63
    JoinScaled, 5-65
    Magnitude, 5-69
    MagSquared, 5-71
    MaxOrder, 5-97
    PolarToCart, complex, 5-96
    Preemphasize, 5-98
    Real, 5-75
    RealToCplx, 5-77
    SortAscend, 5-56
    SortDescend, 5-56
    SortIndexAscend, 5-57
    SortIndexDescend, 5-57
    SplitScaled, 5-66
    SwapBytes, 5-58
    Threshold, 5-79
    Threshold_GT, 5-82
    Threshold_GTAbs, 5-85
    Threshold_GTVal, 5-88
    Threshold_LT, 5-82
    Threshold_LTAbs, 5-85
    Threshold_LTInv, 5-92
    Threshold_LTVal, 5-88
    Threshold_LTValGTVal, 5-88
Convert, 5-60
Convolution and correlation functions, 6-1 thru 6-10
    Conv, 6-2
    ConvCyclic, 6-5
    CrossCorr, 6-8
    UpdateLinear, 6-10
    UpdatePower, 6-12
ConvPartial, 9-6
Copy, 4-3
CopyColumn, 8-74
CopyColumn_Indirect, 8-14
CopyWithPadding, 8-45
Cos, 12-30
Cosh, 12-46
CountInRange, 5-142
CplxToReal, 5-78
CRC32, 13-80
CRC32_BZ2, 13-122
Cross-Architecture Alignment, 1-2
CrossCorr, 6-8, 9-18
CrossCorrCoeff, 8-88
CrossCorrCoeffDecim, 8-87
CrossCorrCoeffInterpolation, 8-90
CrossCorrLagMax, 9-19
cross-platform applications, 2-1
Cubrt, 5-46

D

Data compression functions
    Adler32, 13-80
    BWTFwd, 13-88
    BWTFwd_SmallBlock, 13-91
    BWTFwdGetSize, 13-87
    BWTGetSize_SmallBlock, 13-91
    BWTInv, 13-89
    BWTInv_SmallBlock, 13-92
    BWTInvGetSize, 13-89
    CRC32, 13-80
    CRC32_BZ2, 13-122
    DecodeGIT, 13-102
    DecodeGITGetSize, 13-102

DecodeGITInit, 13-101
DecodeGITInitAlloc, 13-100
DecodeHuff, 13-35
DecodeHuffFree_BZ2, 13-130
DecodeHuffInit, 13-33
DecodeHuffInitAlloc, 13-32
DecodeHuffOne, 13-34
DecodeLZ77, 13-71
DecodeLZ77DynamicHuff, 13-74
DecodeLZ77FixedHuff, 13-73
DecodeLZ77GetPairs, 13-76
DecodeLZ77GetSize, 13-69
DecodeLZ77GetStatus, 13-78
DecodeLZ77Init, 13-69
DecodeLZ77InitAlloc, 13-70
DecodeLZ77InitGetBlockType, 13-72
DecodeLZ77Reset, 13-79
DecodeLZ77SelectHuffMode, 13-62
DecodeLZ77SetPairs, 13-77
DecodeLZ77SetStatus, 13-78
DecodeLZ77StoredBlock, 13-75
DecodeLZSS, 13-48
DecodeLZSSInit, 13-47
DecodeLZSSInitAlloc, 13-47
DecodeRLE, 13-111
EncodeGIT, 13-97
EncodeGITGetSize, 13-96
EncodeGITInit, 13-95
EncodeGITInitAlloc, 13-94
EncodeHuff, 13-30
EncodeHuffFinal, 13-31
EncodeHuffFree_BZ2, 13-126
EncodeHuffInit, 13-28
EncodeHuffInitAlloc, 13-26
EncodeHuffOne, 13-29
EncodeLZ77, 13-59
EncodeLZ77FixedHuff, 13-61
EncodeLZ77Flush, 13-64
EncodeLZ77GetPairs, 13-65
EncodeLZ77GetSize, 13-57
EncodeLZ77GetStatus, 13-67
EncodeLZ77Init, 13-56
EncodeLZ77InitAlloc, 13-57
EncodeLZ77Reset, 13-68
EncodeLZ77SelectHuffMode, 13-60
EncodeLZ77SetPairs, 13-66
EncodeLZ77SetStatus, 13-67
EncodeLZ77StoredBlock, 13-63
EncodeLZSS, 13-43
EncodeLZSSFlush, 13-44
EncodeLZSSInit, 13-42
EncodeLZSSInitAlloc, 13-41
EncodeRLE, 13-110
EncodeRLEInitAlloc, 13-114
GITFree, 13-95
HuffFree, 13-27
HuffGetDstBuffSize, 13-36
HuffGetLenCodeTable, 13-32
HuffGetSize, 13-29
HuffLenCodeTablePack, 13-37
HuffLenCodeTableUnpack, 13-38
LZ77Free, 13-58
LZSSFree, 13-41
LZSSGetSize, 13-42
MTFFwd, 13-108
MTFGetSize, 13-108
MTFInit, 13-107
MTFInitAlloc, 13-106, 13-107
MTFInv, 13-109
RLEFree_BZ2, 13-114
VLCCountBits, 13-10
VLCDecodeBlock, 13-15
VLCDecodeFree, 13-12
VLCDecodeGetSize, 13-14
VLCDecodeInit, 13-12
VLCDecodeInitAlloc, 13-11
VLCDecodeOne, 13-16
VLCDecodeUTupleBlock, 13-22
VLCDecodeUTupleFree, 13-19
VLCDecodeUTupleGetSize, 13-21
VLCDecodeUTupleInit, 13-20
VLCDecodeUTupleInitAlloc, 13-18
VLCDecodeUTupleOne, 13-23
VLCEncodeBlock, 13-8
VLCEncodeFree, 13-6
VLCEncodeGetSize, 13-7
VLCEncodeInit, 13-6
VLCEncodeInitAlloc, 13-5
VLCEncodeOne, 13-9
DcsClustLAccumulate, 8-169
DcsClustLCompute, 8-170
DCT functions, 7-65 thru 7-78

DCTFwd, 7-74
DCTFwdFree, 7-67
DCTFwdGetBufSize, 7-68
DCTFwdGetSize, 7-72
DCTFwdInitAlloc, 7-65
DCTIFwdInit, 7-70
DCTInv, 7-76
DCTInvFree, 7-67
DCTInvGetBufSize, 7-69
DCTInvGetSize, 7-73
DCTInvInit, 7-71
DCTInvInitAlloc, 7-66
DCTFwd, 7-74
DCTFwd_G722, 9-257
DCTFwdFree, 7-67
DCTFwdGetBufSize, 7-68
DCTFwdGetSize, 7-72
DCTFwdInit, 7-70
DCTFwdInitAlloc, 7-65
DCTInv, 7-76
DCTInv_G722, 9-257
DCTInvFree, 7-67
DCTInvGetBufSize, 7-68, 7-69
DCTInvGetSize, 7-73
DCTInvInit, 7-71
DCTInvInitAlloc, 7-65, 7-66
DCTLifter, 8-63
DCTLifterFree, 8-63
DCTLifterGetSize_MulC0, 8-59
DCTLifterInit_MulC0, 8-60
DCTLifterInitAlloc, 8-61
DecDTXBuffer_AMRWB, 9-215
DecDTXBuffer_GSMAMR, 9-165
Decode_G726, 9-266
DecodeAdaptiveVector_G723, 9-114
DecodeAdaptiveVector_G729, 9-49
DecodeChanPairElt_AAC, 10-139
DecodeDatStrElt_AAC, 10-144
DecodeDemux_AMRWBE, 9-248
DecodeExtensionHeader_AAC, 10-161
DecodeFillElt_AAC, 10-145
DecodeGain_AMRWB, 9-212
DecodeGain_G729, 9-57
DecodeGetStateSize_G726, 9-265
DecodeGIT, 13-102
DecodeGITGetSize, 13-102
DecodeGITInit, 13-101
DecodeGITInitAlloc, 13-100
DecodeHuffFree_BZ2, 13-130
DecodeHuffInit, 13-33
DecodeHuffInitAlloc, 13-32
DecodeHuffOne, 13-34
DecodeInit_G726, 9-265
DecodeIsStereo_AAC, 10-150
DecodeLZ77, 13-71
DecodeLZ77DynamicHuff, 13-74
DecodeLZ77FixedHuff, 13-73
DecodeLZ77GetBlockType, 13-72
DecodeLZ77GetPairs, 13-76
DecodeLZ77GetSize, 13-69
DecodeLZ77GetStatus, 13-78
DecodeLZ77Init, 13-69
DecodeLZ77InitAlloc, 13-70
DecodeLZ77Reset, 13-79
DecodeLZ77SetStatus, 13-78
DecodeLZ77StoredBlock, 13-75
DecodeLZSS, 13-48
DecodeLZSSInit, 13-47
DecodeLZSSInitAlloc, 13-47
DecodeMainHeader_AAC, 10-160
DecodeMsPNS_AAC, 10-164
DecodeMsStereo_AAC, 10-148
DecodePNS_AAC, 10-162
DecodePrgCfgElt_AAC, 10-138
DecodeRLE, 13-111
DecodeTNS_AAC, 10-153
DecomposeDCTToMLT, 9-260
DecomposeMLTToDCT, 9-258
Deemphasize_AMRWB, 9-186
Deemphasize_AMRWBE, 9-223
Deemphasize_GSMFR, 9-250
DEFLATE, 13-50
Deinterleave, 10-11
DeinterleaveSpectrum_AAC, 10-151
Delta, 8-77
DeltaDelta, 8-82
Derivative functions, 8-74 thru 8-85
AccCovarianceMatrix, 8-72
CopyColumn, 8-74
Delta, 8-77
DeltaDelta, 8-82
EvalDelta, 8-75

DetectTransient_SBR, 10-177
DFT for given frequency (Goertzel) functions, 7-61 thru 7-64
  Goertz, 7-61
  GoertzTwo, 7-63
DFT functions, 7-41 thru 7-60
  DFTFree_C, 7-44
  DFTFree_R, 7-44
  DFTFwd_CToC, 7-46
  DFTFwd_RToCCS, 7-50
  DFTFwd_RToPack, 7-50
  DFTFwd_RToPerm, 7-50
  DFTGetBufSize_C, 7-45
  DFTGetBufSize_R, 7-45
  DFTInitAlloc_C, 7-42
  DFTInitAlloc_R, 7-42
  DFTInv_CCSToR, 7-53
  DFTInv_CToC, 7-48
  DFTInv_PackToR, 7-53
  DFTInv_PermToR, 7-53
  DFTOutOrdFree_C, 7-57
  DFTOutOrdFwd_CToC, 7-58
  DFTOutOrdGetBufSize_C, 7-57
  DFTOutOrdInitAlloc_C, 7-56
  DFTOutOrdInv_CToC, 7-59
DFTInv_CToC, 7-48
dispatcher functions
  ippStaticInit, 3-16
  ippStaticInitCpu, 3-17
Distance coding, 13-93
Div, 5-33, 5-36, 12-7
Div_Round, 5-39
DivCRev, 5-35
DotProd, 5-138
DotProd_G729, 9-24
DotProdAutoScale, 9-13
DotProdColumn, 8-175
Downsample_AMRWBE, 9-230
DTW, 8-137
Durbin, 8-27

E

Echo Canceller
  FIRSubband_EC, 9-315
  FIRSubbandCoeffUpdate_EC, 9-316
  FIRSubbandLow_EC, 9-315
  FIRSubbandLowCoeffUpdate_EC, 9-316
  SubbandProcessGetSize, 9-291
EMNS functions
  AltInitMCRA, 8-243
  FilterUpdateEMNS, 8-237
  FilterUpdateWiener, 8-239
  GetSizeMCRA, 8-241
  InitMCRA, 8-242
  UpdateNoisePSDMCRA, 8-244
EmptyFBankInitAlloc, 8-53
EncDTXBuffer_AMRWB, 9-213
EncDTXBuffer_GSMAMR, 9-165
EncDTXHandler_GSMAMR, 9-164
EncDTXSID_GSMAMR, 9-162
Encode_G726, 9-264
EncodeGetStateSize_G726, 9-262
EncodeGIT, 13-97
EncodeGITGetSize, 13-96
EncodeGITInit, 13-95
EncodeGITInitAlloc, 13-94
EncodeHuff, 13-30
EncodeHuffFinal, 13-31
EncodeHuffFree_BZ2, 13-126
EncodeHuffInit, 13-28
EncodeHuffInitAlloc, 13-26
EncodeHuffOne, 13-29
EncodeInit_G726, 9-263
EncodeLZ77, 13-59
EncodeLZ77DynamicHuff, 13-62
EncodeLZ77FixedHuff, 13-61
EncodeLZ77Flush, 13-64
EncodeLZ77GetPairs, 13-65, 13-77
EncodeLZ77GetSize, 13-57
EncodeLZ77GetStatus, 13-67
EncodeLZ77Init, 13-56
EncodeLZ77InitAlloc, 13-57
EncodeLZ77Reset, 13-68
EncodeLZ77SelectHuffMode, 13-60
EncodeLZ77SetPairs, 13-66
EncodeLZ77SetStatus, 13-67
EncodeLZ77StoredBlock, 13-63
EncodeLZSS, 13-43
EncodeLZSSFlush, 13-44
EncodeLZSSInit, 13-42
EncodeLZSSInitAlloc, 13-41

EncodeMux_AMRWBE, 9-247
EncodeRLE, 13-110
EncodeRLEInitAlloc, 13-114
EncodeTNS_AAC, 10-170
energy, average, 9-61
Entropy, 8-158
Equal, 11-14
Erf, 12-58
Erfc, 12-60
ErfInv, 12-61
error reporting, 2-13
EstimateTNR_SBR, 10-179
EvalDelta, 8-75
EvalDelta_Aurora, 8-224
EvalFBank, 8-58
excitation, 9-48
    random, 9-87
Exp, 5-47, 12-24
ExpNegSqr, 8-160

