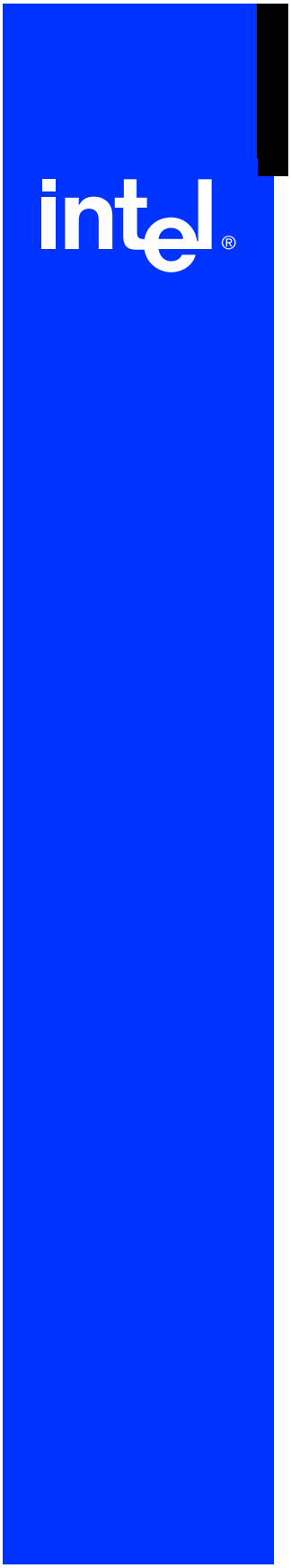


Intel® Integrated Performance Primitives on Intel® Personal Internet Client Architecture Processors

Reference Manual

Version 4.1

Issued in U.S.A.
Document Number: 278378-024

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

Version	Version Information	Date
-019		09/2003
-020		11/2003
-021		11/2003
-022		01/2004
-023	Update manual for version 4.1 Beta	06/2004
-024	Update manual for version 4.1	07/2004

This document describes the Intel® Integrated Performance Primitives (Intel® IPP) software libraries for the Intel®PCA Processors with Intel® Wireless MMX™ Technology. It contains a description of the Intel® IPP software modules, their functionality, their APIs, and other general information pertaining to the form, structure, and use of the Intel® IPP library. The exclusive intent of this document is to disseminate among software developers reference information on IPP.

MPEG is an international standard compression/decompression of digital audio and digital video promoted by ISO. Implementations of MPEG CODECs, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

Contact your local Intel sales office or your distributor to obtain the latest specifications.

Copies of documents which have an ordering 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 website at http://www.intel.com.

Contents

Chapter 1

Introduction

List of Tables

List of Figures

Introduction 1

The Intel® Integrated Performance Primitives (Intel® IPP) provide a rich and powerful set of general and multimedia signal processing functions optimized for the PAX25x and PAX26x family of processors, and Intel® PCA Processors with Intel® Wireless MMX™ Technology (PCA processors with MMX™), which is compliant with the ARM Architecture V5TE. This Reference Manual and the associated software libraries comprise version 4.1 for the Intel® IPP.

NOTE. *Refer to the Release Notes provided with the Intel® IPP software kit for the specific processors supported by the Intel® IPP.*

Related Publications

For the latest Intel® IPP information and updates, please refer to the release notes that are packaged with the Intel® IPP software kit. For related information on the Intel® IPP, refer to the documents listed in your developer's kit's user's guide. Finally, for a list of the supplementary texts, telecommunication standards, and multimedia signal processing standards that are referenced throughout this document, please consult the bibliography in Appendix B.

About This Software

The Intel® IPP library comprises a rich and powerful set of general and multimedia signal processing kernels optimized for maximum performance on the Intel® IPP. The Intel® IPP offers application developers a number of significant advantages:

- The primitives are optimized for maximum performance on the Intel® IPP. Therefore, Intel® IPP enables migration to the Intel® IPP of many computationally intensive applications that have traditionally required special-purpose DSP hardware.

- The Intel® IPP drastically reduces development costs and profoundly accelerates time-to-market by eliminating the need for hand optimization of routines commonly used for multimedia signal processing. Developers are provided with an "off-the-shelf" optimized solution.
- The Intel® IPP is compatible with popular real-time embedded operating systems that run on the Intel® IPP. The primitives are low-level and have been carefully designed to avoid host operating system (OS) dependencies.
- The Intel® IPP supports application porting across certain Intel platforms. A consistent Application Programming Interface (API) is defined for the Intel® IPP that is supported by Intel® IPP, and therefore applications can be ported easily without sacrificing performance.

Feature Summary

The Intel® IPP includes general signal and image processing primitives optimized for the Intel® IPP, as well as primitives that can be used to construct internationally standardized audio, video, image, and speech encoder/decoders (CODECs) for the Intel® IPP.

Primitives available for general one-dimensional (1D) signal processing include the following:

- vector initialization, arithmetic, statistics, thresholding, and measure
- deterministic and random signal generation
- convolution, filtering, windowing, and transforms

And primitives for general two-dimensional (2D) image processing include the following:

- vector initialization, arithmetic, statistics, thresholding, and measure
- color conversions
- morphological operations
- convolution, filtering, windowing, and transforms

Additional primitives are available that allow construction of the following multimedia CODECs:

- video – ITU H.263 decoder, ISO/IEC 14496-2 MPEG-4 decoder, MPEG-4 encoder
- audio – ISO/IEC 11172-3 and 13818-3 (MPEG-1, -2) Layer 3 ("MP3") decoder, MP3 encoder, AAC decoder
- speech – ITU-T G.723.1 CODEC and ETSI GSM-AMR CODEC
- image – ISO/IEC JPEG CODEC, JPEG 2000 CODEC

Cryptographic Primitives provide a rich and powerful set of cryptographic primitive functions:

- Block cipher operations for DES/TDES, AES/Rijndael, Blowfish, and Twofish
- Data authentication algorithms for DES/TDES, Rijndaell128/192/256, Blowfish, and Twofish
- One-way has function operations for SHA-1, SHA-256/384/512, and MD5
- Keyed-hash message authentication code for SHA-1, SHA-56/384/SHA512 and MD5

- Public key cryptographic operations including:
  - Big number arithmetic operation/Modulo reduction, inversion, exponentiation
  - Pseudo-random number generation
  - Probable prime number generation
  - RSA key generation, encryption and decryption
  - DSA key generation, signing, and verification

Hardware Requirements

Intel® IPP libraries are compatible with the Intel® PXA27x Processor Developer's Kit (kit).

- host machine: An Intel® Pentium® IV processor 1.5 GHz based PC with 512 MB RAM or better
- target platform:
  - Platforms that support Intel® PCA Processors with Intel® Wireless MMX™ Technology
  - Intel® DBPXA250, DBPXA255, DBPXA262, or DBPXA263 Development Platform

Software Requirements

The Intel® IPP libraries for the Intel® IPP are compatible with the following operating systems and development environments:

- Microsoft Embedded Visual C++ 4.0 with Service Pack 2 and Pocket PC* 2003 SDK for Microsoft Windows Mobile* 2003-based Pocket PC devices; Microsoft Embedded Visual C++ 4.0 with Service Pack 2 and Smartphone* 2003 SDK for Microsoft Windows Mobile* 2003-based Smartphone devices
- Microsoft Windows Mobile* 2003 software

About This Manual

This manual describes the functions that comprise the Intel® Integrated Performance Primitives (Intel® IPP) for the Intel® IPP. For the purposes of discussion, the functions have been organized according to the type of operation they perform as well as by the data types on which they operate. Each function is introduced by its name. This is followed by the function prototype, definitions of its arguments, and a one-line description of its purpose. Beyond the one-line description, block diagrams and/or equations are given, whenever appropriate, to capture the fine details of the operation associated with a particular primitive.

Intended Audience

This manual assumes that the reader is a skilled programmer having some working knowledge of basic signal processing vocabulary and principles. In particular, the manual presentation is geared towards developers who have had practical experience or theoretical exposure to one or more of the following areas or applications:

- Digital Signal Processing (DSP)
- image processing
- speech recognition
- speech coding
- audio coding
- image coding
- video coding
- cryptography

NOTE. *A bibliography containing several tutorial references is provided in* Appendix B, "Bibliography.".

Manual Organization

This manual is organized as follows:

Chapter 1, "Introduction." Provides background and introductory information on the Intel® IPP libraries. The focus is on the essential Intel® IPP API elements, including header files, binary files, and data types. The remainder of this chapter defines Intel® IPP notation conventions and provides an API summary for Intel® IPP.

Chapter 2, "Vector Initialization, Arithmetic, Thresholding, and Statistics." Gives details on the vector primitives that are provided for initialization, arithmetic, statistics, thresholding, and measure.

Chapter 3, "Signal Generation." Provides information on primitives that offer deterministic and pseudorandom sequence generation. Several primitives are available for synthesizing deterministic signals, including both sinusoidal and triangular sequences. For pseudorandom data streams, primitives that produce both uniformly and Gaussian distributed samples are available.

Chapter 4, "Filtering." Provides information on the digital filtering primitives, including those available for Finite Impulse Response (FIR), Infinite Impulse Response (IIR), cascaded biquad IIR, and Least-Mean-Square (LMS) adaptive FIR filters.

Chapter 5, "Windowing." Provides information on non-parametric and parametric windowing primitives.

Chapter 6, "Convolution." Describes available discrete convolution primitives, including primitives that can process both 1D and 2D signals. Discrete convolution is the operation that defines the output sequence generated by a discrete-time Linear Time-Invariant (LTI) system in response to an arbitrary input sequence.

Chapter 7, "Transforms." Provides information on primitives that implement discrete transforms. Primitives are available for calculating both forward and inverse discrete Fourier transforms using the Fast Fourier Transform (FFT) algorithm with radix-2 block sizes.

Chapter 8, "MP3 Audio Decoder." Provides information on primitives that can be used to implement audio decoders such as those recommended in the ISO/IEC 11172-3 MPEG-1 standard, such as., MPEG-1, layer 3 ("MP3"). Application developers can use the primitives described in this chapter as building blocks for a performance-optimized custom audio decoder.

Chapter 9, "MP3 Audio Encoder." Provides information on the primitives that can be used to implement audio encoders such as those recommended by *ISO/IEC 11172-3:1993, Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5Mbit/s-----Part 3: Audio*. The API enables customers to develop audio encoders based on the Intel XScale® microarchitecture.

Chapter 10, "Advanced Audio Coding." Provides information on the advanced audio coding (AAC) decoder for Intel XScale® microarchitecture, including a complete definition of the function calls and data structures that comprise the API. The AAC decoder API provides a variety of decoder functions, including bit stream unpacking and AAC core decoding functions. This provides customers great flexibility in configuring the decoder system.

Chapter 11, "H.263 Video Decoder and Processing." Provides information on primitives that implement the video decoder recommended in the ITU H.263 standard. Developers can use the primitives described in this chapter to build a performance-optimized custom video decoder.

Chapter 12, "MPEG-4 Video Decoder." Provides information on the primitives that implement the video decoder recommended in the ISO/IEC 14496-2 MPEG-4 video standard. Developers can use the primitives described in this chapter to build a performance-optimized custom video decoder.

Chapter 13, "MPEG-4 Video Encoder." Provides information on the primitives that implement the ISO/IEC 14496-2 MPEG-4 video encoder. MPEG-4 is a widely used coding method for video signals in various applications such as digital storage media, internet, various forms of wired or wireless communication, etc.

Chapter 14, "GSM-AMR Speech CODEC." Provides information to implement the speech CODEC recommended in the ETSI GSM-AMR standard. Developers use the primitives described in this chapter to build performance-optimized customized speech code.

Chapter 15, "G.723.1 Speech CODEC." Provides information on primitives that implement the speech CODEC recommended in the ITU-T G.723.1 standard. Developers use the primitives described in this chapter to build performance-optimized customized speech code.

Chapter 16, "Image Processing." Provides information on primitives that are used for general image processing such as filtering, linear transformations, color space conversions, and morphing.

Chapter 17, "JPEG Image CODEC." Provides information on primitives that implement the image decoder recommended in the ISO/IEC 10918 JPEG standard. Developers use the primitives described in this chapter to build performance-optimized customized speech code.

Chapter 18, "JPEG 2000 Image CODEC." Provides information on the primitives that implement the JPEG 2000 image processing standard is described in *ISO/IEC 15444-1. Information technology – JPEG 2000 image coding system – Part 1: Core coding system*.

Chapter 19, "Cryptography." Provides information on primitives that implement various FIPS published block ciphers, one-way hash functions, as well as integer modulus based public key cryptographic systems.

Chapter 20, "Audio Toolkit API." Provides information on the Audio Toolkit API for the Intel® Integrated Performance Primitives (Intel® IPP). This includes architecture, data structures, and primitives for: Ephraim-Malay Noise Suppressor (EMNS), Acoustic Echo Canceller (AEC), Voice Activity Detector (VAD), and the speech recognition Feature Extractor (FE) and Feature Encoder.

Chapter 21, "H.264 Video Decoder." Provides information on primitives that implement the video decoder recommended in the ITU-T Rec. H.264/ ISO/IEC 14496-10 Advanced Video Coding standard. Developers use the primitives described in this chapter to build a performance-optimized custom video decoder.

Appendix A, "Acronyms." Defines the acronyms that appear in this book.

Appendix B, "Bibliography." Provides a list of the supplementary texts, telecommunication standards, and multimedia signal processing standards that are referenced throughout this chapter.

Appendix C, "extools.h and extrool.c files." Includes the extrools.h header file and the extools.c file that are referred to in other chapters of this manual.

"Index." This index section provides an alphabetic list of all Intel® IPP names and the page numbers where they can be found.

Notations and Conventions

The following conventions are observed in this book:

- Parameters appear in *`italic courier`* type.
- Written code appears in `plain courier` type.
- File names and function names to be defined are delimited by <>.

Intel® Integrated Performance Primitives Library Interface

This section describes the Intel® IPP library interface, which is comprised of the following elements:

- binary image files
- header files
- macros and constants
- API/function prototypes
- data types
- integer scaling conventions
- error handling and status flags
- Qm.n format and conventions

Binary Image Files

The primitives are contained in a collection of static binary libraries, organized by function type, against which users must link their application object files. Two versions of the image files are included in the release: a debug version and a release version. The defining difference between the release and debug versions lies in the input argument error/bounds checking. In the debug library build, each function validates its input parameters, and returns corresponding error messages whenever appropriate. The release version maximizes performance by eliminating the overhead associated with parameter error checking. It is strongly recommended that users link against the debug library image during application development and debugging cycles, and then link against the release image only to produce a final release candidate for their application.

Table 1-1 and Table 1-2 summarize the library binary files against which application object files should be linked when using particular primitives.

Table 1-1 Intel® IPP Binary Image Files for Windows* on Intel® PCA Processors

Function Types	Pocket PC or Smartphone
Signal Processing (Chapters 2-7)	ippSP_XSC41PPC_r.lib ippSP_XSC41PPC_d.lib
Audio CODEC (Chapters 8-10)	ippAC_XSC41PPC_r.lib ippAC_XSC41PPC_d.lib
Video CODEC (Chapters 11-13)	ippVC_XSC41PPC_r.lib ippVC_XSC41PPC_d.lib
Speech CODEC (Chapters 14 and 15)	ippSC_XSC41PPC_r.lib ippSC_XSC41PPC_d.lib
Image Processing (Chapter 16)	ippIP_XSC41PPC_r.lib ippIP_XSC41PPC_d.lib
JPEG CODEC (Chapters 17 and 18)	ippJP_XSC41PPC_r.lib ippJP_XSC41PPC_d.lib
Cryptographic (Chapter 19)	ippCP_XSC41PPC_r.lib ippCP_XSC41PPC_d.lib
Audio Toolkit API (Chapter 20)	ippSR_XSC41PPC_r.lib ippSR_XSC41PPC_d.lib

Table 1-2 Intel® IPP Binary Image Files for Windows* on Intel® PCA Processors with Intel® Wireless MMX™ Technology

Function Types	Pocket PC or Smartphone
Signal Processing (Chapters 2-7)	ippSP_WMMX41PPC_r.lib
	ippSP_WMMX41PPC_d.lib
Audio CODEC (Chapters 8-10)	ippAC_WMMX41PPC_r.lib
	ippAC_WMMX41PPC_d.lib
Video CODEC (Chapters 11-13)	ippVC_WMMX41PPC_r.lib
	ippVC_WMMX41PPC_d.lib
Speech CODEC (Chapters 14 and 15)	ippSC_WMMX41PPC_r.lib
	ippSC_WMMX41PPC_d.lib
Image Processing (Chapter 16)	ippIP_WMMX41PPC_r.lib
	ippIP_WMMX41PPC_d.lib
JPEG CODEC (Chapters 17 and 18)	ippJP_WMMX41PPC_r.lib
	ippJP_WMMX41PPC_d.lib
Cryptographic (Chapter 19)	ippCP_WMMX41PPC_r.lib
	ippCP_WMMX41PPC_d.lib
Audio Toolkit API (Chapter 20)	ippSR_WMMX41PPC_r.lib
	ippSR_WMMX41PPC_d.lib

Header Files

In order to maximize ease of use and portability, the prototypes for the complete set of Intel® IPP functions are defined in a set of ANSI-compliant 'C' language header files.

- To link against any of the primitives contained in the Intel® IPP binary library image, the application developer must include in the application source code the Intel® IPP header file **<ippdefs.h>**. Then, the application source code must also include additional header files, the precise choice of which depends on which particular primitives are being used.
- To link against the vector manipulation and general DSP primitives described in Chapters 2 through 7, the application source code must include the Intel® IPP header file **<ippSP.h>**.
- To link against the audio coding primitives described in Chapters 8 through 10, the application source code must include the Intel® IPP header file **<ippAC.h>**.
- To link against the video coding primitives described in Chapters 11 through 13 the application source code must include the Intel® IPP header file **<ippVC.h>**.
- To link against the speech coding primitives described in Chapters 14 and 15, the application source code must include the Intel® IPP header file **<ippSC.h>**.

- To link the image processing primitives that are described in Chapter 16, the application source code must include the Intel® IPP header file **<ippIP.h>**.
- To link the JPEG CODEC primitives that are described in Chapter 17 and the JPEG 2000 CODEC primitives that are described in Chapter 19, the application source code must include the Intel® IPP header file **<ippJP.h>**.
- To link the cryptographic primitives that are described in Chapter 19, the application source code must include the Intel® IPP header file **<ippCP.h>**.
- To link to the speech recognition API primitives that are described in Chapter 20 the application source code must include the Intel® IPP header files **<ippSR.h>** and **<ippSP.h>**.

Macros are defined symbolic names that are substituted with particular replacement text at compilation or assembly time. The Intel® IPP API makes available for developers several useful macros and constants (for example, π, 2π, min(a,b), max(a,b), etc.). Developers should refer to the `<*.h>` header file for the most up-to-date list of available macros and constants.

Function Prototypes

The Intel® IPP API is comprised of 'C' language function prototypes for each primitive. The function prototype naming scheme adheres to the following format:

`ipp`*<domain><operation>_<function-specific modifier>_<datatype>_<data modifier> (parameter list)*

where the <> delimited fields are defined as follows:

<domain> - a single character that expresses the subset of functionality to which a given function belongs. All of the primitives described in this manual belong to the Intel® IPP domain `<s>`, where `<s>` denotes "signal processing."

<operation> - an abbreviated descriptor that encapsulates the function behavior. For example, "`FIR`".

<function-specific modifier> - a short mnemonic string that augments the operation descriptor; used typically when the operation name is imprecise. For example, consider the primitive `ipps_Threshold_LT_16s(...)`. The operation "Threshold" is a generic operation has several valid interpretations. The function-specific modifier "LT" informs the user of the particular type of threshold that is applied by this primitive, namely a "less than" threshold.

<data type> - Specifies bit depth and/or data layout using a string of the form:

`#<u|s>[c]`

where: The "#" symbol is replaced by an integer that indicates the bit depth, either of the symbols "u" or "s" is included to denote, respectively, "unsigned integer" or "signed integer", and the optional symbol "c" denotes complex data. For the primitives described in this manual, the "#" symbol is replaced by one of the following bit depth indicators: 8, 16, 32, or 64.

For example, the following primitive operates exclusively on a single data type:

```
ippsAdd_16s(Ipp16s *pSrc1, Ipp16s *pSrc2, Ipp16s *pDst, int len)
```

The data type is specified by the suffix "_16s," which implies that both the input and output operands are represented by 16-bit signed integers (Ipp16s).

For functions that operate on more than one data type, the source data type is listed first, followed by destination data type.

<*data modifier*> - The data modifier further describes the data associated with the operation. It may contain implied parameters and/or indicate additional required parameters. The set of Intel® IPP data modifiers is given in the list below. Data modifiers are always presented in alphabetical order.

- D1 - one-dimensional signal (default)
- D2 - two-dimensional signal
- I - in-place operation
- Sfs - Saturated fixed scale operation

Function Arguments

The Intel® IPP API convention for function argument lists can be expressed generally as follows:

```
<input>, <input data length>, <output>, <output data length>,
<parameter list>
```

Whenever an input or output argument is a scalar rather than a vector (non-array), the associated data length argument is eliminated, as in the case of the following example primitive that computes the standard deviation of a vector:

```
ippsStdDev_16s(Ipp16s * pSrc, int len, Ipp16s * pResult)
```

Whenever the input and output vectors have the same length, the input vector length argument will be eliminated. For example, consider the following primitive for pointwise vector addition:

```
ippsAdd_16s(Ipp16s * pSrc1, Ipp16s * pSrc2, Ipp16s * pDst, int len)
```

Data Types

Table 1-3 shows the collection of pure integer data types. All input and output arguments defined in the API are either integer types, pointers to integer types, or data structures that contain integer types. Both real and complex data structures have been defined. Complex vectors are constructed by interleaving the real and imaginary components.

Table 1-3 Intel® IPP Data Types

Data Type	Corresponding Data Type in C	Corresponding Data Type in ARM Assembly
`Ipp8u`	8-bit unsigned integer, That is, unsigned char	Byte
`Ipp8s`	8-bit signed integer, That is, char	-
`Ipp16u`	16-bit unsigned integer, That is, unsigned short, unsigned short int	-
`Ipp16s`	16-bit integer, That is, short, short int, signed short int	Signed Halfword
`Ipp32u`	32-bit unsigned integer, That is, unsigned int, unsigned long, unsigned long int	-
`Ipp32s`	32-bit signed integer, That is, int, long, long int, signed long int	Signed Word
`Ipp64u`	64-bit unsigned integer	-
`Ipp64s`	64-bit signed integer	Signed Double Word
`Ipp8sc`	struct { Ipp8s Re; Ipp8s Im; }, That is, real/imaginary interleaved complex	-
`Ipp16sc`	struct { Ipp16s Re; Ipp16s Im; }, That is, real/imaginary interleaved complex	-
`Ipp32sc`	struct { Ipp32s Re; Ipp32s Im; }, That is, real/imaginary interleaved complex	-
`Ipp64sc`	struct { Ipp64s Re; Ipp64s Im; }, That is, real/imaginary interleaved complex	-
`IppStatus`	32-bit signed integer, That is, int	Signed Word

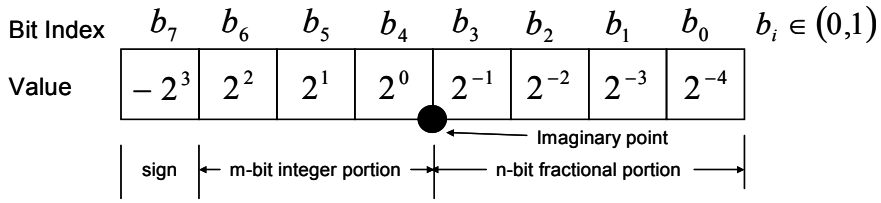
Qm.n Format

Several of the primitives require a fractional interpretation of the integer input and/or output arguments. The "Qm.n" format provides a standard mechanism for representing fractional values using an integer data type. For example, a Q3.4 word is illustrated in Figure 1-1. The integer binary word has been partitioned using an imaginary fixed point. The n-bits to the right of the imaginary point comprise the fractional portion of the value being represented, and these n-bits act as weights for negative powers of 2. The m-bits to the left of the imaginary point comprise the integer portion of the value being represented, and these m-bits act as weights for positive powers of 2. The overall signed Qm.n representation requires a total of m+n+1 bits, with the additional bit required for the sign. As shown in Figure 1-1, the Q3.4 word requires a total of 3+4+1 = 8 bits. The dynamic range for the Q3.4 word spans the open interval [-8, 8), and the precision (or "quantization error") is 1/16. The value represented by a particular set of Q3.4 bits is given by the adding up the weighted powers of 2:

$$value = -b_7 2^3 + b_6 2^2 + \ldots + b_o 2^{-4}$$

where $b_i \in \{0, 1\}$ are the binary bits.

Figure 1-1 Q3.4 Format



In general, the m+n+1-bit Qm.n word can be represented as shown in Figure 1-2. In the figure, each bit cell has the value indicated by a power of 2, and the Qm.n word value is determined by adding together the individual bit cell values weighted by the bits, b_i, where $b_i \in \{0, 1\}, 0 \le i \le m+n$, That is:

$$value = -b_{m+n}2^{-m} + \sum_{i=1}^{m+n} b_{m+n-i}2^{m-i}$$

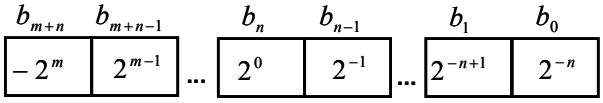
The parameter *m* (number of bits to the left of the point) determines the dynamic range:

$$range = [-2^m, 2^m)$$

and the parameter *n* (number of bits to the right of the point) determines the precision:

$$precision = 2^{-n}$$

Figure 1-2 Qm.n Format



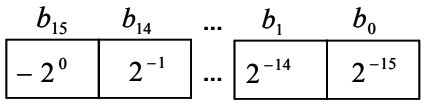
Qm.n Conventions

Two conventions define the use of the Qm.n format in the particular case of the Intel® IPP API. First, all Intel® IPP documentation (this manual, release notes, readme files, etc.) employs the abbreviated notation "Qn" to denote "Qm.n," where m=L-n-1, and L is the word length, in bits, of the underlying data type. In other words, a particular value for m is implied by the combination of a data type and the particular choice of n specified by "Qn." The second naming convention is that all primitives containing arguments interpreted as Qm.n are prototyped such that the suffix "Qn" is appended to the prototype argument names. For the purposes of illustration, two detailed examples are given next.