F

FBankFree, 8-54
FBankGetCenters, 8-54
FBankGetCoeffs, 8-56
FBankSetCenters, 8-55
FBankSetCoeffs, 8-57
FDPFree, 10-40
FDPFwd, 10-44
FDPGetSize, 10-40
FDPInit, 10-39
FDPInitAlloc, 10-38
FDPInv, 10-45
FDPReset, 10-41
FDPResetGroup, 10-43
FDPResetSfb, 10-42
Feature processing functions, 8-21 thru 8-91
    CepstrumToLP, 8-32
    CompensateOffset, 8-23
    CopyWithPadding, 8-45
    DCTLifter, 8-63
    DCTLifterFree, 8-63
    DCTLifterGetSize_MulC0, 8-59
    DCTLifterInit_MulC0, 8-60
    DCTLifterInitAlloc, 8-61
    Durbin, 8-27
    EmptyFBankInitAlloc, 8-53
    EvalFBank, 8-58
    FBankFree, 8-54
    FBankGetCenters, 8-54
    FBankGetCoeffs, 8-56
    FBankSetCenters, 8-55
    FBankSetCoeffs, 8-57
    LinearPrediction, 8-25
    LinearToMel, 8-44
    LPToCepstrum, 8-31
    LPToLSP, 8-40
    LPToReflection, 8-33
    LPToSpectrum, 8-30
    MelFBankGetSize, 8-46
    MelFBankInit, 8-47
    MelFBankInitAlloc, 8-48
    MelLinFBankInitAlloc, 8-50
    MelToLinear, 8-43
    PitchmarkToF0, 8-38
    ReflectionToAR, 8-35
    ReflectionToLP, 8-34
    ReflectionToTilt, 8-36
    Schur, 8-29
    SignChangeRate, 8-24
    UnitCurve, 8-39
    ZeroMean, 8-22
FFT functions, 7-19 thru 7-41
    FFTFree_C, 7-22
    FFTFree_R, 7-22
    FFTFwd_CToC, 7-29
    FFTFwd_RToCCS, 7-34
    FFTFwd_RToPack, 7-34
    FFTFwd_RToPerm, 7-34
    FFTGetBufSize_C, 7-27
    FFTGetBufSize_R, 7-27
    FFTGetSize_C, 7-25
    FFTGetSize_R, 7-25
    FFTInit_C, 7-23
    FFTInit_R, 7-23
    FFTInitAlloc_C, 7-20
    FFTInitAlloc_R, 7-20
    FFTInv_CCSToR, 7-38
    FFTInv_CToC, 7-31
    FFTInv_PackToR, 7-38
    FFTInv_PermToR, 7-38
FFTFwd_RToPerm_AMRWBE, 9-226

FFTFwd_RToPerm_GSMAMR, 9-121
FFTInv_PermToR_AMRWBE, 9-227
FillShortlist_Column, 8-134
FillShortlist_Row, 8-132
filter
    inverse, 9-72
    long-term, 9-70
    short-term, 9-72
FilteredExcitation_G729, 9-87
Filtering functions, 6-14 thru 6-132
    SumWindow, 6-17
FilterMedian, 6-131
FilterUpdateEMNS, 8-237
FilterUpdateWiener, 8-239
Find, FindRev, 11-3
FindC, FindRevC, 11-5
FindCAny, FindRevCAny, 11-6
FindNearest, 5-101
FindNearestOne, 5-100
FindPeaks, 8-246
FIR, 6-41
FIR filter functions, 6-18 thru 6-70
    FIR, 6-41
    FIR_Direct, 6-50
    FIRFree, 6-24
    FIRGenBandpass, 6-67
    FIRGenBandstop, 6-69
    FIRGenHighpass, 6-65
    FIRGenLowpass, 6-63
    FIRGetDlyLine, 6-36
    FIRGetStateSize, 6-30
    FIRGetTaps, 6-33
    FIRInit, 6-25
    FIRInitAlloc, 6-19
    FIRMR_Direct, 6-54
    FIRMRGetStateSize, 6-30
    FIRMRInit, 6-27
    FIRMRInitAlloc, 6-21
    FIROne, 6-39
    FIROne_Direct, 6-47
    FIRSetDlyLine, 6-38
    FIRSetTaps, 6-34
    FIRSparse, 6-60
    FIRSparseGetStateSize, 6-59
    FIRSparseInit, 6-58
FIR LMS filter functions, 6-70 thru 6-92
FIR_Direct, 6-50
FIRBlockFree, 10-35
FIRBlockInitAlloc, 10-34
FIRBlockOne, 10-36
FIRFree, 6-24
FIRGenBandpass, 6-67
FIRGenBandstop, 6-69
FIRGenHighpass, 6-65
FIRGenLowpass, 6-63
FIRGenMidBand_AMRWBE, 9-224
FIRGetDlyLine, 6-36
FIRGetStateSize, 6-30
FIRGetTaps, 6-33
FIRInit, 6-25
FIRInitAlloc, 6-19
FIRLMS, 6-76
FIRLMSFree, 6-72
FIRLMSGetDlyLine, 6-74
FIRLMSGetTaps, 6-73
FIRLMSInitAlloc, 6-71
FIRLMSMRFree, 6-82
FIRLMSMRGetDlyVal, 6-89
FIRLMSMRGetTapsPointer, 6-86
FIRLMSMRInitAlloc, 6-81
FIRLMSMROne, 6-90
FIRLMSMROneVal, 6-92
FIRLMSMRPutVal, 6-90
FIRLMSMRSetMu, 6-83
FIRLMSMRUpdateTaps, 6-83
FIRLMSOne_Direct, 6-78
FIRLMSSetDlyLine, 6-75
FIRMR_Direct, 6-54
FIRMRGetStateSize, 6-30
FIRMRInitAlloc, 6-21
FIRMTInit, 6-27
FIROne, 6-39
FIROne_Direct, 6-47
FIRSetDlyLine, 6-38
FIRSetTaps, 6-34
FIRSparse, 6-60
FIRSparseGetStateSize, 6-59
FIRSparseInit, 6-58
FIRSubband_EC, 9-315
FIRSubbandCoeffUpdate_EC, 9-316
FIRSubbandLow_EC, 9-315
FIRSubbandLowCoeffUpdate_EC, 9-316

Fixed filter banks wavelet transforms, 7-85 thru 7-90
fixed-accuracy arithmetic functions
exponential and logarithmic
Exp, 12-24
Ln, 12-26
Log10, 12-28
hyperbolic
Acosh, 12-52
Asinh, 12-54
Atanh, 12-56
Cosh, 12-46
Sinh, 12-48
Tanh, 12-50
power and root
Cbrt, 12-12
Div, 12-7
Hypot, 12-21
Inv, 12-5
InvCbrt, 12-14
InvSqrt, 12-11
Pow, 12-16
Powx, 12-19
Sqrt, 12-9
rounding
Ceil, 12-65
Floor, 12-64
Modf, 12-71
NearbyInt, 12-68
Rint, 12-70
Round, 12-67
Trunc, 12-66
special
Erf, 12-58
Erfc, 12-60
ErfInv, 12-61
trigonometric
Acos, 12-38
Asin, 12-40
Atan, 12-42
Atan2, 12-44
Cos, 12-30
Sin, 12-32
SinCos, 12-34
Tan, 12-36
FixedCodebookDecode_GSMAMR, 9-155
FixedCodebookSearch_G729, 9-50
FixedCodebookSearch_RTA, 9-343
FixedCodebookSearchRandom_RTA, 9-343
Flag and hint arguments, 7-4 thru 7-5
Flip, 5-99
Floor, 12-64
font conventions, 1-7
FormVector, 8-184
FormVectorVQ, 8-198
fraction delay, 9-47
frequency domain prediction, 10-38
function descriptions, 1-6
function naming, 2-2

G

G.722.1 speech codec
DCTFwd_G722, DCTInv_G722, 9-257
DecomposeDCTToMLT, 9-260
DecomposeMLTToDCT, 9-258
HuffmanEncode_G722, 9-261
G.723.1 speech codec
codebook search functions
ACELPFixedCodebookSearch_G723, 9-99
AdaptiveCodebookSearch_G723, 9-101
MPMLQFixedCodebookSearch_G723, 9-102
OpenLoopPitchSearch_G723, 9-98
ToeplizMatrix_G723, 9-104
filter functions
DecodeAdaptiveVector_G723, 9-114
HarmonicNoiseSubtract_G723, 9-113
HarmonicSearch_G723, 9-112
HighPassFilter_G723, 9-108
IIR16s_G723, 9-109
PitchPostFilter_G723, 9-116
SynthesisFilter_G723, 9-110
TiltCompensation_G723, 9-111
gain quantization functions
GainControl_G723, 9-107
GainQuant_G723, 9-105
LP analysis functions
AutoCorr_G723, 9-90
AutoCorr_NormE_G723, 9-91
LevinsonDurbin_G723, 9-92
LPCToLSF_G723, 9-93
LSFDecode_G723, 9-96
LSFQuant_G723, 9-97
LSFToLPC_G723, 9-95
G.726 speech codec
Decode_G726, 9-266

DecodeGetStateSize_G726, 9-265
DecodeInit_G726, 9-265
Encode_G726, 9-264
EncodeGetStateSize_G726, 9-262
EncodeInit_G726, 9-263
G.728 speech codec
CodebookSearch_G728, 9-287
CombinedFilter_G728, 9-274
CombinedFilterGetStateSize_G728, 9-273
CombinedFilterInit_G728, 9-274
IIR_G728, 9-270
IIR_Init_G728, 9-269
IIRGetStateSize_G728, 9-269
ImpulseResponseEnergy_G728, 9-288
LevinsonDurbin_G728, 9-286
LPCInverseFilter_G728, 9-279
PitchPeriodExtraction_G728, 9-280
PostFilter_G728, 9-277
PostFilterAdapterGetStateSize_G728, 9-278
PostFilterAdapterStateInit_G728, 9-279
PostFilterGetStateSize_G728, 9-276
PostFilterInit_G728, 9-276
SyntesisFilter_G728, 9-272
SynthesisFilterGetStateSize_G728, 9-271
SynthesisFilterInit_G728, 9-272
WinHybrid_G728, 9-284
WinHybridGetStateSize_G728, 9-282
WinHybridInit_G728, 9-283
G.729 codec functions, 9-22
G.729 speech codec
basic functions
DotProd_G729, 9-24
Interpolate_G729, 9-25
codebook search functions
AdaptiveCodebookContribution_G729, 9-62
AdaptiveCodebookGain_G729, 9-63
AdaptiveCodebookSearch_G729, 9-46
DecodeAdaptiveVector_G729, 9-49
FixedCodebookSearch_G729, 9-50
LagWindow_G729, 9-41
OpenLoopPitchSearch_G729, 9-43
ToeplizMatrix_G729, 9-55
filter functions
FilteredExcitation_G729, 9-87
HarmonicFilter, 9-75
HighPassFilter_G729, 9-78
HighPassFilterInit_G729, 9-77
HighPassFilterSize_G729, 9-77
IIR16s_G729, 9-79
LongTermPostFilter_G729, 9-68
PhaseDispersion_G729D, 9-82
PhaseDispersionGetStateSize_G729D, 9-80
PhaseDispersionInit_G729D, 9-81
PhaseDispersionUpdate_G729D, 9-81
Preemphasize_G729, 9-83
RandomNoiseExcitation_G729B, 9-86
ResidualFilter_G729, 9-65
ShortTermPostFilter_G729, 9-72
SynthesisFilter_G729, 9-67
TiltCompensation_G729, 9-73
WinHybrid_G729E, 9-85
WinHybridGetStateSize_G729E, 9-84
WinHybridInit_G729E, 9-84
gain quantization functions
DecodeGain_G729, 9-57
GainControl_G729, 9-58
GainQuant_G729, 9-60
LP analysis functions
AutoCorr_G729, 9-27
LevinsonDurbin_G729, 9-28
LPCToLSP_G729, 9-30
LSFDecode_G729, 9-34
LSFDecodeErased_G729, 9-35
LSFQuant_G729, 9-32
LSFToLSP_G729, 9-32
LSPQuant_G729, 9-37
LSPToLPC_G729, 9-36
LSPToLSF_G729, 9-40
gain
estimation, 9-106
of the adaptive-codebook vector, 9-64
optimum, 9-61
GainControl_G723, 9-107
GainControl_G729, 9-58
GainDecodeTCX_AMRWBE, 9-246
GainQuant_AMRWB, 9-209
GainQuant_AMRWBE, 9-242
GainQuant_G723, 9-105
GainQuant_G729, 9-60
GainQuantTCX_AMRWBE, 9-244
Gaussian distribution functions
RandGauss, 4-39
RandGauss_Direct, 4-40
RandGaussFree, 4-36

RandGaussGetSize, 4-37
RandGaussInit, 4-38
RandGaussInitAlloc, 4-35
GaussianDist, 8-154
GaussianMerge, 8-157
GaussianSplit, 8-156
generalized interval transformation (GIT) coding, 13-93 thru ??
Generalized Interval Transformation coding, 13-93
GetCdbkSize, 8-192
GetCodebook, 8-193
GetCpuClocks(). See ippGetCpuClocks
GetCpuFreqMhz(). See ippGetCpuFreqMhz
GetCpuType(). See ippGetCpuType
GetLibVersion, 3-2
GetMaxCacheSizeB(). See ippGetMaxCacheSizeB
GetNumThreads(). See ippGetNumThreads
GetSizeMCRA, 8-241
GetStatusString(). See ippGetStatusString
GetVarPointDV, 5-102
GIT coding.See generalized interval transformation coding
GITFree, 13-95
Goertz, 7-61
GoertzTwo, 7-63
GSM FR speech codec
Deemphasize_GSMFR, 9-250
HighPassFilter_GSMFR, 9-253
Preemphasize_GSMFR, 9-256
RPEQuantDecode_GSMFR, 9-249
Schur_GSMFR, 9-254
ShortTermAnalysisFilter_GSMFR, 9-251
ShortTermSynthesisFilter_GSMFR, 9-252
WeightingFilter_GSMFR, 9-255
GSM-AMR speech codec, 9-118
adaptive codebook functions, 9-134
AdaptiveCodebookDecode_GSMAMR, 9-148
AdaptiveCodebookGain_GSMAMR, 9-150
AdaptiveCodebookSearch_GSMAMR, 9-145
AlgebraicCodebookSearch_GSMAMR, 9-152
AutoCorr_GSMAMR, 9-122
data structures, 9-4
DecDTXBuffer_GSMAMR, 9-165
discontinuous transmission (DTX) primitives, 9-156
EncDTXBuffer_GSMAMR, 9-165
EncDTXHandler_GSMAMR, 9-164
EncDTXSID_GSMAMR, 9-162
FFTFwd_RToPerm_GSMAMR, 9-121
fixed codebook primitives, 9-152
FixedCodebookDecode_GSMAMR, 9-155
ImpulseResponseTarget_GSMAMR, 9-144
Interpolate_GSMAMR, 9-120
LevinsonDurbin_GSMAMR, 9-124
LP analysis and quantization functions, 9-122
LPCToLSP_GSMAMR, 9-126
LSFToLSP_GSMAMR, 9-129
LSPQuant_GSMAMR, 9-129
LSPToLPC_GSMAMR, 9-127
open-loop pitch search, 9-134
OpenLoopPitchSearchDTXVAD1_GSMAMR, 9-138
OpenLoopPitchSearchDTXVAD2_GSMAMR, 9-141
OpenLoopPitchSearchNonDTX_GSMAMR, 9-136
PostFilter_GSMAMR, 9-167
Preemphasize_GSMAMR, 9-156
QuantLSPDecode_GSMAMR, 9-132
VAD1_GSMAMR, 9-157
VAD2_GSMAMR, 9-160
GZIP, 13-50

H

Hamming window, 9-90
hardware and software requirements, 1-2
HarmonicFilter, 9-75
HarmonicNoiseSubtract_G723, 9-113
HarmonicSearch_G723, 9-112
Hash, 11-21
HighBandCoding_Aurora, 8-221
HighPassFilter_AMRWB, 9-182
HighPassFilter_G723, 9-108
HighPassFilter_G729, 9-78
HighPassFilter_GSMFR, 9-253
HighPassFilter_RTA, 9-346
HighPassFilterGetDlyLine_AMRWB, 9-184
HighPassFilterGetSize_AMRWB, 9-181
HighPassFilterInit_AMRWB, 9-181
HighPassFilterInit_G729, 9-77
HighPassFilterSetDlyLine_AMRWB, 9-185
HighPassFilterSize_G729, 9-77
Hilbert, 7-81

Hilbert transform functions, 7-79 thru 7-82
Hilbert, 7-81
HilbertFree, 7-80
HilbertInitAlloc, 7-79
HilbertFree, 7-80
HilbertInitAlloc, 7-79
HuffFree, 13-27
HuffGetLenCodeTable, 13-32
HuffGetSize, 13-29
HuffLenCodeTablePack, 13-37
HuffLenCodeTableUnpack, 13-38
Huffman coding, ?? thru 13-38
DecodeHuff, 13-35
DecodeHuffInit, 13-33
DecodeHuffInitAlloc, 13-32
DecodeHuffOne, 13-34
EncodeHuff, 13-30
EncodeHuffFinal, 13-31
EncodeHuffInit, 13-28
EncodeHuffInitAlloc, 13-26
EncodeHuffOne, 13-29
HuffFree, 13-27
HuffGetDstBuffSize, 13-36
HuffGetLenCodeTable, 13-32
HuffGetSize, 13-29
HuffLenCodeTablePack, 13-37
HuffLenCodeTableUnpack, 13-38
HuffmanDecode_MP3, 10-113
HuffmanEncode_G722, 9-261
HuffmanEncode_MP3, 10-102
Hypot, 12-21

I

IIR, 6-113
IIR filter functions, 6-93 thru 6-128
IIR, 6-113
IIR_BiQuadDirect, 6-123
IIR_Direct, 6-122
IIRFree, 6-99
IIRGetDlyLine, 6-109
IIRGetStateSize, 6-105
IIRGetStateSize_BiQuad, 6-106
IIRInit, 6-100
IIRInit_BiQuad, 6-102
IIRInitAlloc, 6-95
IIRInitAlloc_BiQuad, 6-97
IIROne, 6-112
IIROne_BiQuadDirect, 6-121
IIROne_Direct, 6-120
IIRSetDlyLine, 6-110
IIRSetTaps, 6-107
IIRSparse, 6-127
IIRSparseGetStateSize, 6-126
IIRSparseInit, 6-124
IIR_BiQuadDirect, 6-123
IIR_Direct, 6-122
IIR_G728, 9-270
IIR_Init_G728, 9-269
IIR16s_G723, 9-109
IIR16s_G729, 9-79
IIRFree, 6-99
IIRGenHighpass, 6-130
IIRGetDlyLine, 6-109
IIRGetStateSize_BiQuad, 6-106
IIRGetStateSize_G728, 9-269
IIRInit, 6-100
IIRInit_BiQuad, 6-102
IIRInitAlloc, 6-95
IIRInitAlloc_BiQuad, 6-97
IIROne, 6-112
IIROne_BiQuadDirect, 6-121
IIROne_Direct, 6-120
IIRSetDlyLine, 6-110
IIRSetTaps, 6-107
IIRSparse, 6-127
IIRSparseGetStateSize, 6-126
IIRSparseInit, 6-124
Imag, 5-76
ImpulseResponseEnergy_G728, 9-288
ImpulseResponseTarget_GSMAMR, 9-144
IndexSelect_VQ, 10-68
InitMCRA, 8-242
Insert, 11-8
Intel Performance Library Suite, 2-1
Intel Performance Primitives software, 1-1
Interleave, 10-9
interleaved to multi-row format conversion, 10-9
Interpolate_G729, 9-25
Interpolate_GSMAMR, 9-120
InterpolateC_NR, 9-7
Inv, 12-5

InvCbrt, 12-14
Inversion frequencies, 13-93
InvInit, 7-71
InvSqrt, 9-14, 12-11
ippAlignPtr, 3-12
ippCoreGetCpuType, 3-7
ippFree, 3-15
ippGetCpuClocks, 3-9
ippGetCpuFreqMhz, 3-9
ippGetGetMaxCacheSizeB, 3-10
ippGetNumThreads, 3-14
ippGetStatusString, 3-6
ippMalloc, 3-14
ipps10Log10(). See 10Log10
ippsAbs(). See Abs
ippsAccCovarianceMatrix(). See AccCovarianceMatrix
ippsACELPFixedCodebookSearch_G723(). See ACELPFixedCodebookSearch_G723
ippsAcos, 12-38
ippsAcosh, 12-52
ippsAdaptiveCodebookContribution_G729(). See AdaptiveCodebookContribution_G729
ippsAdaptiveCodebookGain_G729(). See AdaptiveCodebookGain_G729
ippsAdaptiveCodebookSearch_G723(). See AdaptiveCodebookSearch_G723
ippsAdaptiveCodebookSearch_G729(). See AdaptiveCodebookSearch_G729
ippsAdaptiveCodebookSearch_RTA(). See AdaptiveCodebookSearch_RTA
ippsAdd(). See Add
ippsAddAllRowSum(). See AddAllRowSum
ippsAddC(). See AddC
ippsAddMulColumn(). See AddMulColumn
ippsAddMulRow(). See AddMulRow
ippsAddNRows(). See AddNRows
ippsAddProduct(). See AddProduct
ippsAddProductC(). See AddProductC
ippsAdler32(). See Adler32
ippsALawToLin(). See ALawToLin
ippsALawToMuLaw(). See ALawToMuLaw
ippsAltInitMCRA(). See AltInitMCRA
ippsAnalysisFilter_PQMF_MP3(). See AnalysisFilter_PQMF_MP3
ippsAnalysisFilter_SBR(). See AnalysisFilter_SBR
ippsAnalysisFilterEnc_SBR(). See AnalysisFilterEnc_SBR
ippsAnalysisFilterEncFree_SBR(). See AnalysisFilterEncFree_SBR
ippsAnalysisFilterEncGetSize_SBR(). See AnalysisFilterEncGetSize_SBR
ippsAnalysisFilterEncInit_SBR(). See AnalysisFilterEncInit_SBR
ippsAnalysisFilterEncInitAlloc_SBR(). See AnalysisFilterEncInitAlloc_SBR
ippsAnalysisFilterFree_PQMF_MP3(). See AnalysisFilterFree_PQMF_MP3
ippsAnalysisFilterFree_SBR(). See AnalysisFilterFree_SBR
ippsAnalysisFilterGetSize_PQMF_MP3(). See AnalysisFilterGetSize_PQMF_MP3
ippsAnalysisFilterGetSize_SBR(). See AnalysisFilterGetSize_SBR
ippsAnalysisFilterInit_PQMF_MP3(). See AnalysisFilterInit_PQMF_MP3
ippsAnalysisFilterInit_SBR(). See AnalysisFilterInit_SBR
ippsAnalysisFilterInitAlloc_PQMF_MP3(). See AnalysisFilterInitAlloc_PQMF_MP3
ippsAnalysisFilterInitAlloc_SBR(). See AnalysisFilterInitAlloc_SBR
ippsAnalysisPQMF_MP3(). See AnalysisPQMF_MP3
ippsAnalysisPQMF_MP3_16s32s, 10-80
ippsAnd(). See And
ippsAndC(). See AndC
ippsArctan(). See Arctan
ippsAsin, 12-40
ippsAsinh, 12-54
ippsAtan, 12-42
ippsAtan2, 12-44
ippsAtanh, 12-56
ippsAutoCorr(). See AutoCorr
ippsAutoCorr_G723(). See AutoCorr_G723
ippsAutoCorr_G729(). See AutoCorr_G729
ippsAutoCorr_NormE(). See AutoCorr_NormE
ippsAutoCorr_NormE_G723(). See AutoCorr_NormE_G723
ippsAutoCorrLagMax(). See AutoCorrLagMax
ippsAutoScale(). See AutoScale
ippsBandPassFilter_RTA(). See BandPassFilter_RTA
ippsBhatDist(). See BhatDist
ippsBitReservoirInit_MP3(). See BitReservoirInit_MP3

ippsBlindEqualization_Aurora(). See BlindEqualization_Aurora
ippsBlockDMatrixFree(). See BlockDMatrixFree
ippsBlockDMatrixInitAlloc(). See BlockDMatrixInitAlloc
ippsBuildSignTable(). See BuildSignTable
ippsBuildSymblTableDV4D(). See BuildSymblTableDV4D
ippsBWTFwd(). See BWTFwd
ippsBWTFwd_SmallBlock(). See BWTFwd_SmallBlock
ippsBWTFwdGetSize(). See BWTFwdGetSize
ippsBWTGetSize_SmallBlock(). See BWTGetSize_SmallBlock
ippsBWTInv(). See BWTInv
ippsBWTInv_SmallBlock(). See BWTInv_SmallBlock
ippsBWTInvGetSize(). See BWTInvGetSize
ippsCalcSF(). See CalcSF
ippsCalcStatesDV(). See CalcStatesDV
ippsCartToPolar(). See CartToPolar, complex
ippsCbrt, 12-12
ippsCdbkFree(). See CdbkFree
ippsCdbkGetSize(). See CdbkGetSize
ippsCdbkInit(). See CdbkInit
ippsCdbkInitAlloc(). See CdbkInitAlloc
ippsCeil, 12-65
ippsCepstrumToLP(). See CepstrumToLP
ippsCompare(). See Compare
ippsCompareIgnoreCase(). See CompareIgnoreCase
ippsCompensateOffset(). See CompensateOffset
ippsConcat(). See Concat
ippsConcatC(). See ConcatC
ippsConj(). See Conj
ippsConjCcs(). See ConjCcs
ippsConjFlip(). See ConjFlip
ippsConjPack(). See ConjPack
ippsConv(). See Conv
ippsConvCyclic(). See ConvCyclic
ippsConvert(). See Convert
ippsConvPartial(). See ConvPartial
ippsCopy(). See Copy
ippsCopyColumn(). See CopyColumn
ippsCopyColumn_Indirect(). See CopyColumn_Indirect
ippsCopyWithPadding(). See CopyWithPadding
ippsCos, 12-30
ippsCosh, 12-46
ippsCountInRange(). See CountInRange
ippsCplxToReal(). See CplxToReal
ippsCRC32(). See CRC32
ippsCRC32_BZ2(). See CRC32_BZ2
ippsCrossCorr(). See CrossCorr
ippsCrossCorrCoeff(). See CrossCorrCoeff
ippsCrossCorrCoeffDecim(). See CrossCorrCoeffDecim
ippsCrossCorrCoeffInterpolation(). See CrossCorrCoeffInterpolation
ippsCubrt(). See Cubrt
ippsDcsClustLAccumulate(). See DcsClustLAccumulate
ippsDcsClustLCompute(). See DcsClustLCompute
ippsDCTFwd(). See DCTFwd
ippsDCTFwdFree(). See DCTFwdFree
ippsDCTFwdGetBufSize(). See DCTFwdGetBufSize
ippsDCTFwdGetSize(). See DCTFwdGetSize
ippsDCTFwdInit(). See DCTFwdInit
ippsDCTFwdInitAlloc(). See DCTFwdInitAlloc
ippsDCTInv(). See DCTInv
ippsDCTInvFree(). See DCTInvFree
ippsDCTInvGetBufSize(). See DCTInvGetBufSize
ippsDCTInvGetSize(). See DCTInvGetSize
ippsDCTInvInit(). See DCTInvInit
ippsDCTInvInitAlloc(). See DCTInvInitAlloc
ippsDCTLifter(). See DCTLifter
ippsDCTLifterFree(). See DCTLifterFree
ippsDCTLifterGetSize_MulC0(). See DCTLifterGetSize_MulC0
ippsDCTLifterInit_MulC0(). See DCTLifterInit_MulC0
ippsDCTLifterInitAlloc(). See DCTLifterInitAlloc
ippsDecodeAdaptiveVector_G723(). See DecodeAdaptiveVector_G723
ippsDecodeAdaptiveVector_G729(). See DecodeAdaptiveVector_G729
ippsDecodeChanPairElt_AAC, 10-139
ippsDecodeChanPairElt_AAC(). See DecodeChanPairElt_AAC
ippsDecodeDatStrElt_AAC, 10-144
ippsDecodeDatStrElt_AAC(). See DecodeDatStrElt_AAC
ippsDecodeExtensionHeader_AAC, 10-161
ippsDecodeExtensionHeader_AAC(). See DecodeExtensionHeader_AAC