Example 1: Q0.15, `Ipp16s`

Consider a Q0.15 parameter with the underlying data type of `Ipp16s`. In this case, L=16, and therefore a Q0.15 interpretation is as shown in Figure 1-3.

Figure 1-3 Q0.15 Example with `Ipp16s`



The value, dynamic range, and precision can be obtained easily using the expressions given for a general Qm.n. After substituting the values m=0, n=15, it can be seen that the value, range, and precision, respectively, are given by

$$value = -b_{15}2^0 + \sum_{i=1}^{15} b_{15-i}2^{-i}$$

$$range = [-1,1)$$

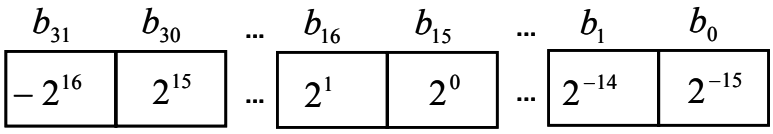
$$precision = 2^{-15}$$

An example primitive that makes use of Q0.15 in combination with `Ipp16s` is the FIR filter.

Example 2: Q16.15, `Ipp32s`

Next, consider a Q16.15 parameter with the underlying data type of `Ipp32s`. In this case, L=32, and therefore a Q16.15 interpretation is as shown in Figure 1-4.

Figure 1-4 Q16.15 Example with `Ipp32s`



The value, dynamic range, and precision can be obtained easily using the expressions given for a general Qm.n. After substituting the values m=16 and n=15, it can be seen that the value, range, and precision, respectively, are given by

$$value = -b_{31}2^{16} + \sum_{i=1}^{31} b_{31-i}2^{16-i}$$

$$range = [-65536, 65536)$$

$$precision = 2^{-15}$$

Examples of primitives that make use of the Q16.15 interpretation in combination with `Ipp32s` are the LMS adaptive FIR filter and the parametric Kaiser window.

Integer Scaling Conventions

The Intel® IPP API includes a scaling mechanism to achieve the maximum possible precision for fixed-point integer operations. Many primitives perform internal computation using a precision higher than the data types that are used for the input and output arguments. This higher precision could be `int`, `long`, or `long long (64-bit integer)`, depending on the implementation and precision requirements. Therefore, it may be necessary to scale the function output arguments to achieve a desired precision in the result. The Intel® IPP library provides saturated fixed scaling as a mechanism that allows users to control the precision of output arguments. In functions that use it, saturated fixed scaling (Sfs) is controlled by the input argument, "scalefactor." For Sfs functions, the output will be multiplied by $2^{-scalefactor}$ before returning to the user. In other words, for functions that internally accumulate results having precision higher than the input and output arguments, it is possible for the user to control which subset of the most significant bits is returned.

A typical function with scaled output has the following format:

```
ippsFunction_Sfs(..., int scaleFactor)
```

In all cases, the mnemonic "Sfs" denotes "Saturated fixed scaling."

The Sfs-enabled primitive performs the required calculation using an internal data type having a higher precision (larger number of bits) than the input and/or output arguments. Once the computation has been completed, the internal high-precision result is shifted by the number of bits indicated in the scale factor (positive scalefactors correspond to right shifts, negative scalefactors to left shifts) and copied into the low-precision output variable.

The user must choose carefully the scalefactor such that significant bits are not truncated from the scaled result. For example, a scaled vector multiply (`ippsMul_16s_Sfs()`) using two 16-bit input operands (`Ipp16s`) could potentially produce a result having 32 significant bits. For this primitive, the internal accumulator actually contains 32 bits. By supplying a scalefactor, the user is able to choose any 16 out of the available 32 result bits. If the top 16 bits of the 32-bit result were needed, then the user would set the scalefactor to 16. On the other hand, if the dynamic range of the input data was constrained such that only 24 significant bits were contained in the internal multiplication result, then the user would select a scalefactor of 8, which would mean that bits 8 through 23 (assuming that bit indices start from 0) were returned in the 16-bit output argument. Clearly, when choosing a scalefactor, the user should consider the dynamic range of the data to avoid the loss of the most significant result bits. Some detailed examples are given for scaled functions later in the manual.

Note that Intel® IPP includes an overflow saturation mechanism. Upon overflow, non-scaled Intel® IPP integer output arguments will saturate to the maximum possible absolute value. For example, upon overflow, a non-scaled output argument of the type `Ipp16s` will saturate to 0x8000 (-32768) for a negative overflow or 0x7fff (32767) for a positive overflow.

Primitive Variables

To maximize flexibility, and ease of use the Intel® IPP API offers up to four variables on each primitive:

- Basic or default
- In-place (I)
- Saturation fixed scale (Sfs)
- In-place and saturation fixed scale (ISfs)

For in-place primitive variables, input and output vectors share common memory. As a result, the contents of the input vector are replaced by contents of the output vector upon return from the primitive. For non-in-place variables, input and output vectors use distinct memory blocks, and therefore the input vector remains unmodified upon return from the primitive call. As described previously, saturation fixed scale primitive variables return outputs that have been scaled by $2^{-scaleFactor}$. That is, output values have been shifted `scaleFactor` samples to the left or right for negative or positive `scaleFactor` values, respectively. Non-Sfs primitive variables can be

viewed as a special case of the Sfs variables in which `scaleFactor` has been set to 0. The ISfs primitive variables combine in-place and saturation fixed scale behavior with the underlying default primitive functionality.

In the interest of clarity and simplicity, the block diagrams, equations, and other detailed behavioral descriptions given throughout the remainder of this manual apply only to the non-in-place and non-scaled (so-called "default") primitive variables, unless explicitly otherwise noted. The user should be aware that the behavior of the scaled and in-place variables can be understood easily by applying the generic in-place and scaled function behavioral rules given above to the default behavioral specification.

Error Handling

The debug versions of the Intel® IPP libraries provide error handling facilities that monitor and report on bad arguments (for example, `NULL` pointers, out-of-range parameter values) and "out of memory" conditions. An appropriate status code is returned upon detection of an error condition. For each function, a list of possible status codes is given under the detailed function descriptions in the remaining chapters of this manual.

The release versions of the Intel® IPP libraries do not provide error handling facilities. Users are responsible for ensuring that all primitives are used correctly. The benefit of disabled error handling is a guarantee of maximum application performance. It is strongly recommended that debug library builds be used during application debug and development cycles. Only when building the final release of an application should developers link against the release version of the Intel® IPP libraries.

Return Codes

IPP functions return status codes 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 an error. The last value of the error status is not stored, and the user decides to check it or not as the function returns. The status codes are of IppStatus type and are global constant integers. The status codes and corresponding messages reported by the Intel® IPP for image processing are listed in Table 1-4.

Table 1-4 IPP Status Code and Messages

Symbolic Status	Associated String
ippStsAlphaTypeErr	Illegal type of image composition operation
ippStsAnchorErr	The anchor point is outside mask
ippStsBadArgErr	Bad Arguments
ippStsBadModulusErr	Bad modulus caused a module inversion failure
ippStsChannelErr	Illegal channel number
ippStsCoeffErr	Non-allowable values of the transformation coefficients
ippStsCOIErr	COI is out of range
ippStsContextMatchErr	The context parameter doesn't match to the operation
ippStsDataTypeErr	Bad or unsupported data type
ippStsDitherLevelsErr	Number of dithering levels is out of range
ippStsDivByZero	Zero value(s) of divisor in the function Div
ippStsDivByZeroErr	An attempt to divide by zero
ippStsDivisorErr	Divisor is equal zero, function is aborted
ippStsDlyLineIndexErr	Invalid value of the delay line sample index
ippStsEpsValErr	Negative epsilon value error
ippStsEvenMedianMaskSize	Even size of Median Filter mask was replaced by odd one
ippStsFftFlagErr	Invalid value of the FFT flag parameter
ippStsFftOrderErr	Invalid value of the FFT order parameter
ippStsFIRLenErr	The number iterations of FIR is less or equal zero
ippStsFIRMRFactorErr	Wrong value of the MR FIR sampling factor parameter
ippStsFIRMRPhaseErr	Wrong value of the MR FIR sampling phase parameter
ippStsGammaRangeErr	Gamma range bounds is less or equal zero
ippStsGrayCoefSumErr	Sum of the ToGray conversion coefficients must be less or equal to 1
ippStsHugeWinErr	Kaiser window is too huge
ippStsIIROrderErr	The order of IIR is less or equal zero
ippStsInSufficientEntropy	Insufficient entropy in the random seed and stimulus bit string caused the prime/key generation to fail

continued

Table 1-4 IPP Status Code and Messages (continued)

Symbolic Status	Associated String
ippStsInterpolationErr	Invalid interpolation mode
ippStsInvalidCryptoKeyErr	A compromised key causes suspension of requested cryptographic operation
ippStsInvZero	INF result. Zero value was met by InvThresh with zero level
ippStsJaehneErr	Magnitude value is negative
ippStsLengthErr	Wrong value of string length
ippStsLnNegArg	Negative value(s) of argument in the Ln function
ippStsLnZeroArg	Zero value<ippSP.h>(s) of argument in the Ln function
ippStsMaskSizeErr	Invalid mask size
ippStsMemAllocErr	Not enough memory for the operation
ippStsMirrorFlipErr	Invalid flip mode
ippStsMisalignedBuf	Misaligned pointer in operation in which must be aligned
ippStsMoment00ZeroErr	Moment value M(0,0) is too small to continue calculation
ippStsNanArg	Not a Number argument value warning
ippStsNoErr	No error
ippStsNoiseValErr	Bad value of noise amplitude for dithering
ippStsNoOperation	No operation has been executed
ippStsNotSupported ModeErr	The requested mode is currently not supported
ippStsNullPtrErr	Null pointer error
ippStsNumChannelsErr	Bad or unsupported number of channels
ippStsOutOfRangeErr	Argument is out of range or point is outside the image
ippStsOverFlowErr	Overflow of result
ippStsQuadErr	The quadrangle degenerates into triangle, line or point
ippStsRectErr	Size of the rectangle is less or equal to the one
ippStsRelFreqErr	Relative frequency value is out of range
ippStsResizeFactorErr	Resize factor(s) less or equal to zero
ippStsSampleFactorErr	Sampling factor is less or equal zero
ippStsSamplePhaseErr	Phase value is out of range: 0 <= phase < factor
ippStsScaleRangeErr	Scale bounds is out of range
ippStsShiftErr	Value of shift is less than zero

continued

Table 1-4 IPP Status Code and Messages (continued)

Symbolic Status	Associated String
ippStsSizeErr	Wrong value of data size
ippStsSqrtNegArg	Negative value(s) of argument in the function Sqrt
ippStsStepErr	Step value is less or equal zero
ippStsStrideErr	The stride value is less than the row length
ippStsThreshNegLevelErr	Negative value of the level in the threshold operation
ippStsThresholdErr	Invalid threshold bounds
ippStsToneFreqErr	The frequency of the Tone is less 0 or greater or equal 0.5
ippStsToneMagnErr	The magnitude of the Tone is less or equal to zero
ippStsTonePhaseErr	The phase of the Tone is less 0 or greater or equal 2*PI
ippStsTrnglAsymErr	The asymmetry of the Triangle is less -PI or greater or equal PI
ippStsTrnglFreqErr	The frequency of the Triangle is less 0 or greater or equal 0.5
ippStsTrnglMagnErr	The magnitude of the Triangle is less or equal to zero
ippStsTrnglPhaseErr	The phase of the Triangle is less 0 or greater or equal 2*PI
ippStsUnderFlowErr	Underflow of result
ippStsWtOffsetErr	Invalid offset value of wavelet filter
otherwise	Unknown Status Code

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 will be completed and the corresponding warning status code will be returned.

Structures and Enumerators

The Intel® IPP Image Processing Domain has a few widely used structures and enumerators.

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

The structure `IppiRect` for storing the geometric position and size of a rectangle is defined as

```
typedef struct {
   int x;
   int y;
   int width;
   int height;
} IppiRect;
```

where x, y denote the coordinates of the top left corner of the rectangle with dimensions width in the x- direction and by height in the y- direction.

The structure `IppiPoint` for storing the geometric position of a point is defined as follows:

```
typedef struct {
   int x;
   int y;
} IppiPoint;
```

where x, y denote the coordinates of the point.

The structure `IppiSize` for storing the size of a rectangle is defined as

```
typedef struct {
   int width;
   int height;
} IppiSize;
```

where width and height denote the dimensions of the rectangle in the x- and y directions, respectively.

Some structures in the library are used to store function-specific (context) information. For example, the `IppiFFTSpec` structure stores twiddle factors and bit reversal indexes that are needed to compute the fast Fourier transform. Another example is the set of internal data structures used by the statistical moment functions. These internal structures are not defined in public header files, and the structure fields are not accessible to the user.

Vector Initialization, Arithmetic, Thresholding, and Statistics

2

Vector Initialization, Arithmetic, Thresholding, and Statistics

This chapter describes the set of Intel® Integrated Performance Primitives (Intel® IPP) that are available for vector initialization and manipulation. In the interest of clarity and simplicity, the block diagrams, equations, and other detailed behavioral descriptions given throughout the remainder of this chapter apply only to the non–in–place and non–scaled (so–called "default") primitive variables, unless explicitly otherwise noted. The user should be aware that the behavior of the scaled and in–place variables can be understood easily by applying to the default behavioral specification the generic in–place and scaled function behavioral rules given in Chapter 1, "Introduction" of this manual. Users should also refer to this chapter for a detailed description of the so–called "Qm.n" format that is used by some of the primitives described in this chapter for fixed–point representation of floating point values.