ippsDecodeFillElt_AAC, 10-145
ippsDecodeFillElt_AAC(). See DecodeFillElt_AAC
ippsDecodeGain_G729(). See DecodeGain_G729
ippsDecodeGIT(). See DecodeGIT
ippsDecodeGITGetSize(). See DecodeGITGetSize
ippsDecodeGITInit(). See DecodeGITInit
ippsDecodeGITInitAlloc(). See DecodeGITInitAlloc
ippsDecodeHuff(). See ippsDecodeHuff
ippsDecodeHuffFree_BZ2(). See DecodeHuffFree_BZ2
ippsDecodeHuffInit(). See ippsDecodeHuffInit
ippsDecodeHuffInitAlloc(). See ippsDecodeHuffInitAlloc
ippsDecodeHuffOne(). See ippsDecodeHuffOne
ippsDecodeIsStereo_AAC(). See DecodeIsStereo_AAC
ippsDecodeIsStereo_AAC_32s, 10-150
ippsDecodeLZ77(). See DecodeLZ77
ippsDecodeLZ77DynamicHuff(). See DecodeLZ77DynamicHuff
ippsDecodeLZ77FixedHuff(). See DecodeLZ77FixedHuff
ippsDecodeLZ77GetBlockType(). See DecodeLZ77GetBlockType
ippsDecodeLZ77GetPairs(). See DecodeLZ77GetPairs
ippsDecodeLZ77GetSize(). See DecodeLZ77GetSize
ippsDecodeLZ77GetStatus(). See DecodeLZ77GetStatus
ippsDecodeLZ77Init(). See DecodeLZ77Init
ippsDecodeLZ77InitAlloc(). See DecodeLZ77InitAlloc
ippsDecodeLZ77Reset(). See DecodeLZ77Reset
ippsDecodeLZ77SetPairs(). See DecodeLZ77SetPairs
ippsDecodeLZ77SetStatus(). See DecodeLZ77SetStatus
ippsDecodeLZ77StoredBlock(). See DecodeLZ77StoredBlock
ippsDecodeLZSS(). See DecodeLZSS
ippsDecodeLZSSInit(). See DecodeLZSSInit
ippsDecodeLZSSInitAlloc(). See DecodeLZSSInitAlloc
ippsDecodeMainHeader_AAC, 10-160
ippsDecodeMainHeader_AAC(). See DecodeMainHeader_AAC
ippsDecodeMsPNS_AAC(). See DecodeMsPNS_AAC
ippsDecodeMsStereo_AAC(). See DecodeMsStereo_AAC
ippsDecodeMsStereo_AAC_32s_I, 10-148
ippsDecodePNS_AAC(). See DecodePNS_AAC
ippsDecodePNS_AAC_32s, 10-162, 10-164
ippsDecodePrgCfgElt_AAC, 10-138
ippsDecodePrgCfgElt_AAC(). See DecodePrgCfgElt_AAC
ippsDecodeRLE(). See DecodeRLE
ippsDecodeTNS_AAC(). See DecodeTNS_AAC
ippsDecodeTNS_AAC_32s_I, 10-153
ippsDeinterleave(). See Deinterleave
ippsDeinterleaveSpectrum_AAC(). See DeinterleaveSpectrum_AAC
ippsDeinterleaveSpectrum_AAC_32s, 10-151
ippsDelta(). See Delta
ippsDeltaDelta(). See DeltaDelta
ippsDetectTransient_SBR(). See DetectTransient_SBR
ippsDFTFree_C(). See DFTFree_C
ippsDFTFree_R(). See DFTFree_R
ippsDFTFwd_CToC(). See DFTFwd_CToC
ippsDFTFwd_RToCCS(). See DFTFwd_RToCCS
ippsDFTFwd_RToPack(). See DFTFwd_RToPack
ippsDFTFwd_RToPerm(). See DFTFwd_RToPerm
ippsDFTGetBufSize_R(). See DFTGetBufSize_R
ippsDFTInitAlloc_C(). See DFTInitAlloc_C
ippsDFTInitAlloc_R(). See DFTInitAlloc_R
ippsDFTInv_CCSToR(). See DFTInv_CCSToR
ippsDFTInv_CToC(). See DFTInv_CToC
ippsDFTInv_PackToR(). See DFTInv_PackToR
ippsDFTInv_PermToR(). See DFTInv_PermToR
ippsDFTOutOrdFree_C(). See DFTOutOrdFree_C
ippsDFTOutOrdFwd_CToC(). See DFTOutOrdFwd_CToC
ippsDFTOutOrdGetBufSize_C(). See DFTOutOrdGetBufSize_C
ippsDFTOutOrdInitAlloc_C(). See DFTOutOrdInitAlloc_C
ippsDFTOutOrdInv_CToC(). See DFTOutOrdInv_CToC
ippsDiv, 12-7
ippsDiv(). See Div
ippsDiv_Round(). See Div_Round
ippsDivC(). See DivC
ippsDivCRev(). See DivCRev
ippsDotProd (). See DotProd

ippsDotProd_G729(). See DotProd_G729
ippsDotProdAutoScale(). See DotProdAutoScale
ippsDotProdColumn(). See DotProdColumn
ippsDTW(). See DTW
ippsDurbin(). See Durbin
ippsEmptyFBankInitAlloc(). See EmptyFBankInitAlloc
ippsEncodeGIT. See EncodeGIT
ippsEncodeGITGetSize(). See EncodeGITGetSize
ippsEncodeGITInit(). See EncodeGITInit
ippsEncodeGITInitAlloc(). See EncodeGITInitAlloc
ippsEncodeHuff(). See EncodeHuff
ippsEncodeHuffFinal(). See EncodeHuffFinal
ippsEncodeHuffFree_BZ2(). See EncodeHuffFree_BZ2
ippsEncodeHuffInit(). See EncodeHuffInit
ippsEncodeHuffInitAlloc(). See EncodeHuffInitAlloc
ippsEncodeHuffOne(). See EncodeHuffOne
ippsEncodeLZ77(). See EncodeLZ77
ippsEncodeLZ77DynamicHuff(). See EncodeLZ77DynamicHuff
ippsEncodeLZ77FixedHuff(). See EncodeLZ77FixedHuff
ippsEncodeLZ77Flush(). See EncodeLZ77Flush
ippsEncodeLZ77GetPairs(). See EncodeLZ77GetPairs
ippsEncodeLZ77GetSize(). See EncodeLZ77GetSize
ippsEncodeLZ77GetStatus(). See EncodeLZ77GetStatus
ippsEncodeLZ77Init(). See EncodeLZ77Init
ippsEncodeLZ77InitAlloc(). See EncodeLZ77InitAlloc
ippsEncodeLZ77Reset(). See EncodeLZ77Reset
ippsEncodeLZ77SelectHuffMode(). See EncodeLZ77SelectHuffMode
ippsEncodeLZ77SetPairs(). See EncodeLZ77SetPairs
ippsEncodeLZ77SetStatus(). See EncodeLZ77SetStatus
ippsEncodeLZ77StoredBlock(). See EncodeLZ77StoredBlock
ippsEncodeLZSS(). See EncodeLZSS
ippsEncodeLZSSFlush(). See EncodeLZSSFlush
ippsEncodeLZSSInit(). See EncodeLZSSInit
ippsEncodeLZSSInitAlloc(). See EncodeLZSSInitAlloc
ippsEncodeRLE(). See EncodeRLE
ippsEncodeTNS_AAC(). See EncodeTNS_AAC
ippsEncodeTNS_AAC_32s_I, 10-170
ippsEntropy(). See Entropy
ippsEqual(). See Equal
ippsErf, 12-58
ippsErfc, 12-60
ippsErfInv, 12-61
ippsEstimateTNR_SBR(). See EstimateTNR_SBR
ippSetDenormAreZeros, 3-11
ippSetFlushToZero, 3-10
ippSetNumThreads, 3-13
ippsEvalDelta(). See EvalDelta
ippsEvalDelta_Aurora(). See EvalDelta_Aurora
ippsEvalFBank(). See EvalFBank
ippsExp, 12-24
ippsExp(). See Exp
ippsExpNegSqr(). See ExpNegSqr
ippsFBankFree(). See FBankFree
ippsFBankGetCenters(). See FBankGetCenters
ippsFBankGetCoeffs(). See FBankGetCoeffs
ippsFBankSetCenters(). See FBankSetCenters
ippsFBankSetCoeffs(). See FBankSetCoeffs
ippsFDPFree(). See FDPFree
ippsFDPFwd(). See FDPFwd
ippsFDPGetSize(). See FDPGetSize
ippsFDPInit(). See FDPInit
ippsFDPInitAlloc(). See FDPInitAlloc
ippsFDPInv(). See FDPInv
ippsFDPReset(). See FDPReset
ippsFDPResetGroup(). See FDPResetGroup
ippsFDPResetSfb(). See FDPResetSfb
ippsFFTFree_C(). See FFTFree_C
ippsFFTFree_R(). See FFTFree_R
ippsFFTFwd_CToC(). See FFTFwd_CToC
ippsFFTFwd_RToCCS(). See FFTFwd_RToCCS
ippsFFTFwd_RToPack(). See FFTFwd_RToPack
ippsFFTFwd_RToPerm(). See FFTFwd_RToPerm
ippsFFTGetBufSize_C(). See FFTGetBufSize_C
ippsFFTGetBufSize_R(). See FFTGetBufSize_R
ippsFFTGetSize_C(). See FFTGetSize_C
ippsFFTGetSize_R(). See FFTGetSize_R
ippsFFTInit_C(). See FFTInit_C
ippsFFTInit_R(). See FFTInit_R
ippsFFTInitAlloc_C(). See FFTInitAlloc_C
ippsFFTInitAlloc_R(). See FFTInitAlloc_R
ippsFFTInv_CCSToR(). See FFTInv_CCSToR

ippsFFTInv_CToC(). See FFTInv_CToC
ippsFFTInv_PackToR(). See FFTInv_PackToR
ippsFFTInv_PermToR(). See FFTInv_PermToR
ippsFillShortlist_Column(). See FillShortlist_Column
ippsFillShortlist_Row(). See FillShortlist_Row
ippsFilterMedian. See FilterMedian
ippsFilterUpdateEMNS(). See FilterUpdateEMNS
ippsFilterUpdateWiener(). See FilterUpdateWiener
ippsFind(). See Find, FindRev
ippsFindC(). See FindC, FindRevC
ippsFindCAny(). See FindCAny, FindRevCAny
ippsFindNearest(). See FindNearest
ippsFindNearestOne(). See FindNearestOne
ippsFindPeaks(). See FindPeaks
ippsFindRevC(). See FindC, FindRevC
ippsFindRevCAny(). See FindCAny, FindRevCAny
ippsFIR(). See FIR
ippsFIR_Direct(). See FIR_Direct
ippsFIRBlockFree(). See FIRBlockFree
ippsFIRBlockInitAlloc(). See FIRBlockInitAlloc
ippsFIRBlockOne(). See FIRBlockOne
ippsFIRFree(). See FIRFree
ippsFIRGenBandpass(). See FIRGenBandpass
ippsFIRGenBandstop(). See FIRGenBandstop
ippsFIRGenHighpass(). See FIRGenHighpass
ippsFIRGenLowpass(). See FIRGenLowpass
ippsFIRGetDlyLine(). See FIRGetDlyLine
ippsFIRGetStateSize(). See FIRGetStateSize
ippsFIRGetTaps(). See FIRGetTaps
ippsFIRInit(). See FIRInit
ippsFIRInitAlloc(). See FIRInitAlloc
ippsFIRLMS(). See FIRLMS
ippsFIRLMSFree(). See FIRLMSFree
ippsFIRLMSGetDlyLine(). See FIRLMSGetDlyLine
ippsFIRLMSGetTaps(). See FIRLMSGetTaps
ippsFIRLMSInitAlloc(). See FIRLMSInitAlloc
ippsFIRLMSMRFree(). See FIRLMSMRFree
ippsFIRLMSMRGetDlyLine(). See FIRLMSMRGetDlyLine
ippsFIRLMSMRGetDlyVal(). See FIRLMSMRGetDlyVal
ippsFIRLMSMRGetTaps(). See FIRLMSMRGetTaps
ippsFIRLMSMRGetTapsPointer(). See FIRLMSMRGetTapsPointer
ippsFIRLMSMRInitAlloc(). See FIRLMSMRInitAlloc
ippsFIRLMSMROne(). See FIRLMSMROne
ippsFIRLMSMROneVal(). See FIRLMSMROneVal
ippsFIRLMSMRPutVal(). See FIRLMSMRPutVal
ippsFIRLMSMRSetDlyLine(). See FIRLMSMRSetDlyLine
ippsFIRLMSMRSetMu(). See FIRLMSMRSetMu
ippsFIRLMSMRSetTaps(). See FIRLMSMRSetTaps
ippsFIRLMSMRUpdateTaps(). See FIRLMSMRUpdateTaps
ippsFIRLMSOne_Direct(). See FIRLMSOne_Direct
ippsFIRLMSSetDlyLine(). See FIRLMSSetDlyLine
ippsFIRMR_Direct(). See FIRMR_Direct
ippsFIRMRGetStateSize(). See FIRMRGetStateSize
ippsFIRMRInit(). See FIRMRInit
ippsFIRMRInitAlloc(). See FIRMRInitAlloc
ippsFIROne(). See FIROne
ippsFIROne_Direct(). See FIROne_Direct
ippsFIRSetDlyLine(). See FIRSetDlyLine
ippsFIRSetTaps(). See FIRSetTaps
ippsFIRSparse(). See FIRSparse
ippsFIRSparseGetStateSize(). See FIRSparseGetStateSize
ippsFIRSparseInit(). See FIRSparseInit
ippsFixedCodebookSearch_G729(). See FixedCodebookSearch_G729
ippsFixedCodebookSearch_RTA(). See FixedCodebookSearch_RTA
ippsFixedCodebookSearchRandom_RTA(). See FixedCodebookSearchRandom_RTA
ippsFlip(). See Flip
ippsFloor, 12-64
ippsFormVector(). See FormVector
ippsFormVectorVQ(). See FormVectorVQ
ippsFree, 3-5
ippsGainControl_G723(). See GainControl_G723
ippsGainControl_G729(). See GainControl_G729
ippsGainQuant_G723(). See GainQuant_G723
ippsGainQuant_G729(). See GainQuant_G729
ippsGaussianDist(). See GaussianDist
ippsGaussianMerge(). See GaussianMerge
ippsGaussianSplit(). See GaussianSplit
ippsGetBufSize_C(). See DFTGetBufSize_C
ippsGetCdbkSize(). See GetCdbkSize
ippsGetCodebook(). See GetCodebook
ippsGetLibVersion(). See GetLibVersion
ippsGetSizeMCRA(). See GetSizeMCRA
ippsGetVarPointDV(). See GetVarPointDV
ippsGITFree(). See GITFree

ippsGoertz(). See Goertz
ippsGoertzTwo(). See GoertzTwo
ippsHarmonicFilter(). See HarmonicFilter
ippsHarmonicNoiseSubtract_G723(). See HarmonicNoiseSubtract_G723
ippsHarmonicSearch_G723(). See HarmonicSearch_G723
ippsHash(). See Hash
ippsHighBandCoding_Aurora(). See HighBandCoding_Aurora
ippsHighPassFilter_G723(). See HighPassFilter_G723
ippsHighPassFilter_G729(). See HighPassFilter_G729
ippsHighPassFilter_RTA(). See HighPassFilter_RTA
ippsHighPassFilterInit_G729(). See HighPassFilterInit_G729
ippsHighPassFilterSize_G729(). See HighPassFilterSize_G729
ippsHilbert(). See Hilbert
ippsHilbertFree(). See HilbertFree
ippsHilbertInitAlloc(). See HilbertInitAlloc
ippsHuffFree(). See HuffFree
ippsHuffGetDstBuffSize(). See ippsHuffGetDstBuffSize
ippsHuffGetLenCodeTable(). See HuffGetLenCodeTable
ippsHuffGetSize(). See HuffGetSize
ippsHuffLenCodeTablePack(). See ippsHuffLenCodeTablePack
ippsHuffLenCodeTableUnpack(). See ippsHuffLenCodeTableUnpack
ippsHuffmanDecode_MP3(). See HuffmanDecode_MP3
ippsHuffmanEncode_MP3(). See HuffmanEncode_MP3
ippsHuffmanEncode_MP3_32s1u, 10-102
ippsHypot, 12-21
ippsIIR(). See IIR
ippsIIR_BiQuadDirect(). See IIR_BiQuadDirect
ippsIIR_Direct(). See IIR_Direct
ippsIIR16s_G723(). See IIR16s_G723
ippsIIR16s_G729(). See IIR16s_G729
ippsIIRFree(). See IIRFree
ippsIIRGetDlyLine(). See IIRGetDlyLine
ippsIIRGetStateSize().See IIRGetStateSize
ippsIIRGetStateSize_BiQuad().See IIRGetStateSize_BiQuad
ippsIIRInitAlloc(). See IIRInitAlloc
ippsIIRInitAlloc_BiQuad(). See IIRInitAlloc_BiQuad
ippsIIROne(). See IIROne
ippsIIROne_BiQuadDirect(). See IIROne_BiQuadDirect
ippsIIROne_Direct(). See IIROne_Direct
ippsIIRSetDlyLine(). See IIRSetDlyLine
ippsIIRSparse(). See IIRSparse
ippsIIRSparseGetStateSize(). See IIRSparseGetStateSize
ippsIIRSparseInit(). See IIRSparseInit
ippsImag(). See Imag
ippsIndexSelect_VQ(). See IndexSelect_VQ
ippsInitBitReservoir_MP3, 10-107
ippsInitMCRA(). See InitMCRA
ippsInsert (). See Insert
ippsInterleave(). See Interleave
ippsInterpolate_G729(). See Interpolate_G729
ippsInv, 12-5
ippsInvCbrt, 12-14
ippsInvSqrt, 12-11
ippsInvSqrt(). See InvSqrt
ippsJoin(). See Join
ippsJoinScaled(). See JoinScaled
ippsJointStereoEncode_MP3(). See JointStereoEncode_MP3
ippsJointStereoEncode_MP3_32s_I, 10-91
ippsLagWindow_G729(). See LagWindow_G729
ippsLevinsonDurbin_G723(). See LevinsonDurbin_G723
ippsLevinsonDurbin_G729(). See LevinsonDurbin_G729
ippsLevinsonDurbin_GSMAMR, 9-124
ippsLevinsonDurbin_RTA(). See LevinsonDurbin_RTA
ippsLinearPrediction(). See LinearPrediction
ippsLinearToMel(). See LinearToMel
ippsLinToALaw(). See LinToALaw
ippsLinToMuLaw(). See LinToMuLaw
ippsLn, 12-26
ippsLn(). See Ln
ippsLog10, 12-28
ippsLogAdd(). See LogAdd
ippsLogGauss(). See LogGauss
ippsLogGaussAdd(). See LogGaussAdd
ippsLogGaussAddMultiMix(). See

LogGaussAddMultiMix
ippsLogGaussMax(). See LogGaussMax
ippsLogGaussMaxMultiMix(). See LogGaussMaxMultiMix
ippsLogGaussMixture(). See LogGaussMixture
ippsLogGaussMixtureSelect(). See LogGaussMixtureSelect
ippsLogGaussMultiMix(). See LogGaussMultiMix
ippsLogGaussSingle(). See LogGaussSingle
ippsLogSub(). See LogSub
ippsLogSum(). See LogSum
ippsLongTermPostFilter_G729(). See LongTermPostFilter_G729
ippsLongTermPredict_AAC(). See LongTermPredict_AAC
ippsLongTermPredict_AAC_32s, 10-172
ippsLongTermReconstruct_AAC(). See LongTermReconstruct_AAC
ippsLongTermReconstruct_AAC_32s, 10-166
ippsLowercase(). See Lowercase
ippsLowercaseLatin(). See Lowercase
ippsLowHighFilter_Aurora(). See LowHighFilter_Aurora
ippsLPCToLSF_G723(). See LPCToLSF_G723
ippsLPCToLSP_G729(). See LPCToLSP_G729
ippsLPCToLSP_GSMAMR_16s, 9-126
ippsLPCToLSP_RTA(). See LPCToLSP_RTA
ippsLPToCepstrum(). See LPToCepstrum
ippsLPToLSP(). See LPToLSP
ippsLPToReflection(). See LPToReflection
ippsLPToSpectrum(). See LPToSpectrum
ippsLSFDecode_G723(). See LSFDecode_G723
ippsLSFDecode_G729(). See LSFDecode_G729
ippsLSFDecodeErased_G729(). See LSFDecodeErased_G729
ippsLSFQuant_G723(). See LSFQuant_G723
ippsLSFQuant_G729(). See LSFQuant_G729
ippsLSFToLPC_G723(). See LSFToLPC_G723
ippsLSFToLSP_G729(). See LSFToLSP_G729
ippsLShiftC(). See LShiftC
ippsLSPQuant_G729(). See LSPQuant_G729
ippsLSPQuant_GSMAMR_16s, 9-129
ippsLSPQuant_RTA(). See LSPQuant_RTA
ippsLSPToLPC_G729(). See LSPToLPC_G729
ippsLSPToLPC_GSMAMR_16s, 9-127
ippsLSPToLPC_RTA(). See LSPToLPC_RTA
ippsLSPToLSF_G729(). See LSPToLSF_G729
ippsLtpUpdate_AAC(). See LtpUpdate_AAC
ippsLtpUpdate_AAC_32s, 10-174
ippsLZ77Free(). See LZ77Free
ippsLZSSFree(). See LZSSFree
ippsLZSSGetSize(). See LZSSGetSize
ippsMagnitude(). See Magnitude
ippsMagSquared(). See MagSquared
ippsMahDist(). See MahDist
ippsMahDistMultiMix(). See MahDistMultiMix
ippsMahDistSingle(). See MahDistSingle
ippsMakeFloat(). See MakeFloat
ippsMalloc, 3-4
ippsMatVecMul(). See MatVecMul
ippsMax(). See Max
ippsMaxAbs(). See MaxAbs
ippsMaxAbsIndx(). See MaxAbsIndx
ippsMaxEvery(). See MaxEvery
ippsMaxIndx(). See MaxIndx
ippsMaxOrder(). See MaxOrder
ippsMDCTFwd(). See MDCTFwd, MDCTInv
ippsMDCTFwd_AAC_32s, 10-167
ippsMDCTFwd_AAC_32s(). See MDCTFwd_AAC_32s
ippsMDCTFwd_AAC_32s_I(). See MDCTFwd_AAC_32s_I
ippsMDCTFwd_MP3(). See MDCTFwd_MP3
ippsMDCTFwd_MP3_32s, 10-85
ippsMDCTFwdFree(). See MDCTFwdFree, MDCTInvFree
ippsMDCTFwdGetBufSize(). See MDCTFwdGetBufSize, MDCTInvGetBufSize
ippsMDCTFwdGetSize(). See MDCTFwdGetSize, MDCTInvGetSize
ippsMDCTFwdInit(). See MDCTFwdInit, MDCTInvInit
ippsMDCTFwdInitAlloc(). See MDCTFwdInitAlloc, MDCTInvInitAlloc
ippsMDCTInv(). See MDCTFwd, MDCTInv
ippsMDCTInv_AAC_32s_I(). See MDCTInv_AAC_32s_I
ippsMDCTInv_AAC_32s16s, 10-157
ippsMDCTInv_AAC_32s16s(). See MDCTInv_AAC_32s16s
ippsMDCTInv_MP3(). See MDCTInv_MP3
ippsMDCTInvFree(). See MDCTFwdFree, MDCTInvFree
ippsMDCTInvGetBufSize(). See MDCTFwdGetBufSize, MDCTInvGetBufSize

ippsMDCTInvGetSize(). See MDCTFwdGetSize, MDCTInvGetSize
ippsMDCTInvInit(). See MDCTFwdInit, MDCTInvInit
ippsMDCTInvInitAlloc(). See MDCTFwdInitAlloc, MDCTInvInitAlloc
ippsMean(). See Mean
ippsMeanColumn(). See MeanColumn
ippsMeanStdDev(). See MeanStdDev
ippsMeanVarAcc(). See MeanVarAcc
ippsMeanVarColumn(). See MeanVarColumn
ippsMelFBankGetSize(). See MelFBankGetSize
ippsMelFBankInit(). See MelFBankInit
ippsMelFBankInitAlloc(). See MelFBankInitAlloc
ippsMelFBankInitAlloc_Aurora(). See MelFBankInitAlloc_Aurora
ippsMelLinFBankInitAlloc(). See MelLinFBankInitAlloc
ippsMelToLinear(). See MelToLinear
ippsMin(). See Min
ippsMinAbs(). See MinAbs
ippsMinAbsIndx(). See MinAbsIndx
ippsMinEvery(). See MinEvery
ippsMinIndxt(). See MinIndx
ippsMinMax(). See MinMax
ippsMinMaxIndx(). See MinMaxIndx
ippsModf, 12-71
ippsMove(). See Move
ippsMPMLQFixedCodebookSearch_G723(). See MPMLQFixedCodebookSearch_G723
ippsMTFFree(). See MTFFree
ippsMTFFwd(). See MTFFwd
ippsMTFGetSize(). See MTFGetSize
ippsMTFInit(). See MTFInit
ippsMTFInitAlloc(). See MTFInitAlloc
ippsMTFInv(). See MTFInv
ippsMul(). See Mul
ippsMul_NR(). See Mul_NR
ippsMuLawToALaw(). See MuLawToALaw
ippsMuLawToLin(). See MuLawToLin
ippsMulC_NR(). See MulC_NR
ippsMulColumn(). See MulColumn
ippsMulPack(). See MulPack
ippsMulPackConj(). See MulPackConj
ippsMulPerm(). See MulPerm
ippsMulPowerC_NR(). See MulPowerC_NR
ippsNearbyInt, 12-68
ippsNewVar(). See NewVar
ippsNoiselessDecode_AAC, 10-173
ippsNoiselessDecode_AAC(). See NoiselessDecode_AAC
ippsNoiselessDecoder_LC_AAC, 10-140
ippsNoiselessDecoder_LC_AAC(). See NoiselessDecoder_LC_AAC
ippsNoiseSpectrumUpdate_Aurora(). See NoiseSpectrumUpdate_Aurora
ippsNorm(). See Norm
ippsNormalize(). See Normalize
ippsNormalizeColumn(). See NormalizeColumn
ippsNormalizeInRange(). See NormalizeInRange
ippsNormDiff(). See NormDiff
ippsNormEnergy(). See NormEnergy
ippsNot(). See Not
ippsNthMaxElement(). See NthMaxElement
ippsOpenLoopPitchSearch_G723(). See OpenLoopPitchSearch_G723
ippsOpenLoopPitchSearch_G729(). See OpenLoopPitchSearch_G729
ippsOr(). See Or
ippsOrC(). See OrC
ippsOutProbPreCalc(). See OutProbPreCalc
ippsPackBits(). See PackBits
ippsPackFrameHeader_MP3, 10-104
ippsPackFrameHeader_MP3(). See PackFrameHeader_MP3
ippsPackScalefactors_MP3(). See PackScalefactors_MP3
ippsPackScalefactors_MP3_8s1u, 10-99
ippsPackSideInfo_MP3, 10-105
ippsPackSideInfo_MP3(). See PackSideInfo_MP3
ippsPeriodicity(). See Periodicity
ippsPeriodicityLSPE(). See PeriodicityLSPE
ippsPhase(). See Phase, complex
ippsPitchmarkToF0(). See PitchmarkToF0
ippsPitchPostFilter_G723(). See PitchPostFilter_G723
ippsPolarToCart(). See PolarToCart, complex
ippsPostFilter_RTA(). See PostFilter_RTA
ippsPostFilterGetStateSize_RTA(). See PostFilterGetStateSize_RTA
ippsPostFilterInit_RTA(). See PostFilterInit_RTA
ippsPow, 12-16
ippsPow34(). See Pow34
ippsPow43(). See Pow43
ippsPow43Scale(). See Pow43Scale

ippsPowerSpectr(). See PowerSpectr, complex
ippsPowx, 12-19
ippsPredictCoef_SBR(). See PredictCoef_SBR
ippsPredictOneCoef_SBR(). See PredictOneCoef_SBR
ippsPreemphasize(). See Preemphasize
ippsPreemphasize_G729(). See Preemphasize_G729
ippsPreSelect_VQ(). See PreSelect_VQ
ippsPsych_MP3_16s, 10-86
ippsPsychoacousticModelTwo_MP3(). See PsychoacousticModelTwo_MP3
ippsQMFDecode_RTA(). See QMFDecode_RTA
ippsQMFEncode_RTA(). See QMFEncode_RTA
ippsQMFGetStateSize_RTA(). See QMFGetStateSize_RTA
ippsQMFInit_RTA(). See QMFInit_RTA
ippsQRTransColumn(). See QRTransColumn
ippsQuantInv_AAC(). See QuantInv_AAC
ippsQuantInv_AAC_32s_I, 10-146
ippsQuantize_MP3(). See Quantize_MP3
ippsQuantize_MP3_32s_I, 10-94
ippsQuantLSPDecode_GSMAMR_16s, 9-132
ippsRandGauss(). See RandGauss
ippsRandGauss_Direct(). See RandGauss_Direct
ippsRandGaussFree(). See RandGaussFree
ippsRandGaussGetSize(). See RandGaussGetSize
ippsRandGaussInit(). See RandGaussInit
ippsRandGaussInitAlloc(). See RandGaussInitAlloc
ippsRandomNoiseExcitation_G729B(). See RandomNoiseExcitation_G729B
ippsRandUniform_Direct(). See RandUniform_Direct
ippsRandUniformFree(). See RandUniformFree
ippsRandUniformGetSize(). See RandUniformGetSize
ippsRandUniformInit(). See RandUniformInit
ippsRandUniformInitAlloc(). See RandUniformInitAlloc
ippsRandUnifrom(). See RandUnifrom
ippsReal(). See Real
ippsRealToCplx(). See RealToCplx
ippsRecSqrt(). See RecSqrt
ippsReflectionToAR(). See ReflectionToAR
ippsReflectionToLP(). See ReflectionToLP
ippsReflectionToTilt(). See ReflectionToTilt
ippsRegExpFind(). See RegExpFind
ippsRegExpFree(). See RegExpFree
ippsRegExpGetSize(). See RegExpGetSize
ippsRegExpInit(). See RegExpInit
ippsRegExpInitAlloc(). See RegExpInitAlloc
ippsRegExpSetMatchLimit(). See RegExpSetMatchLimit
ippsRemove(). See Remove
ippsReplaceC(). See ReplaceC
ippsReQuantize_MP3(). See ReQuantize_MP3
ippsResamplePolyphase(). See ResamplePolyphase
ippsResamplePolyphaseFree(). See ResamplePolyphaseFree
ippsResamplePolyphaseGetFilter(). See ResamplePolyphaseGetFilter
ippsResamplePolyphaseGetSize(). See ResamplePolyphaseGetSize
ippsResamplePolyphaseInit(). See ResamplePolyphaseInit
ippsResamplePolyphaseInitAlloc(). See ResamplePolyphaseInitAlloc
ippsResamplePolyphaseSetFilter(). See ResamplePolyphaseSetFilter
ippsResidualFilter_Aurora(). See ResidualFilter_Aurora
ippsResidualFilter_G729(). See ResidualFilter_G729
ippsRint, 12-70
ippsRLEFree_BZ2(). See RLEFree_BZ2
ippsRound, 12-67
ippsRShiftC(). See RShiftC
ippsSampleDown(). See SampleDown
ippsSampleUp(). See SampleUp
ippsScale(). See Scale
ippsScaleLM(). See ScaleLM
ippsSchur(). See Schur
ippsSet(). See Set
ippsShortTermPostFilter_G729(). See ShortTermPostFilter_G729
ippsSignChangeRate(). See SignChangeRate
ippsSin, 12-32
ippsSinC(). See SinC
ippsSinCos, 12-34
ippsSinh, 12-48
ippsSmoothedPowerSpectrumAurora(). See SmoothedPowerSpectrumAurora
ippsSortAscend(). See SortAscend
ippsSortDescend(). See SortDescend
ippsSortIndexAscend(). See SortIndexAscend