The chapter is organized as follows:

"Vector Initialization" gives details on the primitives available for vector initialization.

"Vector Arithmetic – 16 Bit Data Type" covers 16-bit vector mathematical primitives.

"Vector Arithmetic – 32 Bit Data Type" covers 32-bit vector mathematical primitives.

"Vector Thresholding" covers vector sorting primitives.

"Vector Thresholding" covers vector thresholding primitives.

"Vector Statistics" covers vector statistics primitives.

"Vector Measure" covers vector measure (norms).

"Up/Down Sampling Primitives" covers upsampling and downsampling primitives.

Vector Initialization

Initialization primitives are available for vector copy, vector set, and vector zero. Vector copy allows the contents of one vector to be duplicated in another vector. Vector set changes all elements of a vector a single value. Vector zero changes all elements of a vector have the value zero. Details on each of these primitives are given next.

Copy_16s

Prototype

```
IppStatus ippsCopy_16s(const Ipp16s * pSrc, Ipp16s * pDst, int len);
```

Description

Copies the `len` elements of the vector pointed to by `pSrc` into the `len` elements of the vector pointed to by `pDst`. That is:

$$pDst[i] = pSrc[i],\ i = 0, 1, \ldots, len - 1$$

Input Arguments

- `pSrc` – pointer to the source vector
- `len` – number of elements contained in the source and destination vectors

Output Arguments

- `pDst` – pointer to the destination vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Set_16s

Prototype

```
IppStatus ippsSet_16s(Ipp16s val, Ipp16s * pDst, int len);
```

Description

Initialize the `len` elements of the vector pointed by `pDst` to the value of the variable `val`. That is:

$$pDst[i] = val, \quad len = 0, 1, \ldots, len - 1$$

Input Arguments

- `val` – the value to be copied into the `len` elements of the vector pointed to by `pDst`
- `len` – the number of vector elements to be initialized

Output Arguments

`pDst` – pointer to the vector to be initialized

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Zero_16s

Prototype

```
IppStatus ippsZero_16s(Ipp16s * pDst, int len);
```

Description

Initialize the `len` elements of the vector pointed by `pDst` to the value 0.

$$pDst[i] = 0,\ i = 0, 1, \ldots, len - 1$$

Input Arguments

`len` – the length of the vector to be initialized

Output Arguments

`pDst` – pointer to the vector to be initialized

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Vector Initialization Usage Examples

Example 2-1 shows the usage of the ippsZero_16s primitive. In the example, a vector containing 100 elements is initialized such that all elements have a value of 0.

Example 2-1 ippsZero_16s

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"
int main()
{
       Ipp16s x[100];
       int i;

       /* initialize vector x to zero */
       ippsZero_16s(x, 100);

       /* print out the output */
       for ( i = 0; i < 100; i ++ ) {
              printf("%6d", x[i]);
              if ( (i+1)%5 == 0 ) {
                 printf("\n");
              }
       }

       return(0);
}
```

Vector Arithmetic – 16 Bit Data Type

Mathematical primitives are available for pointwise vector add, subtract, multiply, square, square root, exponentiation, natural logarithm, absolute value, and normalization. For non–pointwise operations, a vector dot (inner) product is also included in the API. The details for all of these primitives are given next. The “default” function behavior and arguments are described for each primitive. Scaled and in–place primitive variables can be understood easily by applying the behavioral rules given in Chapter 1.

Abs_16s

Prototype

```
IppStatus ippsAbs_16s(const Ipp16s * pSrc, Ipp16s * pDst, int len);
IppStatus ippsAbs_16s_I(Ipp16s * pSrcDst, int len);
```

Description

Pointwise vector magnitude – Computes the absolute values of the elements of a real–valued vector.

$$pDst[k] = |pSrc[k]|, k = 0, 1, \ldots, len - 1$$

Input Arguments

- `pSrc, pSrcDst` – pointer to the input vector
- `len` – number of elements contained in both the input and output vectors

`pSrc, pSrcDst` – pointer to the input vector

`len` – number of elements contained in both the input and output vectors

Output Arguments

- `pDst, pSrcDst` – pointer to the vector of absolute values

`pDst, pSrcDst` – pointer to the vector of absolute values

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Add_16s

Prototype

```
IppStatus ippsAdd_16s(const Ipp16s * pSrc1, const Ipp16s * pSrc2, Ipp16s
   * pDst, int len);
IppStatus ippsAdd_16s_I(const Ipp16s * pSrc, Ipp16s * pSrcDst, int len);
IppStatus ippsAdd_16s_Sfs(const Ipp16s * pSrc1, const Ipp16s * pSrc2,
   Ipp16s * pDst, int len, int scaleFactor);
IppStatus ippsAdd_16s_ISfs(const Ipp16s * pSrc, Ipp16s * pSrcDst, int
   len, int scaleFactor);
```

Description

Pointwise vector addition – adds the elements of one vector to the corresponding elements of a second vector. That is:

$$pDst[i] = pSrc1[i] + pSrc2[i],\ i = 0, 1, \ldots, len - 1$$

Input Arguments

- `pSrc1`, `pSrc2`, `pSrc`, `pSrcDst` – pointers to the input vector(s)
- `len` – number of elements contained in the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (only for the scaled primitive)

AddC_16s

Prototype

```
IppStatus ippsAddC_16s (const Ipp16s * pSrc, Ipp16s val, Ipp16s * pDst,
   int len);
IppStatus ippsAddC_16s_I(Ipp16s val, Ipp16s * pSrcDst, int len);
IppStatus ippsAddC_16s_Sfs (const Ipp16s * pSrc, Ipp16s val, Ipp16s *
   pDst, int len, int scaleFactor);
```

```
IppStatus ippsAddC_16s_ISfs(Ipp16s val, Ipp16s * pSrcDst, int len, int
   scaleFactor);
```

Description

Pointwise vector addition of a constant – adds the value of the variable `val` to each of the elements of a vector of length `len`. That is: $pDst[i] = (pSrc[i]) + val$, $i = 0, 1, \ldots, len - 1$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the input vector
- `val` – the value to be added to the elements of the input vector
- `len` – number of elements contained in the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (only for the scaled primitive)

Output Arguments

`pDst`, `pSrcDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Output Arguments

`pDst`, `pSrcDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

DotProd_16s

Prototype

```
IppStatus ippsDotProd_16s(const Ipp16s * pSrc1, const Ipp16s * pSrc2, int
   len, Ipp16s * pResult);
```

```
IppStatus ippsDotProd_16s_Sfs(const Ipp16s * pSrc1, const Ipp16s * pSrc2,
   int len, Ipp16s * pResult, int scaleFactor);
```

Description

Vector dot (inner) product – computes the dot (inner) product of two vectors. That is:

$$*pResult = \sum_{j=0}^{len-1} pSrc1[j] \cdot pSrc2[j]$$

Input Arguments

- `pSrc1`, `pSrc2` – pointers to the input vectors
- `len` – number of elements contained in each of the input vectors
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pResult` – pointer to the value of the dot product

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Exp_16s

Prototype

```
IppStatus ippsExp_16s(const Ipp16s * pSrc, Ipp16s * pDst, int len);
IppStatus ippsExp_16s_I(Ipp16s * pSrcDst, int len);
IppStatus ippsExp_16s_Sfs(const Ipp16s * pSrc, Ipp16s * pDst, int len,
   int scaleFactor);
IppStatus ippsExp_16s_ISfs(Ipp16s * pSrcDst, int len, int scaleFactor);
```

Description

Pointwise vector exponentiation – raises the constant e to the power of each vector element. That is:

$$pDst[i] = e^{pSrc[i]} \qquad 0 \le i < len$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the vector of powers to which the constant e should be raised
- `len` – number of elements contained in both the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst`, `pSrcDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Ln_16s

Prototype

```
IppStatus ippsLn_16s(const Ipp16s * pSrc, Ipp16s * pDst, int len);
IppStatus ippsLn_16s_I(Ipp16s * pSrcDst, int len);
IppStatus ippsLn_16s_Sfs(const Ipp16s * pSrc, Ipp16s * pDst, int len, int
   scaleFactor);
IppStatus ippsLn_16s_ISfs(Ipp16s * pSrcDst, int len, int scaleFactor);
```

Description

Pointwise vector natural logarithm – computes the natural log (log base e) of each element of a vector. That is:

$$pDst[i] = \ln(pSrc[i]) \qquad 0 \le i < len$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the input vector
- `len` – number of elements contained in both the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

NOTE. *The elements of the input vector should be positive, or the result of this function is unpredictable*

Output Arguments

`pDst`, `pSrcDst`– pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsErr` – input is not positive
- `ippStsBadArgErr` – bad arguments

Mul_16s

Prototype

```
IppStatus ippsMul_16s(const Ipp16s * pSrc1, const Ipp16s * pSrc2, Ipp16s
   * pDst, int len);
IppStatus ippsMul_16s_I(const Ipp16s * pSrc, Ipp16s * pSrcDst, int len);
IppStatus ippsMul_16s_Sfs(const Ipp16s * pSrc1, const Ipp16s * pSrc2,
   Ipp16s * pDst, int len, int scaleFactor);
IppStatus ippsMul_16s_ISfs(const Ipp16s * pSrc, Ipp16s * pSrcDst, int
   len, int scaleFactor);
```

Description

Pointwise vector multiplication – multiplies the elements of one vector by the corresponding elements of a second vector. That is:

$$pDst[i] = pSrc1[i] \times pSrc2[i],\ i = 0, 1, \ldots, len - 1$$

Input Arguments

- `pSrc`, `pSrc1`, `pSrc2`, `pSrcDst` – pointers to the input vector(s)
- `len` – number of elements contained in the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst`, `pSrcDst` – pointer to the vector containing the output product

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

MulC_16s

Prototype

```
IppStatus ippsMulC_16s(const Ipp16s * pSrc, Ipp16s val, Ipp16s * pDst,
   int len);
IppStatus ippsMulC_16s_Sfs(const Ipp16s * pSrc, Ipp16s val, Ipp16s *
   pDst, int len, int scaleFactor);
IppStatus ippsMulC_16s_I(Ipp16s val, Ipp16s * pSrcDst, int len);
IppStatus ippsMulC_16s_ISfs(Ipp16s val, Ipp16s * pSrcDst, int len, int
   scaleFactor);
```

Description

Pointwise vector multiplication by a constant – multiplies each element of a vector by the value of the variable `val`. That is:

$$pDst[i] = pSrc[i] \times val,\ i = 0, 1, \ldots, len - 1$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the input vector

- `val` – the value to be multiplied with the elements of the input vector
- `len` – number of elements contained in the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst`, `pSrcDst` – pointer to the vector containing the output product

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Normalize_16s

Prototype

```
IppStatus ippsNormalize_16s(const Ipp16s * pSrc, Ipp16s * pDst, int len,
   Ipp16s offset, int normFactor);
IppStatus ippsNormalize_16s_Sfs(const Ipp16s * pSrc, Ipp16s * pDst, int
   len, Ipp16s offset, int normFactor, int scaleFactor);
```

Description

Vector normalization – subtracts the value of the variable `offset` from the elements of a vector, and then divides the resulting `offset` vector by the value of the variable `normFactor`. That is:

$$pDst[k] = (pSrc[k] - offset)/(normFactor),\ k = 0, 1, 2, \ldots, len - 1$$

- `pSrc` – pointer to the input vector (the vector to be normalized)
- `len` – number of elements contained in the input and output vectors
- `offset` – the constant to be subtracted from the elements of the input vector
- `normFactor` – the constant by which the offset elements of the input vector are divided `normFactor` is not zero
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst` – pointer to the normalized output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Sqrt_16s

Prototype

```
IppStatus ippsSqrt_16s(const Ipp16s * pSrc, Ipp16s * pDst, int len);
IppStatus ippsSqrt_16s_I(Ipp16s * pSrcDst, int len);
IppStatus ippsSqrt_16s_Sfs(const Ipp16s *pSrc, Ipp16s * pDst, int len,
   int scaleFactor);
IppStatus ippsSqrt_16s_ISfs(Ipp16s * pSrcDst, int len, int scaleFactor);
```

Description

Pointwise vector square root – computes the square root of each vector element. That is:

$$pDst[i] = \sqrt{pSrc[i]}, i = 0, 1, \ldots, len-1$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the vector of which to compute the pointwise square root
- `len` – number of elements contained in both the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

NOTE. *The elements of the input vector must be non–negative, otherwise the behavior of this function is undefined, and unpredictable output values may result.*

Output Arguments

`pDst`, `pSrcDst` – pointer to the square root output vector

Returns

- `ippStsNoErr` – no error
- `ippStsErr` – input is negative
- `ippStsBadArgErr` – bad arguments

Sqr_16s

Prototype

```
IppStatus ippsSqr_16s(const Ipp16s * pSrc, Ipp16s * pDst, int len);
IppStatus ippsSqr_16s_I(Ipp16s * pSrcDst, int len);
IppStatus ippsSqr_16s_Sfs(const Ipp16s * pSrc, Ipp16s * pDst, int len,
   int scaleFactor);
IppStatus ippsSqr_16s_ISfs(Ipp16s * pSrcDst, int len, int scaleFactor);
```

Description

Pointwise vector square – raises each element of a vector to the second power. That is:

$$pDst[i] = pSrc^2[i] \text{ , } i = 0, 1, \ldots, len-1$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the input vector
- `len` – number of elements contained in both the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst`, `pSrcDst` – pointer to the squared output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Sub_16s

Prototype