ippsSortIndexDescend(). See SortIndexDescend
ippsSplitC(). See SplitC
ippsSplitScaled(). See SplitScaled
ippsSplitVQ(). See SplitVQ
ippsSpread(). See Spread
ippsSqr(). See Sqr
ippsSqrt, 12-9
ippsSqrt(). See Sqrt
ippsStdDev(). See StdDev
ippsSub(). See Sub
ippsSubC(). See SubC
ippsSubCRev(). See SubCRev
ippsSubRow(). See SubRow
ippsSum(). See Sum
ippsSumColumn(). See SumColumn
ippsSumColumnAbs(). See SumColumnAbs
ippsSumColumnSqr(). See SumColumnSqr
ippsSumLn(). See SumLn
ippsSumMeanVar(). See SumMeanVar
ippsSumRow(). See SumRow
ippsSumRowAbs(). See SumRowAbs
ippsSumRowSqr(). See SumRowSqr
ippsSVD(). See SVD
ippsSwapBytes(). See SwapBytes
ippsSynthesisDownFilter_SBR(). See SynthesisDownFilter_SBR
ippsSynthesisDownFilterFree_SBR(). See SynthesisDownFilterFree_SBR
ippsSynthesisDownFilterGetSize_SBR(). See SynthesisDownFilterGetSize_SBR
ippsSynthesisDownFilterInit_SBR(). See SynthesisDownFilterInit_SBR
ippsSynthesisDownFilterInitAlloc_SBR(). See SynthesisDownFilterInitAlloc_SBR
ippsSynthesisFilter_DTS(). See SynthesisFilter_DTS
ippsSynthesisFilter_G723(). See SynthesisFilter_G723
ippsSynthesisFilter_G729(). See SynthesisFilter_G729
ippsSynthesisFilter_PQMF_MP3(). See SynthesisFilter_PQMF_MP3
ippsSynthesisFilter_SBR(). See SynthesisFilter_SBR
ippsSynthesisFilterFree_DTS(). See SynthesisFilterFree_DTS
ippsSynthesisFilterFree_PQMF_MP3(). See SynthesisFilterFree_PQMF_MP3
ippsSynthesisFilterFree_SBR(). See SynthesisFilterFree_SBR
ippsSynthesisFilterGetSize_DTS(). See SynthesisFilterGetSize_DTS
ippsSynthesisFilterGetSize_PQMF_MP3(). See SynthesisFilterGetSize_PQMF_MP3
ippsSynthesisFilterGetSize_SBR(). See SynthesisFilterGetSize_SBR
ippsSynthesisFilterInit_DTS(). See SynthesisFilterInit_DTS
ippsSynthesisFilterInit_PQMF_MP3(). See SynthesisFilterInit_PQMF_MP3
ippsSynthesisFilterInitAlloc_DTS(). See SynthesisFilterInitAlloc_DTS
ippsSynthesisFilterInitAlloc_PQMF_MP3(). See SynthesisFilterInitAlloc_PQMF_MP3
ippsSynthesisFilterInitAlloc_SBR(). See SynthesisFilterInitAlloc_SBR
ippsSynthesisInit_SBR(). See SynthesisInit_SBR
ippsSynthPQMF_MP3(). See SynthPQMF_MP3
ippsTabsCalculation_Aurora(). See TabsCalculation_Aurora
ippsTan, 12-36
ippsTanh, 12-50
ippStaticInit, 3-16
ippStaticInitCpu, 3-17
ippsThreshold(). See Threshold
ippsThreshold_GT(). See Threshold_GT
ippsThreshold_GTAbs(). See Threshold_GTAbs
ippsThreshold_GTVal(). See Threshold_GTVal
ippsThreshold_LT(). See Threshold_LT
ippsThreshold_LTAbs(). See Threshold_LTAbs
ippsThreshold_LTInv(). See Threshold_LTInv
ippsThreshold_LTVal(). See Threshold_LTVal
ippsThreshold_LTValGTVal(). See Threshold_LTValGTVal
ippsTiltCompensation_G723(). See TiltCompensation_G723
ippsTiltCompensation_G729(). See TiltCompensation_G729
ippsToeplizMatrix_G723(). See ToeplizMatrix_G723
ippsToeplizMatrix_G729(). See ToeplizMatrix_G729
ippsTone_Direct(). See Tone_Direct
ippsToneFree(). See ToneFree
ippsToneGetStateSizeQ15(). See ToneGetStateSizeQ15
ippsToneInitAllocQ15(). See ToneInitAllocQ15
ippsToneInitQ15(). See ToneInitQ15

ippsToneQ15(). See ToneQ15
ippsTriangle_Direct(). See Triangle_Direct
ippsTriangleFree(). See TriangleFree
ippsTriangleGetStateSizeQ15(). See TriangleGetStateSizeQ15
ippsTriangleInitAllocQ15(). See TriangleInitAllocQ15
ippsTriangleInitQ15(). See TriangleInitQ15
ippsTriangleQ15(). See TriangleQ15
ippsTrimC(). See TrimC
ippsTrimCAny(). See TrimCAny
ippsTrimEndCAny(). See TrimCAny
ippsTrimStartCAny(). See TrimCAny
ippsTrunc, 12-66
ippsUnitCurve(). See UnitCurve
ippsUnpackADIFHeader_AAC, 10-136
ippsUnpackADIFHeader_AAC(). See UnpackADIFHeader_AAC
ippsUnpackADTSFrameHeader_AAC, 10-137
ippsUnpackADTSFrameHeader_AAC(). See UnpackADTSFrameHeader_AAC
ippsUnpackFrameHeader_MP3(). See UnpackFrameHeader_MP3
ippsUnpackScaleFactors_MP3(). See UnpackScaleFactors_MP3
ippsUnpackSideInfo_MP3(). See UnpackSideInfo_MP3
ippsUpdateGConst(). See UpdateGConst
ippsUpdateLinear(). See UpdateLinear
ippsUpdateMean(). See UpdateMean
ippsUpdateNoisePSDMCRA(). See UpdateNoisePSDMCRA
ippsUpdatePathMetricsDV(). See UpdatePathMetricsDV
ippsUpdatePower(). See UpdatePower
ippsUpdateVar(). See UpdateVar
ippsUpdateWeight(). See UpdateWeight
ippsUppercase(). See Uppercase
ippsUppercaseLatin(). See Uppercase
ippsVADDecision_Aurora(). See VADDecision_Aurora
ippsVADFlush_Aurora(). See VADFlush_Aurora
ippsVADGetBufSize_Aurora(). See VADGetBufSize_Aurora
ippsVADInit_Aurora(). See VADInit_Aurora
ippsVarColumn(). See VarColumn
ippsVecMatMul(). See VecMatMul
ippsVectorJaehne(). See VectorJaehne
ippsVectorRamp(). See VectorRamp
ippsVectorSlope(). See VectorSlope
ippsVLCCountBits(). See VLCCountBits
ippsVLCCountEscBits_AAC(). See VLCCountEscBits_AAC
ippsVLCCountEscBits_MP3(). See VLCCountEscBits_MP3
ippsVLCDecodeBlock(). See VLCDecodeBlock
ippsVLCDecodeEscBlock_AAC(). See VLCDecodeEscBlock_AAC
ippsVLCDecodeEscBlock_MP3(). See VLCDecodeEscBlock_MP3
ippsVLCDecodeFree(). See VLCDecodeFree
ippsVLCDecodeGetSize(). See VLCDecodeGetSize
ippsVLCDecodeInit(). See VLCDecodeInit
ippsVLCDecodeInitAlloc(). See VLCDecodeInitAlloc
ippsVLCDecodeOne(). See VLCDecodeOne
ippsVLCDecodeUTupleBlock(). See VLCDecodeUTupleBlock
ippsVLCDecodeUTupleEscBlock_AAC(). See VLCDecodeUTupleEscBlock_AAC
ippsVLCDecodeUTupleEscBlock_MP3(). See VLCDecodeUTupleEscBlock_MP3
ippsVLCDecodeUTupleFree(). See VLCDecodeUTupleFree
ippsVLCDecodeUTupleGetSize(). See VLCDecodeUTupleGetSize
ippsVLCDecodeUTupleInit(). See VLCDecodeUTupleInit
ippsVLCDecodeUTupleInitAlloc(). See VLCDecodeUTupleInitAlloc
ippsVLCDecodeUTupleOne(). See VLCDecodeUTupleOne
ippsVLCEncodeBlock(). See VLCEncodeBlock
ippsVLCEncodeEscBlock_AAC(). See VLCEncodeEscBlock_AAC
ippsVLCEncodeEscBlock_MP3(). See VLCEncodeEscBlock_MP3
ippsVLCEncodeFree(). See VLCEncodeFree
ippsVLCEncodeGetSize(). See VLCEncodeGetSize
ippsVLCEncodeInit(). See VLCEncodeInit
ippsVLCEncodeInitAlloc(). See VLCEncodeInitAlloc
ippsVLCEncodeOne(). See VLCEncodeOne
ippsVQ(). See VQ
ippsVQCodeBookFree(). See VQCodeBookFree
ippsVQCodeBookGetSize(). See

VQCodeBookGetSize
ippsVQCodeBookInit(). See VQCodeBookInit
ippsVQCodeBookInitAlloc(). See VQCodeBookInitAlloc
ippsVQMainSelect(). See VQMainSelect
ippsVQrReconstruction(). See VQReconstruction
ippsVQSingle_Sort(). See VQSingle_Sort
ippsVQSingle_Thresh(). See VQSingle_Thresh
ippsWaveProcessing_Aurora(). See WaveProcessing_Aurora
ippsWeightedMeanColumn(). See WeightedMeanColumn
ippsWeightedMeanVarColumn(). See WeightedMeanVarColumn
ippsWeightedSum(). See WeightedSum
ippsWeightedVarColumn(). See WeightedVarColumn
ippsWienerFilterDesign_Aurora(). See WienerFilterDesign_Aurora
ippsWinBartlett(). See WinBartlett
ippsWinBlackman(). See WinBlackman
ippsWinHamming(). See WinHamming
ippsWinHann(). See WinHann
ippsWinKaiser(). See WinKaiser
ippsWTFwd(). See WTFwd
ippsWTFwdFree(). See WTFwdFree
ippsWTFwdGetDlyLine(). See WTFwdGetDlyLine
ippsWTFwdInitAlloc(). See WTFwdInitAlloc
ippsWTFwdSetDlyLine(). See WTFwdSetDlyLine
ippsWTHaarFwd(). See WTHaarFwd
ippsWTHaarInv(). See WTHaarInv
ippsWTInv(). See WTInv
ippsWTInvFree(). See WTInvFree
ippsWTInvGetDlyLine(). See WTInvGetDlyLine
ippsWTInvInitAlloc(). See WTInvInitAlloc
ippsWTInvSetDlyLine(). See WTInvSetDlyLine
ippsXor(). See Xor
ippsXorC(). See XorC
ippsZero(). See Zero
ippsZeroMean(). See ZeroMean
ISFQuant_AMRWB, 9-205
ISFQuantDecode_AMRWB, 9-207
ISFQuantDecode_AMRWBE, 9-239
ISFQuantDecodeDTX_AMRWB, 9-209
ISFQuantDecodeHighBand_AMRWBE, 9-240
ISFQuantDTX_AMRWB, 9-208
ISFQuantHighBand_AMRWBE, 9-241
ISFToISP_AMRWB, 9-177
ISPToISF_Norm_AMRWB, 9-176
ISPToLPC_AMRWB, 9-174

J

Join, 5-63
JoinScaled, 5-65
JointStereoEncode_MP3, 10-91

L

LagWindow_G729, 9-41
LevinsonDurbin_G723, 9-92
LevinsonDurbin_G728, 9-286
LevinsonDurbin_G729, 9-28
LevinsonDurbin_GSMAMR, 9-124
LevinsonDurbin_RTA, 9-336
line spectral frequencies, 9-32
linear prediction analysis, 9-27
linear predictors, 10-34
LinearPrediction, 8-25
LinearToMel, 8-44
LinToALaw, 9-330
LinToMuLaw, 9-328
Ln, 5-49, 12-26
Log10, 12-28
LogAdd, 8-94
LogGauss, 8-105
LogGaussAdd, 8-118
LogGaussAddMultiMix, 8-121
LogGaussMax, 8-111
LogGaussMaxMultiMix, 8-115
LogGaussMixture, 8-123
LogGaussMixtureSelect, 8-127
LogGaussMultiMix, 8-109
LogGaussSingle, 8-102
Logical and shift functions, 5-6 thru ??
And, 5-7
AndC, 5-6
LShiftC, 5-13
Not, 5-12
Or, 5-9
OrC, 5-8
RShiftC, 5-14
Xor, 5-11
XorC, 5-10

LogSub, 8-95
LogSum, 8-96
LongTermPostFilter_G729, 9-68
LongTermPredict_AAC, 10-172
LongTermReconstruct_AAC, 10-166
Lowercase, 11-19
LowercaseLatin, 11-19
LowHighFilter_Aurora, 8-220
LP coefficients, 9-36
LPCInverseFilter_G728, 9-279
LPCToISP_AMRWB, 9-173
LPCToISP_AMRWBE, 9-221
LPCToLSF_G723, 9-93
LPCToLSP_G729, 9-30
LPCToLSP_GSMAMR, 9-126
LPCToLSP_RTA, 9-335
LPToCepstrum, 8-31
LPToLSP, 8-40
LPToReflection, 8-33
LPToSpectrum, 8-30
LSFDecode_G723, 9-96
LSFDecode_G729, 9-34
LSFDecodeErased_G729, 9-35
LSFQuant_G723, 9-97
LSFQuant_G729, 9-32
LSFToLPC_G723, 9-95
LSFToLSP_G729, 9-32
LSFToLSP_GSMAMR, 9-129
LShiftC, 5-13
LSP coefficients, 9-41
LSPQuant_G729, 9-37
LSPQuant_GSMAMR, 9-129
LSPQuant_RTA, 9-345
LSPToLPC_G729, 9-36
LSPToLPC_GSMAMR, 9-127
LSPToLPC_RTA, 9-335
LSPToLSF_G729, 9-40
LtpUpdate_AAC, 10-174
LZ77 dictionary-based compression, 13-50
LZ77Free, 13-58
LZSS compression, 13-40 thru ??
    EncodeLZSSInitAlloc, 13-41
LZSS compression algorithm, 13-40
LZSSFree, 13-41
LZSSGetSize, 13-42