```
IppStatus ippsSub_16s(const Ipp16s * pSrc1, const Ipp16s * pSrc2, Ipp16s
   * pDst, int len);
IppStatus ippsSub_16s_I(const Ipp16s * pSrc, Ipp16s * pSrcDst, int len);
IppStatus ippsSub_16s_Sfs(const Ipp16s * pSrc1, const Ipp16s * pSrc2,
   Ipp16s * pDst, int len, int scaleFactor);
IppStatus ippsSub_16s_ISfs(const Ipp16s * pSrc, Ipp16s * pSrcDst, int
   len, int scaleFactor);
```

Description

Pointwise vector subtraction – subtracts the elements of one vector from the corresponding elements of a second vector. That is:

$$pDst[i] = pSrc2[i] - pSrc1[i], \; i = 0, 1, \ldots, len - 1$$

Input Arguments

- `pSrc`, `pSrc1`, `pSrc2`, `pSrcDst`– pointers to the input vector(s)
- `len` – number of elements contained in the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (only for the scaled primitive)

Output Arguments

`pDst`, `pSrcDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

SubC_16s_I

Prototype

```
IppStatus ippsSubC_16s (const Ipp16s * pSrc, Ipp16s val, Ipp16s * pDst,
   int len);
IppStatus ippsSubC_16s_I(Ipp16s val, Ipp16s * pSrcDst, int len);
IppStatus ippsSubC_16s_Sfs (const Ipp16s * pSrc, Ipp16s val, Ipp16s *
   pDst, int len, int scaleFactor);
IppStatus ippsSubC_16s_ISfs(Ipp16s val, Ipp16s * pSrcDst, int len, int
   scaleFactor);
```

Description

Pointwise vector subtraction of a constant – subtracts the value of the variable `var` from each element of a vector. That is:

$$pDst[i] = pSrc[i] - val,\ i = 0, 1, \ldots, len - 1$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the input vector
- `len` – number of elements contained in the input and output vectors
- `val` – the value to be subtracted from the elements of the input vector
- `scaleFactor` – saturation fixed scalefactor (only for the scaled primitive)

Output Arguments

`pDst`, `pSrcDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

SubCRev_16s_I

Prototype

```
IppStatus ippsSubCRev_16s_I(Ipp16s val, Ipp16s * pSrcDst, int len);
IppStatus ippsSubCRev_16s_ISfs(Ipp16s val, Ipp16s * pSrcDst, int len, int
   scaleFactor);
```

Description

Pointwise inverse vector subtraction of a constant – subtracts each element of a vector from the value of the variable `val`. That is:

$$pSrcDst[i] = val - pSrcDst[i], \quad i = 0, 1, \ldots, len - 1$$

Input Arguments

- `val` – value from which the elements of the input vector are subtracted
- `pSrcDst` – pointers to the vector containing the elements to be subtracted from the value of the variable `val`
- `len` – number of elements contained in the input and output vectors
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pSrcDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Usage Examples of Vector Math for 16-Bit Data

Example 2-2 illustrates the usage of the `ippsAdd_16s` primitive. Two vectors of length 100 are added together pointwise, and then the result is stored in a third vector.

Example 2-2 ippsAdd_16s

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"

int main()
{
 Ipp16s x[100], y[100], z[100];
      int i;

      /* Initialize x and y vector */
      for ( i = 0; i < 100; i ++ ) {
            x[i] = y[i] = i;
      }

      /* Add the vectors x and y, store result in the vector z */
      ippsAdd_16s(x, y, z, 100);

      /* print out the output */
      for ( i = 0; i < 100; i ++ ) {
          printf("%6d", z[i]);
          if ( (i+1)%5 == 0 ) {
              printf("\n");
          }
      }

      return(0);
}
```

Example 2-3 illustrates the usage of the `ippsMul_16s` primitive. One vector of length 100 is multiplied pointwise with a second vector of length 100, and then the result is stored in a third vector.

Example 2-3 ippsMul_16s

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"


int main()
{
       Ipp16s x[100], y[100], z[100];
       int i;

       /* Initialize x and y vector */
       for ( i = 0; i < 100; i ++ ) {
           x[i] = y[i] = i;
       }

        /* Multiply y vector to x vector and store result in z vector*/
       ippsMul_16s(x, y, z, 100);

       /* print out the output */
       for ( i = 0; i < 100; i ++ ) {
           printf("%6d", z[i]);
           if ( (i+1)%5 == 0 ) {
           printf("\n");
           }
       }

       return(0);
}
```

Example 2-4 illustrates the usage of the `ippsNormalize_16s` primitive. A value of 50 is subtracted from a vector containing 100 elements. Next, the resulting vector is divided pointwise by a scalefactor of 3.

Example 2-4 ippsNormalize_16s

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"

int main()
{
     Ipp16s x[100], y[100], offset = 50;
     int i, normFactor = 3;

     /* Initialize x and y vector */
     for ( i = 0; i < 100; i ++ ) {
        x[i] = i;
     }

     /* Threshold the vector x */
     ippsNormalize_16s(x, y, 100, offset, normFactor);

     /* print out the output */
     for ( i = 0; i < 100; i ++ ) {
        printf("%6d", y[i]);
        if ( (i+1)%5 == 0 ) {
                   printf("\n");
        }
     }

     return(0);
}
```

Example 2-5 illustrates the usage of the `ippsDotProd_16s` primitive. The dot product is computed for two vectors, each containing 100 elements.

Example 2-5 ippsDotProd_16s

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"

int main()
{
       Ipp16s x[100], y[100], result;
       int i;

       /* Initialize x and y vector */
       for ( i = 0; i < 100; i ++ ) {
           x[i] = y[i] = i;
       }

       /* Dot product y vector to x vector */
       ippsDotProd_16s(x, y, 100, &result);

       /* print out the output */
       printf("%6d", result);

       return(0);
}
```

Example 2-6 illustrates the usage of the `ippsLn_16s_I` primitive.

Example 2-6 ippsLn_16s_I

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"

int main()
{
       Ipp16s x[100];
       int i;

       /* Initialize x */
       for ( i = 0; i < 100; i ++ ) {
              x[i] = i + 1;
       }

       /* Calculate the natural logarithm of vector x */
       ippsLn_16s_I(x, 100);

       /* print out the output */
       for ( i = 0; i < 100; i ++ ) {
           printf("%6d", x[i]);
           if ( (i+1)%5 == 0 ) {
                     printf("\n");
           }
       }

       return(0);
}
```

Vector Arithmetic – 32 Bit Data Type

Mathematical primitives are available for pointwise vector add, subtract, multiply, square, square root, exponentiation, natural logarithm, absolute value, and normalization. For non–pointwise operations, a vector dot (inner) product is also included in the API. The details for all of these

primitives are given next. The “default” function behavior and arguments are described for each primitive. Scaled and in-place primitive variables can be understood easily by applying the behavioral rules given in Chapter 1.

10Log10_32s

Prototype

```
IppStatus ipps10Log10_32s (const Ipp32s * pSrc, Ipp32s * pDst,
   int len);
IppStatus ipps10Log10_32s_I (Ipp32s * pSrcDst, int len);
IppStatus ipps10Log10_32s_Sfs (const Ipp32s * pSrc, Ipp32s * pDst, int
   len, int scaleFactor);
IppStatus ipps10Log10_32s_ISfs (Ipp32s * pSrcDst, int len, int
   scaleFactor);
```

Description

Pointwise vector scaled base 10 logarithm – This function computes the log base 10 of each element of a vector and scales the result by 10.0. That is:

$$pDst[k] = 10.0 * log10(pSrc1[k]),\ k = 0, 1, \ldots, len-1$$

This function is used in conversion to decibels. The output is rounded to the nearest integer.

Input Arguments

- `pSrc`, `SrcDst` – pointer to the input vector
- `len` – number of elements contained in the input and output vectors (0 < `len` < 65536)
- `scaleFactor` – saturation fixed scale factor (scaled primitives only)
  (-32 < scaleFactor < 32)

NOTE. *The elements of the input vector must be positive, or the result of this function is unpredictable.*

Output Arguments

`pDst`, `pSrcDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsError` – input is not positive
- `ippStsNullPtrErr` – `pSrc`, `pSrcDst`, or `pDst` is NULL
- `ippStsLengthErr` – illegal value for len
- `ippStsRangeErr` – `pSrc[i]`<0 or `scaleFactor` is out of range

Abs_32s

Prototype

```
IppStatus ippsAbs_32s (const Ipp32s *pSrc, Ipp32s *pDst, int len);
IppStatus ippsAbs_32s_I (const Ipp32s *pSrcDst, int len);
IppStatus ippsAbs_32sc32s (const Ipp32sc *pSrc, Ipp32s *pDst, int len);
IppStatus ippsAbs_32sc32s_Sfs (const Ipp32sc *pSrc, Ipp32s *pDst, int
   len, int scaleFactor);
```

Description

Pointwise vector magnitude – This function computes the absolute values of the elements of a real-valued (or complex valued in the case of ippAbs_32sc32s) vector.

$$pDst[k] = |pSrc[k]|,\ k = 0, 1, \ldots, len-1$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the input vector
- `len` – number of elements contained in both the input and output vectors (0 < `len` < 65536)
- `scaleFactor` – saturation fixed scale factor (scaled primitives only)
  (-32 < `scaleFactor` < 32)

Output Arguments

- `pDst`, `pSrcDst` – pointer to the vector of absolute values

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrcOld`, `pSrcDst`, or `pDst` is NULL
- `ippStsLengthErr` – illegal value for len
- `ippStsRangeErr` – `scaleFactor` is out of range

Add_32s

Prototype

```
IppStatus ippsAdd_32s(const Ipp32s * pSrc1, const Ipp32s * pSrc2, Ipp32s
   * pDst, int len);
IppStatus ippsAdd_32s_Sfs(const Ipp32s * pSrc1, const Ipp32s * pSrc2,
   Ipp32s * pDst, int len, int scaleFactor);
```

Description

Adds the elements of two vectors. Results in a third vector.

$pDst[k] = pSrc1[k] + pSrc2[k]$, $k = 0, 1, \ldots, len - 1$.

Adds the elements of two vectors with output scaling. Results in a third vector.

$pDst[k] = (pSrc1[k] + pSrc2[k]) \times 2^{-scaleFactor}$, $k = 0, 1, \ldots, len - 1$.

Input Arguments

- `pSrc1` – pointer to the vector to add
- `pSrc2` – pointer to the vector to be added
- `len` – length of the input and output vector
- `scaleFactor` – scaling factor – range:[-31,31]

Output Arguments

`pDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:

— a pointer was NULL
— one or more vectors was 0 in length.
— `scaleFactor` was outside the range of –31 to 31

Add_32sc

Prototype

```
IppStatus ippsAdd_32sc(const Ipp32sc * pSrc1, const Ipp32sc * pSrc2,
   Ipp32sc * pDst, int len);
IppStatus ippsAdd_32sc_Sfs(const Ipp32sc * pSrc1, const Ipp32sc * pSrc2,
   Ipp32sc * pDst, int len, int scaleFactor);
```

Description

Adds the elements of two vectors. Results in a third vector.

$pDst[k].re = pSrc1[k].re + pSrc2[k].re$,

$pDst[k].im = pSrc1[k].im + pSrc2[k].im$

where $k = 0, 1, \ldots, len - 1$.

Adds the elements of two vectors with output scaling. Results in a third vector.

$pDst[k].re = (pSrc1[k].re + pSrc2[k].re) \times 2^{-scaleFactor}$

$pDst[k].im = (pSrc1[k].im + pSrc2[k].im) \times 2^{-scaleFactor}$

where $k = 0, 1, \ldots, len - 1$.

Input Arguments

- `pSrc1` – pointer to the vector to add
- pSrc2 – pointer to the vector to be added
- `len` – length of the input and output vector
- `scaleFactor` – Scaling factor – range:[-31,31]

Output Arguments

`pDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - one or more vectors was 0 in length.
  - `scaleFactor` was outside the range of –31 to 31

AddWeightedQ31_32s

Prototype

```
IppStatus ippsAddWeightedQ31_32s (const Ipp32s *pSrcOld,
   const Ipp32s *pSrcNew, Ipp32s *pDst, int len, Ipp32s weightQ31);
IppStatus ippsAddWeightedQ31_32s_I (const Ipp32s *pSrcNew,
   Ipp32s *pSrcDst, int len, Ipp32s weightQ31);
```

Description

This function calculates the weighted average of two vectors. The weighted average expressed in the equation

$$X_{dest}(k) = weight \cdot X_{new}(k) + (1 - weight) \cdot X_{old}(k)$$

where *k* is the element index within the vector and *weight* is a Q31 value between 0 and 1.

Input Arguments

- `pSrcOld, pSrcDst` – pointer to the real-valued vector representing the previous average
- `pSrcNew` – pointer to the real-valued vector representing the new measurement
- `len` – number of elements contained in both input and output vectors ($0 <$ `len` < 65536)
- `weightQ31` – real-valued Q31 scalar ($0.0_{Q31} \leq$ `weightQ31` $< 1.0_{Q31}$)

Output Arguments

`pDst, pSrcDst` – pointer to the real-valued output vector

Returns

- `ippStsNoErr` – no error

- ippStsNullPtrErr – pSrcOld, pSrcNew, pSrcDst, or pDst is NULL
- ippStsLengthErr – illegal value for len
- ippStsRangeErr – weightQ31 is out of range

Div_32s_Sfs

Prototype

```
IppStatus ippsDiv_32s_Sfs (const Ipp32s *pSrc1, const Ipp32s *pSrc2,
   Ipp32s *pDst, int len, int scaleFactor);
IppStatus ippsDiv_32s_ISfs (const Ipp32s *pSrc, Ipp32s *pSrcDst, int len,
   int scaleFactor);
IppStatus ippsDivQ15_32s (const Ipp32s *pSrc1, const Ipp32s *pSrc2,
   Ipp32s *pDstQ15, int len);
IppStatus ippsDivQ15_32s_I (const Ipp32s *pSrc, Ipp32s *pSrcDst, int
   len);
```

Description

Pointwise vector division – This function divides the elements of one vector by the corresponding elements of a second vector. That is:

$$pDst[k] = pSrc1[k] / pSrc2[k],\ k = 0, 1, \ldots, len-1$$

Input Arguments

- IpSrc,pSrc1, pSrcDst – pointer to the numerator input vector
- pSrc2 – pointer to the denominator input vector
- len – number of elements contained in the input and output vectors (0 < len < 65536)
- scaleFactor – saturation fixed scale factor (scaled primitives only) (-32 < scaleFactor < 32)

Output Arguments

- pDstQ15, pSrcDst – pointer to the Q15 format output vector ($-32768.0_{Q15} \le$ pDstQ15[k] $< 32768.0_{Q15}$)
- pDst, pSrcDst – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pSrc1`, `pSrc2`, `pSrcDst`, `pDstQ15`, or `pDst` is NULL
- `ippStsLengthErr` – illegal value for `len`
- `ippStsRangeErr` – `scaleFactor` is out of range

FilterMedian_32s

Prototype

```
IppStatus ippsFilterMedian_32s (const Ipp32s* pSrc, Ipp32s* pDst,
   int len, int maskSize);
IppStatus ippsFilterMedian_32s_I (Ipp32s* pSrcDst, int len, int
   maskSize);
```

Description

This function computes the median values for each element of the input array `pSrcDst`, and stores the result in `pSrcDst`. See Chapter 16, "Image Processing", for a description of 2-dimensional median filtering primitives.

NOTE. *The value of a non-existent point is equal to the last point value. Specifically, x[-1]=x[0] and x[len]=x[len-1].*

Input Arguments

- `pSrcDst` – pointer to input and output array
- `len` – number of elements contained in the input and output vectors ($0 <$ `len` < 65536)
- `maskSize` – median mask size. If an even value is specified, the function subtracts 1 and uses the odd value of the filter mask for median filtering ($0 <$ `maskSize` ≤ 256).

Output Arguments

`pSrcDst` – pointer to input and output array

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pSrcDst`, or `pDst` is NULL
- `ippStsSizeErr` – indicates an error when len is less than or equal to 0, `maskSize` is less than or equal to zero, `maskSize` is greater than 256, or `maskSize` is greater than `len`
- `ippStsEvenMedianMaskSize` – indicates a warning when the median mask length is even

Ln_32s

Prototype

```
IppStatus ippsLn_32s (const Ipp32s * pSrc, Ipp32s * pDst, int len);
IppStatus ippsLn_32s_I (Ipp32s * pSrcDst, int len);
IppStatus ippsLn_32s_Sfs (const Ipp32s * pSrc, Ipp32s * pDst, int len,
   int scaleFactor);
IppStatus ippsLn_32s_ISfs (Ipp32s * pSrcDst, int len, int scaleFactor);
```

Description

Pointwise vector natural logarithm – This function computes the natural log (log base e) of each element of a vector. That is:

$$pDst[k] = ln(pSrc1[k]),\ k = 0, 1, \ldots, len-1$$

The output is rounded to the nearest integer.

Input Arguments

- `pSrc`, `SrcDst` – pointer to the input vector
- `len` – number of elements contained in the input and output vectors ($0 <$ `len` < 65536)
- `scaleFactor` – saturation fixed scale factor (scaled primitives only)
  ($-32 <$ scaleFactor < 32)

NOTE. *The elements of the input vector must be positive, or the result of this function is unpredictable.*

Output Arguments

`pDst`, `pSrcDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsError` – input is not positive
- `ippStsNullPtrErr` – `pSrc`, `pSrcDst`, or `pDst` is NULL
- `ippStsLengthErr` – illegal value for len
- `ippStsRangeErr` – `pSrc[i]`<0 or `scaleFactor` is out of range

MagSquared_32sc32s

Prototype

```
IppStatus ippsMagSquared_32sc32s (const Ipp32sc *pSrc,
   Ipp32s *pDst, int len);
IppStatus ippsMagSquared_32sc32s_Sfs (const Ipp32sc *pSrc,
   Ipp32s *pDst, int len, int scaleFactor);
```

Description

This function calculates the squared magnitude of the entries of a complex vector. (Note: This primitive performs the same role as the `ippsPowerSpectr` primitive on the IA32 platform.)

$$pDst[k] = |pSrc[k]|^2,\ k = 0, 1, \ldots, len-1$$

Input Arguments

- `pSrc` – pointer to the complex-valued vector that contains the FFT output
- `len` – number of elements contained in both input and output vectors (0 < `len` < 65536)

- `scaleFactor` – saturation fixed scale factor (scaled primitive only) (-32 < scaleFactor < 32)

Output Arguments

`pDst` – pointer to the real-valued magnitude squared output vector

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc` or `pDst` is NULL
- `ippStsLengthErr` – illegal value for `len`
- `ippStsRangeErr` – `scaleFactor` is out of range

Mul_32s32sc_Sfs

Prototype

```
IppStatus ippsMul_32s32sc_Sfs (const Ipp32s *pSrc1,
   const Ipp32sc *pSrc2, Ipp32sc *pDst, int len, int scaleFactor);
IppStatus ippsMul_32s32sc_ISfs (const Ipp32s *pSrc, Ipp32sc *pSrcDst,
   int len, int scaleFactor);
```

Description

This function performs pointwise multiplication of a real vector by a complex vector.

$$pDst[k].re = pSrc1[k] * pSrc2[k].re, \quad k = 0, 1, \ldots, len-1$$
$$pDst[k].im = pSrc1[k] * pSrc2[k].im, \quad k = 0, 1, \ldots, len-1$$

Input Arguments

- `pSrc`, `pSrc1` – pointer to the real-valued input vector
- `pSrc2`, `pSrcDst` – pointer to the complex-valued input vector
- `len` – number of elements contained in both input and output vectors (0 < `len` < 65536)
- `scaleFactor` – saturation fixed scale factor (-32 < `scaleFactor` < 32)

Output Arguments

`pDst`, `pSrcDst` – pointer to the complex-valued product (output) vector

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pSrc1`, `pSrc2`, `pSrcDst`, or `pDst` is NULL
- `ippStsLengthErr` – illegal value for `len`
- `ippStsRangeErr` – `scaleFactor` is out of range

Mul_32sc

Prototype

```
IppStatus ippsMul_32sc(const Ipp32sc * pSrc1, const Ipp32sc * pSrc2,
   Ipp32sc * pDst, int len);
IppStatus ippsMul_32sc_Sfs(const Ipp32sc * pSrc1, const Ipp32sc * pSrc2,
   Ipp32sc * pDst, int len, int scaleFactor);
```

Description

Multiplies the elements of two vectors. Results in a third vector.

pDst[*k*].re = *pSrc1*[*k*].re × *pSrc2*[*k*].re - *pSrc1*[*k*].im × *pSrc2*[*k*].im,

pDst[*k*].im = *pSrc1*[*k*].re × *pSrc2*[*k*].im + *pSrc1*[*k*].im × *pSrc2*[*k*].re

Where k = 0, 1, … , *len* - 1.

Multiplies the elements of two vectors with output scaling. Results in a third vector.

pDst[*k*].re = (*pSrc1*[*k*].re × *pSrc2*[*k*].re - *pSrc1*[*k*].im × *pSrc2*[*k*].im) × $2^{-scaleFactor}$,

pDst[*k*].im =(*pSrc1*[*k*].re × *pSrc2*[*k*].im + *pSrc1*[*k*].im × *pSrc2*[*k*].re) × $2^{-scaleFactor}$

Where k = 0, 1, … , *len* - 1.

Input Arguments

- `pSrc1` – pointer to the vector to be multiplied
- `pSrc2` – pointer to the vector to multiply
- `len` – length of the input and output vector
- `scaleFactor` - scaling factor – range:[-31,31]

Output Arguments

`pDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - one or more vectors was 0 in length.
  - `scaleFactor` was outside the range of –31 to 31

Mul_32s

Prototype

```
IppStatus ippsMul_32s(const Ipp32s * pSrc1, const Ipp32s * pSrc2, Ipp32s
   * pDst, int len);
```

```
IppStatus ippsMul_32s_Sfs(const Ipp32s * pSrc1, const Ipp32s * pSrc2,
   Ipp32s * pDst, int len, int scaleFactor);
```

Description

Multiplies the elements of two vectors. Results in a third vector.

$pDst[k] = pSrc1[k] \times pSrc2[k]$, $k = 0, 1, \ldots, len - 1$.

Multiplies the elements of two vectors with output scaling. Results in a third vector.

$pDst[k] = (pSrc1[k] \times pSrc2[k]) \times 2^{-scaleFactor}$, $k = 0, 1, \ldots, len - 1$.

Input Arguments

- `pSrc1` – pointer to the vector to be multiplied
- `pSrc2` – pointer to the vector to multiply
- `len` – length of the input and output vector
- `scaleFactor` – scaling factor. – range:[-31,31]

Output Arguments

`pDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - one or more vectors was 0 in length.
  - `scaleFactor` was outside the range of –31 to 31

Sub_32sc

Prototype

```
IppStatus ippsSub_32sc(const Ipp32sc * pSrc1, const Ipp32sc * pSrc2,
   Ipp32sc * pDst, int len);
IppStatus ippsSub_32sc_Sfs(const Ipp32sc * pSrc1, const Ipp32sc * pSrc2,
   Ipp32sc * pDst, int len, int scaleFactor);
```

Description

Subtracts the elements of two vectors. Results in a third vector.

pDst[*k*].re = *pSrc2*[*k*].re - *pSrc1*[*k*].re,

pDst[*k*].im = *pSrc2*[*k*].im - *pSrc1*[*k*].im

where *k* = 0, 1, … , *len* - 1.

Subtracts the elements of two vectors with output scaling. Results in a third vector.

pDst[*k*].re = (*pSrc2*[*k*].re - p*Src1*[*k*].re) × $2^{-scaleFactor}$

pDst[*k*].im = (*pSrc2*[*k*].im - p*Src1*[*k*].im) × $2^{-scaleFactor}$

where *k* = 0, 1, … , *len* - 1.

Input Arguments

- `pSrc1` – pointer to the subtrahend, i.e., the vector to be subtracted from the minuend
- `pSrc2` – pointer to the minuend, i.e., the vector from which the subtrahend will be subtracted
- `len` – length of the input and output vector
- `scaleFactor` – scaling factor – range:[-31,31]

Output Arguments

`pDst` – pointer to output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - one or more vectors was 0 in length.
  - `scaleFactor` was outside the range of –31 to 31

Sub_32s

Prototype

```
IppStatus ippsSub_32s(const Ipp32s * pSrc1, const Ipp32s * pSrc2, Ipp32s
   * pDst, int len);
IppStatus ippsSub_32s_Sfs(const Ipp32s * pSrc1, const Ipp32s * pSrc2,
   Ipp32s * pDst, int len, int scaleFactor);
```

Description

Subtracts the elements of two vectors. Results in a third vector.

$pDst[k] = pSrc2[k] - pSrc1[k]$, $k = 0, 1, ..., len - 1$.

Subtracts the elements of two vectors with output scaling. Results in a third vector.

$pDst[k] = (pSrc2[k] - pSrc1[k]) \times 2^{-scaleFactor}$, $k = 0, 1, ..., len - 1$.