M

MA predictor, 9-33
Magnitude, 5-69
MagSquared, 5-71
MahDist, 8-99
MahDistMultiMix, 8-100
MahDistSingle, 8-98
MakeFloat, 10-24
mantissa conversion and scaling functions, 10-22
manual organization, 1-5
MatVecMul, 8-20
Max, 5-119
MaxAbs, 5-121
MaxAbsIndx, 5-122
MaxEvery, 5-141
MaxIndx, 5-120
MaxOrder, 5-97
MDCTFwd, MDCTInv, 10-31
MDCTFwd_AAC_32s, 10-167
MDCTFwd_AAC_32s_I, 10-168
MDCTFwd_MP3, 10-85
MDCTFwdFree, MDCTInvFree, 10-28
MDCTFwdGetBufSize, MDCTInvGetBufSize, 10-30
MDCTFwdGetSize, MDCTInvGetSize, 10-29
MDCTFwdInit, MDCTInvInit, 10-27
MDCTFwdInitAlloc, MDCTInvInitAlloc, 10-25
MDCTInv_AAC_32s_I, 10-159
MDCTInv_AAC_32s16s, 10-157
MDCTInv_MP3, 10-117
Mean, 5-128
MeanColumn, 8-140
MeanSrdDev, 5-132
MeanVarAcc, 8-153
MeanVarColumn, 8-142
Median filter functions, 6-131 thru 6-132
    FilterMedian, 6-131
MelFBankGetSize, 8-46
MelFBankInit, 8-47
MelFBankInitAlloc, 8-48
MelFBankInitAlloc_Aurora, 8-216
MelLinFBankInitAlloc, 8-50
MelToLinear, 8-43
memory allocation functions, 3-3 thru 3-6
    ippsFree, 3-5
    ippsMalloc, 3-4
Min, 5-123

MinAbs, 5-125
MinAbsIndx, 5-125
MinEvery, 5-141
MinIndx, 5-124
MinMax, 5-126
MinMaxIndx, 5-127
MMX technology, 1-1
Model adaptation functions, 8-172 thru 8-184
AddMulColumn, 8-172
AddMulRow, 8-173
DotProdColumn, 8-175
MulColumn, 8-176
QRTransColumn, 8-174
SumColumnAbs, 8-177
SumColumnSqr, 8-178
SumRowAbs, 8-179
SumRowSqr, 8-180
SVD, SVDSort, 8-181
WeightedSum, 8-182
Model estimation functions, 8-139 thru 8-171
BhatDist, 8-161
DcsClustLAccumulate, 8-169
DcsClustLCompute, 8-170
Entropy, 8-158
ExpNegSqr, 8-160
GaussianDist, 8-154
GaussianMerge, 8-157
GaussianSplit, 8-156
MeanColumn, 8-140
MeanVarAcc, 8-153
MeanVarColumn, 8-142
NormalizeColumn, 8-150
NormalizeInRange, 8-151
OutProbPreCalc, 8-167
SinC, 8-159
UpdateGConst, 8-166
UpdateMean, 8-163
UpdateVar, 8-164
UpdateWeight, 8-165
VarColumn, 8-141
WeightedMeanColumn, 8-144
WeightedMeanVarColumn, 8-147
WeightedVarColumn, 8-145
Model evaluation functions, 8-92 thru 8-139
AddNRows, 8-92
BuildSignTable, 8-130
DTW, 8-137
FillShortlist_Column, 8-134
FillShortlist_Row, 8-132
LogAdd, 8-94
LogGauss, 8-105
LogGaussAdd, 8-118
LogGaussAddMultiMix, 8-121
LogGaussMax, 8-111
LogGaussMaxMultiMix, 8-115
LogGaussMixture, 8-123
LogGaussMixtureSelect, 8-127
LogGaussMultiMix, 8-109
LogGaussSingle, 8-102
LogSub, 8-95
LogSum, 8-96
MahDist, 8-99
MahDistMultiMix, 8-100
MahDistSingle, 8-98
ScaleLM, 8-93
Modf, 12-71
modified discrete cosine transform (MDCT), 10-25
Move, 4-6
Move To Front data transform method, 13-105
move to front data transform.See MTF data transform
MP3 audio decoder, 10-108
MP3 audio decoder functions, 10-108 thru 10-120
HuffmanDecode_MP3, 10-113
MDCTInv_MP3, 10-117
ReQuantize_MP3, 10-115
SynthesisFilter_PQMF_MP3, 10-123
SynthesisFilterFree_PQMF_MP3, 10-122
SynthesisFilterGetSize_PQMF_MP3, 10-122
SynthesisFilterInit_PQMF_MP3, 10-120
SynthesisFilterInitAlloc_PQMF_MP3, 10-121
SynthPQMF_MP3, 10-119
UnpackFrameHeader_MP3, 10-109
UnpackScaleFactors_MP3, 10-111
UnpackSideInfo_MP3, 10-110
MP3 audio encoder
bit reservoir structure, 10-77
psychoacoustic model, 10-76
psychoacoustic model structure, 10-75
MP3 audio encoder API
enumerated types, 10-78
MP3, data structures, 10-73

MP3, macro and constant definitions, 10-72
MPEG-2
    global macros, 10-125
MPMLQFixedCodebookSearch_G723, 9-102
MTF data transform, 13-105 thru ??
    MTFFree, 13-107
    MTFGetSize, 13-108
    MTFInit, 13-107
    MTFInitAlloc, 13-106
MTFFree, 13-107
MTFFwd, 13-108
MTFGetSize, 13-108
MTFInit, 13-107
MTFInitAlloc, 13-106
MTFInv, 13-109
Mul, 5-24
Mul_NR, 9-8
MuLawToALaw, 9-331
MuLawToLin, 9-327
MulC, 5-22
MulC_NR, 9-10
MulColumn, 8-176
MulPack, 7-14
MulPackConj, 7-18
MulPerm, 7-14, 7-16
MulPowerC_NR, 9-11
Multiplication of packed data, 7-13 thru 7-18
    MulPack, 7-14
    MulPackConj, 7-18
    MulPerm, 7-14, 7-16
multi-rate FIR LMS filter functions
    FIRLMSMRFree, 6-82
    FIRLMSMRGetDlyLine, 6-87
    FIRLMSMRGetDlyVal, 6-89
    FIRLMSMRGetTaps, 6-84
    FIRLMSMRGetTapsPointer, 6-86
    FIRLMSMRInitAlloc, 6-81
    FIRLMSMROne, 6-90
    FIRLMSMROneVal, 6-92
    FIRLMSMRPutVal, 6-90
    FIRLMSMRSetDlyLine, 6-88
    FIRLMSMRSetMu, 6-83
    FIRLMSMRSetTaps, 6-85
    FIRLMSMRUpdateTaps, 6-83

N

NearbyInt, 12-68
NewVar, 8-70
NoiselessDecode_AAC, 10-173
NoiselessDecoder_LC_AAC, 10-140
NoiseSpectrumUpdate_Aurora, 8-213
Norm, 5-133
Normalize, 5-54
NormalizeColumn, 8-150
NormalizeInRange, 8-151
NormDiff, 5-135
NormEnergy, 8-66
Not, 5-12
notational conventions, 1-7
NthMaxElement, 8-17

O

online version, 1-7
OpenLoopPitchSearch_AMRWB, 9-178
OpenLoopPitchSearch_AMRWBE, 9-220
OpenLoopPitchSearch_G723, 9-98
OpenLoopPitchSearch_G729, 9-43
OpenLoopPitchSearchDTXVAD1_GSMAMR, 9-138
OpenLoopPitchSearchDTXVAD2_GSMAMR, 9-141
OpenLoopPitchSearchNonDTX_GSMAMR, 9-136
Or, 5-9
OrC, 5-8
out-of-order DFT, 7-42
OutProbPreCalc, 8-167

P

PackBits, 4-5
packed format for Fourier transform, 7-5
    CCS format, 7-6
    pack format, 7-5
    perm format, 7-6
PackFrameHeader_MP3, 10-104
PackScalefactors_MP3, 10-99
PackSideInfo_MP3, 10-105
parallelism, 1-1
Periodicity, 8-248
PeriodicityLSPE, 8-247
Phase, complex, 5-72
PhaseDispersion_G729D, 9-82

PhaseDispersionGetStateSize_G729D, 9-80
PhaseDispersionInit_G729D, 9-81
PhaseDispersionUpdate_G729D, 9-81
pitch, 9-43
    close loop, 9-101
    open loop, 9-98
pitch super resolution functions
    CrossCorrCoeff, 8-88
    CrossCorrCoeffDecim, 8-87
    CrossCorrCoeffInterpolation, 8-90
PitchmarkToF0, 8-38
PitchPeriodExtraction_G728, 9-280
PitchPostFilter_G723, 9-116
platforms supported, 1-2
PolarToCart, complex, 5-96
polynomial coefficients, 9-94
Polyphase functions
    ResamplePolyphase, 8-209
    ResamplePolyphaseFree, 8-208
    ResamplePolyphaseGetFilter, 8-203
    ResamplePolyphaseGetSize, 8-201
    ResamplePolyphaseInit, 8-200
    ResamplePolyphaseInitAlloc, 8-206
    ResamplePolyphaseSetFilter, 8-202
post filter, 9-116
PostFilter_G728, 9-277
PostFilter_GSMAMR, 9-167
PostFilter_RTA, 9-341
PostFilterAdapterGetStateSize_G728, 9-278
PostFilterAdapterStateInit_G728, 9-279
PostFilterGetStateSize_G728, 9-276
PostFilterGetStateSize_RTA, 9-340
PostFilterInit_G728, 9-276
PostFilterInit_RTA, 9-341
PostFilterLowBand_AMRWBE, 9-225
Pow, 12-16
Pow34, 10-13
Pow43, 10-16
Pow43Scale, 10-17
power weighting, 9-11
PowerSpectr, complex, 5-73
Powx, 12-19
PredictCoef_SBR, 10-218
prediction error, 9-61
prediction, in frequency domain, 10-38
PredictOneCoef_SBR, 10-220
Preemphasize, 5-98
Preemphasize_AMRWB, 9-185
Preemphasize_G729, 9-83
Preemphasize_GSMAMR, 9-156
Preemphasize_GSMFR, 9-256
prefix, in function names, 1-8
prequantization, 10-13
PreSelect_VQ, 10-64
PsychoacousticModelTwo_MP3, 10-86

Q

QMFDecode_RTA, 9-338
QMFEncode_RTA, 9-339
QMFGetStateSize_RTA, 9-337
QMFInit_RTA, 9-338
QRTransColumn, 8-174
QuantInv_AAC, 10-146
quantization
    in speech codecs, 9-33
    inverse, 9-96
    process, 9-38
Quantize_MP3, 10-94
QuantLSPDecode_GSMAMR, 9-132
QuantTCX_AMRWBE, 9-243

R

RandGauss, 4-39
RandGauss_Direct, 4-40
RandGaussFree, 4-36
RandGaussGetSize, 4-37
RandGaussInit, 4-38
RandGaussInitAlloc, 4-35
RandomNoiseExcitation_G729B, 9-86
RandUniform_Direct, 4-34
RandUniformFree, 4-30
RandUniformGetSize, 4-32
RandUniformInit, 4-31
RandUniformInitAlloc, 4-29
RandUnifrom, 4-32
Real, 5-75
RealToCplx, 5-77
RecSqrt, 8-71
ReflectionToAR, 8-35
ReflectionToLP, 8-34
ReflectionToTilt, 8-36

RegExpFind, 11-30
RegExpFree, 11-27
RegExpGetSize, 11-29
RegExpInit, 11-27
RegExpInitAlloc, 11-26
RegExpSetMatchLimit, 11-29
regular expressions, 11-25
related publications, 1-7
Remove, 11-9
ReplaceC, 11-17
ReQuantize_MP3, 10-115
ResamplePolyphase, 8-209
ResamplePolyphase_AMRWBE, 9-237
ResamplePolyphaseFree, 8-208
ResamplePolyphaseGetFilter, 8-203
ResamplePolyphaseGetSize, 8-201
ResamplePolyphaseInit, 8-200
ResamplePolyphaseInitAlloc, 8-206
ResamplePolyphaseSetFilter, 8-202
residual signal, 9-66
ResidualFilter_AMRWB, 9-172
ResidualFilter_Aurora, 8-217
ResidualFilter_G729, 9-65
RFC1950, 13-50, 13-65, 13-68, 13-71, 13-73
RFC1951, 13-50, 13-59, 13-61, 13-62, 13-65, 13-68, 13-71, 13-73, 13-74, 13-80
RFC1952, 13-50, 13-65, 13-68, 13-71, 13-73
Rice-Colomb coding, 13-93
Rint, 12-70
RLE.See run length encoding
RLEFree_BZ2, 13-114
Round, 12-67
rounding mode, in speech codec functions, 9-2
RPEQuantDecode_GSMFR, 9-249
RShiftC, 5-14
RT Audio
    AdaptiveCodebookSearch_RTA, 9-342
    BandPassFilter_RTA, 9-347
    FixedCodebookSearch_RTA, 9-343
    FixedCodebookSearchRandom_RTA, 9-343
    HighPassFilter_RTA, 9-346
    LevinsonDurbin_RTA, 9-336
    LPCToLSP_RTA, 9-335
    LSPQuant_RTA, 9-345
    LSPToLPC_RTA, 9-335
    PostFilter_RTA, 9-341
    PostFilterGetStateSize_RTA, 9-340
    PostFilterInit_RTA, 9-341
    QMFDecode_RTA, 9-338
    QMFEncode_RTA, 9-339
    QMFGetStateSize_RTA, 9-337
    QMFInit_RTA, 9-338
run length encoding (RLE), 13-110 thru ??
    DecodeRLE, 13-111
    EncodeRLE, 13-110