Input Arguments

- `pSrc1` – pointer to the subtrahend, i.e., the vector to be subtracted from the minuend
- `pSrc2` – pointer to the minuend, i.e., the vector from which the subtrahend will be subtracted
- `len` – length of the input and output vector
- `scaleFactor` – scaling factor. – range:[-31,31]

Output Arguments

- `pDst` – pointer to output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - one or more vectors was 0 in length.
  - `scaleFactor` was outside the range of –31 to 31

Sum_32s

Prototype

```
IppStatus ippsSum_32s (const Ipp32s *pSrc, int len, Ipp32s *pDst);
IppStatus ippsSum_32s_Sfs (const Ipp32s *pSrc, int len, Ipp32s *pDst, int
   scaleFactor);
```

Description

Vector sum – this function computes the sum of the elements of the input vector

$$pDst = pSrc[0]+pSrc[1]+...+ pSrc[len-1]$$

Input Arguments

- `pSrc` - pointer to the input vector
- `len` – number of elements contained in the input vector (0 < `len` < 65536)
- `scaleFactor` – saturation fixed scale factor (scaled primitives only) (-32 < `scaleFactor` < 32)

Output Arguments

`pDst` – pointer to the output

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc` or `pDst` is NULL
- `ippStsLengthErr` – illegal value for `len`
- `ippStsRangeErr` – `scaleFactor` is out of range

Usage Examples of Vector Math for 32-Bit Data

Usage of 32-bit Addition Functions

Below is an example to add two vectors with length of 100 and store the result in a third vector:

```
#include <stdio.h>
     #include "ippdefs.h"
#include "ippSP.h"

int main()
{
    Ipp32s  x[100], y[100], z[100];
    int i;

    /* Initialize x and y vector */
    for ( i = 0; i < 100; i ++ ) {
         x[i] = y[i] = i;
    }

    /* Add y vector to x vector and store result in z vector*/
    ippsAdd_32s(x, y, z, 100);

    /* print out the output */
    for ( i = 0; i < 100; i ++ ) {
        printf("%6d", z[i]);
        if ( (i+1)%5 == 0 ) {
            printf("\n");
        }
    }

    return(0);
}
```

Usage of 32-bit Subtraction Functions

Below is an example to subtract a vector from another vector with length of 100 and store the result in a third vector:

```
#include <stdio.h>
     #include "ippdefs.h"
```

```
#include "ippSP.h"

int main()
{
    Ipp32s  x[100], y[100], z[100];
    int i;

    /* Initialize x and y vector */
    for( i = 0; i < 100; i ++ ){
         x[i] = y[i] = i;
    }

    /* Subtract y vector from x vector and store result in z vector*/
    ippsSub_32s(x, y, z, 100);

    /* print out the output */
    for ( i = 0; i < 100; i ++ ) {
        printf("%6d", z[i]);
        if ( (i+1)%5 == 0 ) {
            printf("\n");
        }
    }

    return(0);
}
```

Usage of 32-bit Multiplication Functions

Below is an example to multiply two vectors with length of 100 and store the result in a third vector:

```
     #include <stdio.h>
     #include "ippdefs.h"
#include "ippSP.h"

int main()
{
    Ipp32s  x[100], y[100], z[100];
```

```
    int i;

    /* Initialize x and y vector */
    for ( i = 0; i < 100; i ++ ) {
         x[i] = y[i] = i;
        }

    /* Multiply y vector to x vector and store result in z vector*/
    ippsMul_32s(x, y, z, 100);

    /* print out the output */
    for ( i = 0; i < 100; i ++ ) {
        printf("%6d", z[i]);
        if ( (i+1)%5 == 0 ) {
            printf("\n");
        }
    }

    return(0);
}
```

Vector Thresholding

Vector sorting is useful in statistical applications where the median of a vector is desired. Vector sorting may also be useful in some signal processing applications. For example, see [Tan00].

Threshold_GT_16s

Prototype

```
IppStatus ippsThreshold_GT_16s(const Ipp16s * pSrc, Ipp16s * pDst, int
   len, Ipp16s threshold);
IppStatus ippsThreshold_GT_16s_I(Ipp16s * pSrcDst, int len, Ipp16s
   threshold);
```

Description

Vector GT threshold – compares each element of a vector against a threshold, then replaces supra–threshold elements with the value of the variable `thresh`. That is:

$$pDst[i] = \begin{cases} thresh, & pSrc[i] > threshold \\ pSrc[i], & \text{otherwise} \end{cases} \quad 0 \le i < len$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the vector to which the threshold is applied
- `threshold` – threshold value
- `len` – number of elements contained in the input and output vectors

Output Arguments

`pDst`, `pSrcDst` – pointer to the thresholded output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Threshold_LT_16s

Prototype

```
IppStatus ippsThreshold_LT_16s(const Ipp16s * pSrc, Ipp16s * pDst, int
   len, Ipp16s threshold);
IppStatus ippsThreshold_LT_16s_I(Ipp16s * pSrcDst, int len, Ipp16s
   threshold);
```

Description

Vector LT threshold – compares each element of a vector against a threshold, then replaces sub–threshold elements with the value of the variable `thresh`. That is:

$$pDst[i] = \begin{cases} thresh, & pSrc[i] < threshold \\ pSrc[i], & \text{otherwise} \end{cases} \quad 0 \le i < len$$

Input Arguments

- pSrc, pSrcDst – pointer to the vector to which the threshold is applied
- threshold – threshold value
- len – number of elements contained in the input and output vectors

Output Arguments

pDst, pSrcDst – pointer to the thresholded output vector

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments

Threshold_GTVal_16s

Prototype

```
IppStatus ippsThreshold_GTVal_16s(const Ipp16s * pSrc, Ipp16s * pDst, int
   len, Ipp16s threshold, Ipp16s val);
IppStatus ippsThreshold_GTVal_16s_I(Ipp16s * pSrcDst, int len, Ipp16s
   threshold, Ipp16s val);
```

Description

Vector GT threshold with value replacement – compares each element of a vector against a threshold, then replaces supra-threshold elements with the value of the variable val. That is:

$$pDst[i] = \begin{cases} val, & pSrc[i] > threshold \\ pSrc[i], & \text{otherwise} \end{cases} \quad 0 \le i < len$$

Input Arguments

- pSrc, pSrcDst – pointer to the vector to which the threshold is applied
- threshold – threshold value
- val – supra–threshold replacement value
- len – number of elements contained in the input and output vectors

Output Arguments

- pDst, pSrcDst – pointer to the thresholded output vector

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments

Threshold_LTVal_16s

Prototype

```
IppStatus ippsThreshold_LTVal_16s(const Ipp16s * pSrc, Ipp16s * pDst, int
   len, Ipp16s threshold, Ipp16s val);
IppStatus ippsThreshold_LTVal_16s_I(Ipp16s * pSrcDst, int len, Ipp16s
   threshold, Ipp16s val);
```

Description

Vector LT threshold with value replacement – compares each element of a vector against a threshold, then replaces sub–threshold elements with the value of the variable val. That is:

$$pDst[i] = \begin{cases} val, & pSrc[i] < threshold \\ pSrc[i], & \text{otherwise} \end{cases} \qquad 0 \le i < len$$

Input Arguments

- pSrc, pSrcDst – pointer to the vector to which the threshold is applied
- threshold – threshold value
- val – sub–threshold replacement value
- len – number of elements contained in the input and output vectors

Output Arguments

pDst, pSrcDst – pointer to the thresholded output vector

Returns

- ippStsNoErr – no error

- ippStsBadArgErr – bad arguments

Threshold_LTValGTVal_16s

Prototype

```
IppStatus IppsThreshold_LTValGTVal_16s (const Ipp16s * pSrc, Ipp16s *
   pDst, int len, Ipp16s gtThreshold, Ipp16s gtVal, Ipp16s ltThreshold,
   Ipp16s ltVal);
IppStatus IppsThreshold_LTValGTVal_16s_I (Ipp16s * pSrcDst, int len,
   Ipp16s gtThreshold, Ipp16s gtVal, Ipp16s ltThreshold, Ipp16s ltVal);
```

Description

Vector GTLT threshold with value replacement – compares each element of a vector against upper and lower bounds (gtThreshold and ltThreshold, respectively). Replaces elements below the lower bound with the value of the variable ltVal, and replaces elements above the upper bound with the value of the variable gtVal.

$$pDst[i] = \begin{cases} gtVal, & pSrc[i] > gtThreshold \\ ltVal, & pSrc[i] < ltThreshold \\ pSrc[i], & \text{otherwise} \end{cases} \qquad 0 \le i < len$$

Input Arguments

- pSrc, pSrcDst – pointer to the vector to which the thresholds are applied
- len – number of elements contained in the input and output vectors
- gtThreshold – upper bound. gtThreshold must be greater than ltThreshold
- ltThreshold – lower bound
- ltVal – sub–threshold replacement value (replaces elements below ltThreshold)
- gtVal – supra–threshold replacement value (replaces elements above gtThreshold)

Output Arguments

pSrc, pSrcDst – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad argument

Threshold_LT_32s

Prototype

```
IppStatus ippsThreshold_LT_32s (const Ipp32s *pSrc, Ipp32s *pDst,
   int len, Ipp32s threshold);
IppStatus ippsThreshold_LT_32s_I (Ipp32s *pSrcDst, int len, Ipp32s
   threshold);
```

Description

Vector LT threshold – compares each element of a vector to a threshold, then replaces sub-threshold elements with the value of the `threshold` variable.

$$pDst[i] = \begin{cases} threshold, & pSrc[i] < threshold \\ pSrc[i], & otherwise \end{cases} \qquad 0 \le i < len$$

Input Arguments

- `pSrc`, `SrcDst` – pointer to the vector to which the threshold is applied
- `len` – number of elements contained in the input and output vectors (0 < `len` < 65536)
- `threshold` – threshold value

Output Arguments

`pDst`, `pSrcDst` – pointer to the threshold output vector

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pSrcDst`, or `pDst` is NULL
- `ippStsLengthErr` – illegal value for `len`

Vector Thresholding Usage Examples

Example 2-7 illustrates the usage of the `ippsThreshold_GT_16s_I` primitive. The elements of a 100–element vector are compared against a threshold value. Elements larger than threshold are replaced by the threshold value.

Example 2-7 ippsThreshold_GT_16s

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"

int main()
{
      Ipp16s x[100];
      int i;
      ipp16s threshold = 50;

      /* Initialize x and y vector */
      for ( i = 0; i < 100; i ++ ) {
         x[i] = i;
      }

      /* Threshold the vector x */
      ippsThreshold_GT_16s_I(x, 100, threshold);

      /* print out the output */
      for ( i = 0; i < 100; i ++ ) {
         printf("%6d", x[i]);
         if ( (i+1)%5 == 0 ) {
                  printf("\n");
         }
      }

      return(0);
}
```

continued

Vector Statistics

Statistical vector primitives are available that perform a number of common operations, including min, max, mean, standard deviation, biased autocorrelation, unbiased autocorrelation, and cross–correlation. The details for all of these primitives are given next. The "default" function behavior and arguments are described for each primitive. Scaled and in–place primitive variables can be understood easily by applying the behavioral rules given in Chapter 1.

Max_16s

Prototype

```
IppStatus ippsMax_16s(const Ipp16s * pSrc, int len, Ipp16s * pResult);
```

Description

Vector maximum – examines all vector elements and returns the largest (signed). That is:

$$*pResult = max\{pSrc[i]\},\ 0 \le i < len$$

Input Arguments

- `pSrc` – pointer to the vector of data within which the largest element is identified
- `len` – number of elements contained in the input vector

Output Arguments

`pResult` – pointer to the value of the largest element contained in the input vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Min_16s

Prototype

```
IppStatus ippsMin_16s(const Ipp16s * pSrc, int len, Ipp16s * pResult);
```

Description

Vector minimum – examines all vector elements and returns the smallest (signed). That is:

$$*pResult = min\{pSrc[i]\},\ 0 \le i < len - 1$$

Input Arguments

- `pSrc` – pointer to the vector of data within which the smallest element is identified
- `len` – number of elements contained in the input vector

Output Arguments

`pResult` – pointer to the value of the smallest element contained in the input vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Mean_16s

Prototype

```
IppStatus ippsMean_16s(const Ipp16s * pSrc, int len, Ipp16s * pResult);
```

Description

Vector arithmetic mean – computes the arithmetic mean of a data vector. That is:

$$*pResult = \frac{1}{len} \cdot \sum_{k=0}^{len-1} pSrc[k]$$

Input Arguments

- `pSrc` – pointer to the vector of data for which the arithmetic mean is computed
- `len` – number of elements contained in the vector

Output Arguments

`pResult` – pointer to the arithmetic mean of the data contained in the input vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

StdDev_16s

Prototype

```
IppStatus ippsStdDev_16s(const Ipp16s * pSrc, int len, Ipp16s * pResult);
IppStatus ippsStdDev_16s_Sfs(const Ipp16s * pSrc, int len, Ipp16s *
   pResult, int scaleFactor);
```

Description

Vector standard deviation – computes the standard deviation of a data vector. That is:

$$* pResult = \sum_{k=0}^{len-1} (pSrc[k])^2 / len$$

Input Arguments

- `pSrc` – pointer to the vector of data for which the standard deviation is computed
- `len` – number of elements contained in the vector
- `scalefactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pResult` – pointer to the standard deviation of the data contained in the input vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

AutoCorr_16s

Prototype

```
IppStatus ippsAutoCorr_16s(const Ipp16s * pSrc, int srcLen, Ipp16s *
   pDst, int dstLen);
IppStatus ippsAutoCorr_16s_Sfs(const Ipp16s * pSrc, int srcLen, Ipp16s *
   pDst, int dstLen, int scaleFactor);
```

Description

Sample autocorrelation – computes a sample autocorrelation for a vector of data. That is:

$$pDst[n] = \sum_{k=0}^{srcLen-1} pSrc[k] \times pSrc[k+n] \qquad 0 \le n < dstLen \qquad pSrc[k] = 0 \ \ if \ \ k \ge srcLen$$

Input Arguments

- `pSrc` – pointer to the vector of data for which the sample autocorrelation is computed
- `srcLen` – number of elements contained in the vector of data
- `dstLen` – number of autocorrelation lags to estimate
- `scalefactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst` – pointer to the sample autocorrelation sequence

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

AutoCorr_NormA_16s

Prototype