S

SampleDown, 5-145
SampleUp, 5-143
Sampling functions, 5-143 thru 5-147
    SampleDown, 5-145
    SampleUp, 5-143
Scale, 10-22
scale factor bands, 10-22
scale factors calculation, 10-20
ScaleLM, 8-93
Schur, 8-29
Schur_GSMFR, 9-254
Set, 4-8
SetDenormAreZeros(). See ippSetDenormAreZeros
SetFlushToZero(). See ippSetFlushToZero
SetNumThreads(). See ippSetNumThreads
Shift functions, 5-13
ShortTermAnalysisFilter_GSMFR, 9-251
ShortTermPostFilter_G729, 9-72
ShortTermSynthesisFilter_GSMFR, 9-252
signal name conventions, 1-8
SignChangeRate, 8-24
SIMD instructions, 1-1
Sin, 12-32
SinC, 8-159
SinCos, 12-34
single-rate FIR LMS filter functions
    FIRLMS, 6-76
    FIRLMSFree, 6-72
    FIRLMSGetDlyLine, 6-74
    FIRLMSGetTaps, 6-73
    FIRLMSInitAlloc, 6-71
    FIRLMSOne_Direct, 6-78
    FIRLMSSetDlyLine, 6-75
Sinh, 12-48

sliding window technique, 13-59, 13-71
SmoothedPowerSpectrumAurora, 8-212
SNR_AMRWBE, 9-219
SortAscend, 5-56
SortDescend, 5-56
SortIndexAscend, 5-57
SortIndexDescend, 5-57
special vector functions
    VectorJaehne, 4-41
    VectorRamp, 4-44
    VectorSlope, 4-43
spectral data prequantization, 10-13
spectral values restoration, 10-22
speech codecs common functions
    AutoCorr, 9-14
    AutoCorr_NormE, 9-17
    AutoCorrLagMax, 9-15
    AutoScale, 9-12
    ConvPartial, 9-6
    CrossCorr, 9-18
    CrossCorrLagMax, 9-19
    DotProdAutoScale, 9-13
    InterpolateC_NR, 9-7
    InvSqrt, 9-14
    Mul_NR, 9-8
    MulC_NR, 9-10
    MulPowerC_NR, 9-11
    SynthesisFilter_G723, 9-20
speech, synthesized, 9-67
SplitC, 11-24
SplitScaled, 5-66
SplitVQ, 8-196
Spread, 10-60
Sqr, 5-42
Sqrt, 5-44, 12-9
StaticInit(). See ippStaticInit
StaticInitCpu(). See ippStaticInitCpu
Statistical functions, 5-117 thru 5-140
    CountInRange, 5-142
    DotProd, 5-138
    Max, 5-119
    MaxAbs, 5-121
    MaxAbsIndx, 5-122
    MaxEvery, 5-141
    MaxIndx, 5-120
    Mean, 5-128
    MeanStdDev, 5-132
    Min, 5-123
    MinAbs, 5-125
    MinAbsIndx, 5-125
    MinEvery, 5-141
    MinIndx, 5-124
    MinMax, 5-126
    MinMaxIndx, 5-127
    Norm, 5-133
    Normalize, 5-54
    NormDiff, 5-135
    Phase, complex, 5-72
    PowerSpectr, complex, 5-73
    StdDev, 5-130
    Sum, 5-117
StdDev, 5-130
Streaming SIMD Extensions, 1-1
string functions
    Compare, 11-10
    CompareIgnoreCase, 11-13
    CompareIgnoreCaseLatin, 11-13
    Concat, 11-22
    ConcatC, 11-23
    Equal, 11-14
    Find, FindRev, 11-3
    FindC, FindRevC, 11-5
    FindCAny, FindRevCAny, 11-6
    Hash, 11-21
    Insert, 11-8
    Lowercase, 11-19
    LowercaseLatin, 11-19
    RegExpFind, 11-30
    RegExpFree, 11-27
    RegExpGetSize, 11-29
    RegExpInit, 11-27
    RegExpSetMatchLimit, 11-29
    Remove, 11-9
    ReplaceC, 11-17
    SplitC, 11-24, 11-26
    TrimC, 11-15
    TrimCAny, 11-16
    TrimEndCAny, 11-16
    TrimStartCAny, 11-16
    Uppercase, 11-18
    UppercaseLatin, 11-18
Sub, 5-31

SubbandProcessGetSize, 9-291
SubC, 5-27
SubCRev, 5-29
SubRow, 8-13
Sum, 5-117
SumColumn, 8-10
SumColumnAbs, 8-177
SumColumnSqr, 8-178
SumLn, 5-52
SumMeanVar, 8-68
SumRow, 8-11
SumRowAbs, 8-179
SumRowSqr, 8-180
SVD, SVDSort, 8-181
SwapBytes, 5-58
SyntesisFilter_G728, 9-272
SynthesisDownFilter_SBR, 10-211
SynthesisDownFilterFree_SBR, 10-192
SynthesisDownFilterGetSize_SBR, 10-196
SynthesisDownFilterInit_SBR, 10-201
SynthesisDownFilterInitAlloc_SBR, 10-189
SynthesisFilter_AMRWB, 9-187
SynthesisFilter_AMRWBE, 9-222
SynthesisFilter_DTS, 10-225
SynthesisFilter_G723, 9-20, 9-110
SynthesisFilter_G729, 9-67
SynthesisFilter_PQMF_MP3, 10-123
SynthesisFilter_SBR, 10-207
SynthesisFilterFree_DTS, 10-224
SynthesisFilterFree_PQMF_MP3, 10-122
SynthesisFilterFree_SBR, 10-191
SynthesisFilterGetSize_DTS, 10-224
SynthesisFilterGetSize_PQMF_MP3, 10-122
SynthesisFilterGetSize_SBR, 10-194
SynthesisFilterGetStateSize_G728, 9-271
SynthesisFilterInit_DTS, 10-222
SynthesisFilterInit_G728, 9-272
SynthesisFilterInit_PQMF_MP3, 10-120
SynthesisFilterInitAlloc_DTS, 10-223
SynthesisFilterInitAlloc_PQMF_MP3, 10-121
SynthesisFilterInitAlloc_SBR, 10-188
SynthesisInit_SBR, 10-199
SynthPQMF_MP3, 10-119

T

TabsCalculation_Aurora, 8-217
Tan, 12-36
Tanh, 12-50
Threshold, 5-79
Threshold_GT, 5-82
Threshold_GTAbs, 5-85
Threshold_GTVal, 5-88
Threshold_LT, 5-82
Threshold_LTAbs, 5-85
Threshold_LTInv, 5-92
Threshold_LTVal, 5-88
Threshold_LTValGTVal, 5-88
tilt, 9-74
TiltCompensation_G723, 9-111
TiltCompensation_G729, 9-73
ToeplizMatrix_G723, 9-104
ToeplizMatrix_G729, 9-55
Tone, 4-16
Tone_Direct, 4-17
ToneFree, 4-12
tone-generating function, 4-16
tone-generating functions
    Tone_Direct, 4-17
    ToneFree, 4-12
    ToneGetStateSizeQ15, 4-12
    ToneInitAllocQ15, 4-11
    ToneInitQ15, 4-13
    ToneQ15, 4-14
ToneGetStateSizeQ15, 4-12
ToneInitAllocQ15, 4-11
ToneInitQ15, 4-13
ToneQ15, 4-14
transcendental mathematical functions, 12-1
Transform support functions, 7-7 thru 7-18
Triangle, 4-25
Triangle_Direct, 4-27
TriangleFree, 4-22
triangle-generating function, 4-25
triangle-generating functions
    Triangle_Direct, 4-27
    TriangleFree, 4-22
    TriangleGetStateSizeQ15, 4-22
    TriangleInitAllocQ15, 4-20
    TriangleInitQ15, 4-23
    TriangleQ15, 4-24

TriangleGetStateSizeQ15, 4-22
TriangleInitAllocQ15, 4-20
TriangleInitQ15, 4-23
TriangleQ15, 4-24
TrimC, 11-15
TrimCAny, 11-16
TrimEndCAny, 11-16
TrimStartCAny, 11-16
Trunc, 12-66

U

Understanding windowing functions, 5-106 thru 5-107
uniform distribution functions
    RandUniformFree, 4-30
    RandUniformGetSize, 4-32
    RandUniformInit, 4-31
    RandUniformInitAlloc, 4-29
    RandUnifrom, 4-32
    RandUnifrom_Direct, 4-34
UnitCurve, 8-39
Unpack of packed data, 7-7 thru 7-13
    ConjCcs, 7-11
    ConjPack, 7-7
    ConjPerm, 7-9
UnpackADIFHeader_AAC, 10-136
UnpackADTSFrameHeader_AAC, 10-137
UnpackFrameHeader_MP3, 10-109
UnpackScaleFactors_MP3, 10-111
UnpackSideInfo_MP3, 10-110
UpdateGConst, 8-166
UpdateLinear, 6-10
UpdateMean, 8-163
UpdateNoisePSDMCRA, 8-244
UpdatePathMetricsDV, 5-105
UpdatePower, 6-12
UpdateVar, 8-164
UpdateWeight, 8-165
Uppercase, 11-18
UppercaseLatin, 11-18
Upsample_AMRWBE, 9-231
User filter banks wavelet transforms, 7-91 thru 7-106

V

VAD_AMRWB, 9-189
VAD1_GSMAMR, 9-157
VAD2_GSMAMR, 9-160
VADDecision_Aurora, 8-230
VADFlush_Aurora, 8-231
VADGetBufSize_Aurora, 8-229
VADGetSize_AMRWB, 9-188
VADInit_AMRWB, 9-189
VADInit_Aurora, 8-229
VarColumn, 8-141
Variable Length coding, 13-4
variable length coding table, 13-4
VecMatMul, 8-18
Vector correlation functions
    AutoCorr, 6-6
vector initialization functions
    Copy, 4-3
    Move, 4-6
    PackBits, 4-5
    Set, 4-8
    Zero, 4-9
Vector quantization functions, 8-184 thru 8-199
    CdbkFree, 8-191
    CdbkGetSize, 8-187
    CdbkInit, 8-188
    CdbkInitAlloc, 8-190
    FormVector, 8-184
    FormVectorVQ, 8-198
    GetCdbkSize, 8-192
    GetCodebook, 8-193
    SplitVQ, 8-196
    VQ, 8-194
vector quantizer, 9-38
VectorJaehne, 4-41
VectorRamp, 4-44
VectorSlope, 4-43
version information function, 3-2
Viterbi decoder functions, 5-102 thru 5-105
    BuildSymblTableDV4D, 5-104
    CalcStatesDV, 5-103
    GetVarPointDV, 5-102
    UpdatePathMetricsDV, 5-105
VLC functions, 10-47
VLCCountBits, 13-10
VLCCountEscBits_AAC, 10-56

VLCCountEscBits_MP3, 10-55
VLCDecodeBlock, 13-15
VLCDecodeEscBlock_AAC, 10-50
VLCDecodeEscBlock_MP3, 10-47
VLCDecodeFree, 13-12
VLCDecodeGetSize, 13-14
VLCDecodeInit, 13-12
VLCDecodeInitAlloc, 13-11
VLCDecodeOne, 13-16
VLCDecodeUTupleBlock, 13-22
VLCDecodeUTupleEscBlock_AAC, 10-53
VLCDecodeUTupleEscBlock_MP3, 10-52
VLCDecodeUTupleFree, 13-19
VLCDecodeUTupleGetSize, 13-21
VLCDecodeUTupleInit, 13-20
VLCDecodeUTupleInitAlloc, 13-18
VLCDecodeUTupleOne, 13-23
VLCEncodeBlock, 13-8
VLCEncodeEscBlock_AAC, 10-58
VLCEncodeEscBlock_MP3, 10-57
VLCEncodeFree, 13-6
VLCEncodeGetSize, 13-7
VLCEncodeInit, 13-6
VLCEncodeInitAlloc, 13-5
VLCEncodeOne, 13-9
Voice Activity Detection functions
    FindPeaks, 8-246
    Periodicity, 8-248
    PeriodicityLSPE, 8-247
VQCodeBookFree, 10-63
VQCodeBookGetSize, 10-64
VQCodeBookInit, 10-62
VQCodeBookInitAlloc, 10-61
VQMainSelect, 10-66
VQReconstruction, 10-71
VQSingle_Sort, 8-195
VQSingle_Thresh, 8-195

W

Wavelet transform functions, 7-83 thru 7-106
    WTFwd, 7-95
    WTFwdFree, 7-94
    WTFwdGetDlyLine, 7-98
    WTFwdInitAlloc, 7-91
    WTFwdSetDlyLine, 7-98
    WTHaarFwd, 7-85
    WTHaarInv, 7-85
    WTInv, 7-101
    WTInvFree, 7-94
    WTInvGetDlyLine, 7-104
    WTInvInitAlloc, 7-91
    WTInvSetDlyLine, 7-104
WaveProcessing_Aurora, 8-218
WeightedMeanColumn, 8-144
WeightedMeanVarColumn, 8-147
WeightedSum, 8-182
WeightedVarColumn, 8-145
weighting matrix, 9-97
WeightingFilter_GSMFR, 9-255
WienerFilterDesign_Aurora, 8-214
WinBartlett, 5-107
WinBlackman, 5-109
Windowing functions, 5-106 thru 5-117
    WinBartlett, 5-107
    WinBlackman, 5-109
    WinHamming, 5-112
    WinHann, 5-114
    WinKaiser, 5-115
WinHamming, 5-112
WinHann, 5-114
WinHybrid_G728, 9-284
WinHybrid_G729E, 9-85
WinHybridGetStateSize_G728, 9-282
WinHybridGetStateSize_G729E, 9-84
WinHybridInit_G728, 9-283
WinHybridInit_G729E, 9-84
WinKaiser, 5-115
WTFwd, 7-95
WTFwdFree, 7-94
WTFwdGetDlyLine, 7-98
WTFwdInitAlloc, 7-91
WTFwdSetDlyLine, 7-98
WTHaarFwd, 7-85
WTHaarInv, 7-85
WTInv, 7-101
WTInvFree, 7-94
WTInvGetDlyLine, 7-104
WTInvInitAlloc, 7-91
WTInvSetDlyLine, 7-104

X

Xor, 5-11
XorC, 5-10

Z

Zero, 4-9
ZeroMean, 8-22
ZLIB, 13-50
ZLIB functions, 13-50 thru 13-82
    Adler32, 13-80
    CRC32, 13-80
    DecodeLZ77, 13-71
    DecodeLZ77DynamicHuff, 13-74
    DecodeLZ77FixedHuff, 13-73
    DecodeLZ77GetBlockType, 13-72
    DecodeLZ77GetPairs, 13-76
    DecodeLZ77GetSize, 13-69
    DecodeLZ77GetStatus, 13-78
    DecodeLZ77Init, 13-69
    DecodeLZ77InitAlloc, 13-70
    DecodeLZ77Reset, 13-79
    DecodeLZ77SetPairs, 13-77
    DecodeLZ77SetStatus, 13-78
    DecodeLZ77StoredBlock, 13-75
    EncodeLZ77, 13-59
    EncodeLZ77DynamicHuff, 13-62
    EncodeLZ77FixedHuf, 13-61
    EncodeLZ77Flush, 13-64
    EncodeLZ77Free, 13-58
    EncodeLZ77GetPairs, 13-65
    EncodeLZ77GetSize, 13-57
    EncodeLZ77GetStatus, 13-67
    EncodeLZ77Init, 13-56
    EncodeLZ77InitAlloc, 13-57
    EncodeLZ77Reset, 13-68
    EncodeLZ77SelectHuffMode, 13-60
    EncodeLZ77SetPairs, 13-66
    EncodeLZ77SetStatus, 13-67
    EncodeLZ77StoredBlock, 13-63