```
IppStatus ippsAutoCorr_NormA_16s(const Ipp16s * pSrc, int srcLen, Ipp16s
   * pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_16s_Sfs(const Ipp16s * pSrc, int srcLen,
   Ipp16s * pDst, int dstLen, int scaleFactor);
```

Description

Biased autocorrelation – forms a biased estimate of the autocorrelation sequence associated with a data vector. That is:

$$pDst[n] = \begin{cases} \frac{1}{srcLen} \sum_{k=0}^{srcLen-1} pSrc[k] \times pSrc[k+n] & 0 \le n < dstLen \\ 0 & srcLen \le n < dstLen \end{cases}$$

$$pSrc(k) = 0 \quad if \quad k \ge srcLen$$

NOTE. *The quality of the autocorrelation estimate will improve as the number of samples in the data vector increases relative to the number of lags computed. In fact, the biased autocorrelation estimate will become asymptotically unbiased as the number of elements in the vector becomes arbitrarily large. As a rule, the number of points contained in the data vector should be at least ten times the number of desired autocorrelation lags. That is, for computation of 10 lags, the data vector should contain at least 100 samples.*

Input Arguments

- `pSrc` – pointer to the vector of data for which the biased autocorrelation sequence is estimated
- `srcLen` – number of elements contained in the vector of data
- `dstLen` – number of autocorrelation lags to estimate (typically *`dstLen`* « *`srcLen`* – see Note)
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst` – pointer to the biased estimate of the autocorrelation sequence

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

AutoCorr_NormB_16s

Prototype

```
IppStatus ippsAutoCorr_NormB_16s(const Ipp16s *pSrc, int srcLen, Ipp16s
   *pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_16s_Sfs(const Ipp16s *pSrc, int srcLen,
   Ipp16s *pDst, int dstLen, int scaleFactor);
```

Description

Unbiased autocorrelation – forms an unbiased estimate of the autocorrelation sequence associated with a data vector. That is:

$$pDst[n] = \begin{cases} \left[\frac{1}{(srcLen - n)}\right] \sum_{k=0}^{srcLen-1} pSrc(k) \times pSrc[k+n] & 0 \le n < dstLen \\ 0 & srcLen \le n < dstLen \end{cases}$$

$$pSrc(k) = 0 \quad if \quad k \ge srcLen$$

Input Arguments

- `pSrc` – pointer to the vector of data for which the unbiased autocorrelation sequence is estimated
- `srcLen` – number of elements contained in the vector of data
- `dstLen` – number of autocorrelation lags to estimate (typically $dstLen \ll srcLen$ – see Note under "AutoCorr_NormA_16s")
- `scalefactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst` – pointer to the unbiased estimate of the autocorrelation sequence

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

CrossCorr_16s

Prototype

```
IppStatus ippsCrossCorr_16s(const Ipp16s * pSrc1, int src1Len, const
   Ipp16s * pSrc2, int src2Len, Ipp16s * 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);
```

Description

Estimates a set of cross–correlation lags for two vectors. That is:

$$pDst[n] = \sum_{k=0}^{src1Len-1} pSrc1[k] \cdot pSrc2[k+n+lowLag] \qquad 0 \le n < dstLen$$

pSrc2[k]=0 if k>=src2Len

Input Arguments

- `pSrc1`, `pSrc2` – pointers to the vectors of data for which the cross correlation lags are estimated
- `src1Len`, `src2Len` – number of elements contained in the vectors for which the cross correlation lags are estimated
- `dstLen` – number of cross correlation lags to compute
- `lowLag` – lowest cross correlation lag to compute; the primitive computes the set of cross correlation lags, τ, such that $lowLag \le \tau < lowLag + DstLen$
- `scalefactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pDst` – pointer to the estimated cross correlation sequence

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Vector Statistics Usage Examples

Example 2-8 illustrates the usage of the `ippsMean_16s` primitive.

Example 2-8 ippsMean_16s

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"

int main()
{
        Ipp16s x[100], result;
        int i;

        /* Initialize x */
        for ( i = 0; i < 100; i ++ ) {
          x[i] = i;
        }

       /* Calculate the mean value of vector x */
       ippsMean_16s(x, 100, &result);


       /* print out the output */
       printf("%6d", result);

       return(0);
}
```

Example 2-9 illustrates the usage of the `ippsAutoCorr_16s_Sfs` primitive.

Example 2-9 ippsAutoCorr_16s

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"
#define srcLen 10
#define dstLen 10
#define factor 2

int main()
{
       Ipp16s src[srcLen];
       Ipp16s dst[dstLen];
       int i;

       for ( i = 0; i < srcLen; i ++ ) {
          src[i] = (i – srcLen/2) * 1000;
       }

       /* calculate the autocorrelation */
       ippsAutoCorr_16s_Sfs(src, srcLen, dst, dstLen, factor);


       for ( i = 0; i < dstLen; i ++ ) {
           printf("%6d", dst[i]);
           if ( i != 0 && i%5 == 0 ) {
                     printf("\n");
             }
       }

       return(0);
}
#undef srcLen
#undef dstLen
#undef factor
```

Vector Measure

Primitives are available to compute several of the most common vector measures, including the L_1, L_2, and L_∞ norms. For each primitive, both absolute and differencing versions are available. The details for these primitives are given next. The "default" function behavior and arguments are described for each primitive. The scaled primitive variables can be understood easily by applying the behavioral rules given in Chapter 1.

NormDiff_L1_16s32s

Prototype

```
IppStatus ippsNormDiff_L1_16s32s(const Ipp16s * pSrc1, const Ipp16s *
   pSrc2, int len, Ipp32s * pResult);
IppStatus ippsNormDiff_L1_16s32s_Sfs(const Ipp16s * pSrc1, const Ipp16s *
   pSrc2, int len, Ipp32s * pResult, int scaleFactor);
```

Description

Vector L_1 difference – computes the L_1 measure of difference between two vectors. That is:

$$*pResult = \sum_{k=0}^{len-1} |pSrc1[k] - pSrc2[k]|$$

Input Arguments

- `pSrc1`, `pSrc2` – pointers to the input vectors for the difference calculation
- `len` – number of samples contained in the input vectors
- `scalefactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pResult` – pointer to the result of the difference calculation

Returns

- `ippStsNoErr` – no error

- ippStsBadArgErr – bad arguments

NormDiff_L2_16s32s

Prototype

```
IppStatus ippsNormDiff_L2_16s32s(const Ipp16s * pSrc1, const Ipp16s *
   pSrc2, int len, Ipp32s * pResult);
IppStatus ippsNormDiff_L2_16s32s_Sfs(const Ipp16s * pSrc1, const Ipp16s *
   pSrc2, int len, Ipp32s * pResult, int scaleFactor);
```

Description

Vector L_2 difference – computes the L_2 measure of difference between two vectors. That is:

$$*pResult = \left(\sum_{k=0}^{len-1} |pSrc1[k] - pSrc2[k]|^2 \right)^{1/2}$$

Input Arguments

- pSrc1, pSrc2 – pointers to the input vectors for the difference calculation
- len – number of samples contained in the input vectors
- scalefactor – saturation fixed scalefactor (scaled primitive only)

Output Arguments

pResult – pointer to the result of the difference calculation

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments

NormDiff_Inf_16s32s

Prototype

```
IppStatus ippsNormDiff_Inf_16s32s(const Ipp16s * pSrc1, const Ipp16s *
   pSrc2, int len, Ipp32s * pResult);
IppStatus ippsNormDiff_Inf_16s32s_Sfs(const Ipp16s * pSrc1, const Ipp16s
   * pSrc2, int len, Ipp32s * pResult, int scaleFactor);
```

Description

Vector L_∞ difference – computes the L_∞ measure of difference between two vectors. That is:

$$*pResult = \max_{0 \le k < len}(|pSrc1[k] - pSrc2[k]|)$$

Input Arguments

- `pSrc1`, `pSrc2` – pointers to the input vectors for the difference calculation
- `len` – number of samples contained in the input vectors
- `scalefactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pResult` – pointer to the result of the difference calculation

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Norm_L1_16s32s

Prototype

```
IppStatus ippsNorm_L1_16s32s(const Ipp16s * pSrc, int len, Ipp32s *
   pResult);
IppStatus ippsNorm_L1_16s32s_Sfs(const Ipp16s * pSrc, int len, Ipp32s *
   pResult, int scaleFactor);
```

Description

Vector L_1 norm – computes the L_1 norm of a vector. That is:

$$*pResult = \sum_{k=0}^{len-1} |pSrc(k)|$$

Input Arguments

- `pSrc` – pointer to the input vector for the norm computation
- `len` – number of samples contained in the input vector
- `scalefactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pResult` – pointer to the value of the L_1 norm for the input vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Norm_L2_16s32s

Prototype

```
IppStatus ippsNorm_L2_16s32s(const Ipp16s * pSrc, int len, Ipp32s *
   pResult);
IppStatus ippsNorm_L2_16s32s_Sfs(const Ipp16s * pSrc, int len, Ipp32s *
   pResult, int scaleFactor);
```

Description

Vector L_2 norm – computes the L_2 norm of a vector. That is:

$$*pResult = \left(\sum_{k=0}^{len-1} |pSrc(k)|^2 \right)^{1/2}$$

Input Arguments

- `pSrc` – pointer to the input vector for the norm computation
- `len` – number of samples contained in the input vector
- `scalefactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pResult` – pointer to the value of the L_2 norm for the input vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Norm_Inf_16s32s

Prototype

```
IppStatus ippsNorm_Inf_16s32s(const Ipp16s * pSrc, int len, Ipp32s *
   pResult);
IppStatus ippsNorm_Inf_16s32s_Sfs(const Ipp16s * pSrc, int len, Ipp32s *
   pResult, int scaleFactor);
```

Description

Vector L_∞ norm – computes the L_∞ norm of a vector. That is:

$$*pResult = \max_{0 \le k < len} |pSrc(k)|$$

Input Arguments

- `pSrc` – pointer to the input vector for the norm computation
- `len` – number of samples contained in the input vector
- `scaleFactor` – saturation fixed scalefactor (scaled primitive only)

Output Arguments

`pResult` – pointer to the value of the L_∞ norm for the input vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Up/Down Sampling Primitives

UpSampleSize

Prototype

```
IppStatus ippsUpSampleSize(int srcLen, int sampleFactor, int phase, int *
   pDstLen);
IppStatus ippsUpSample_16s(const Ipp16s * pSrc, int srcLen, int *
   pSrcDstPhase, Ipp16s * pDst, int sampleFactor);
IppStatus ippsDownSampleSize(int srcLen, int sampleFactor, int phase, int
   * pDstLen);
IppStatus ippsDownSample_16s(const Ipp16s * pSrc, int srcLen, int *
   pSrcDstPhase, Ipp16s * pDst, int sampleFactor);
```

Description

- Calculate the length of the up-sample output signal.
- Increase the target signal's sampling rate.
- Calculate the length of the down-sample output signal.
- Decrease the target signal's sampling rate.

Input Arguments

- `pSrc` – pointer to the input vector, whose sampling rate will be changed
- `srcLen` – the length of the input vector
- `dstLen` – the length of the output vector
- `phase` – the parameter, which determine the relative position of the input vector and the output vector
- `pSrcDstPhase` – pointer to the parameter, which determines the relative position of the input vector and output vector
- `sampleFactor` – sample factor, which indicates the relationship between the old frequency and the new one

Output Arguments

- `pDst` – pointer to the output vector, which is the result of re-sampling

- `pDstLen` – pointer to the length of the result vector
- `pSrcDstPhase` – pointer to the updated parameter, which contains phase information

Returns

- `ippStsNoErr` –no error
- `ippStsBadArgErr` – bad arguments

Signal Generation 3

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) on Intel® PCA Processors with Intel® Wireless MMX™ Technology (PCA processors with MMX™) that are available for generating sinusoidal, triangular, and pseudo-random sequences of samples.

For several signal generator types, Intel® IPP offers both initialized and direct versions. Direct signal generators require only a single call to the library in order to obtain a desired sequence of samples. On the other hand, at the expense of two additional initialization function calls, initialized signal generators offer the benefit of improved execution efficiency over the direct versions. While they are functionally identical from an input/output point of view, the initialized and direct functions offer the user an opportunity to trade initialization overhead library calls for improved speed of execution.

In the interest of simplicity and consistency, the mathematical expressions given in this chapter to describe the behavior of each function generator represent the particular case of the non-in-place and non-scaled function variable (the so-called "default" function version). The user should be aware that the behavior of any scaled and/or in-place variables can be understood easily by applying to the default behavioral specification the generic in-place and scaled function behavioral rules that are given in Chapter 1. Moreover, several of the function generators described in this chapter make use of the Qm.n integer fixed-point representation of floating point function parameters. Complete details on the Qm.n format are given in Chapter 1.

The rest of this chapter is organized as follows. First, deterministic signal generators are described. Sections "Sinusoidal Signals" and "Triangular Signals", respectively, are concerned with the sinusoidal and triangular signal generators. Then, pseudo-random signal generation is addressed. Sections "Uniformly Distributed Pseudo-Random Signals" and "Normally Distributed Pseudo-Random Signals", respectively, provide details on the random signal generators that synthesize output sequences for which samples are selected from uniform and Gaussian distributions. Finally, section "Signal Generation Usage Examples" provides example 'C' language source listings that demonstrate the usage of the Intel® IPP signal generators.

Sinusoidal Signals

This section describes the primitives available for generation of sinusoids, including both the initialized and direct variables. While the initialized and direct primitive variables are functionally identical from an input/output point of view, the initialized generator internally achieves increased execution efficiency relative to the direct primitive, by exploiting an iterative computational technique. The increased efficiency, however, comes at the expense of two overhead function calls into the library during initialization time. On the other hand, the direct version offers a simplified API that consists of only a single call into the library. By employing a computational scheme based on linear interpolation, however, the direct variable incurs a slight performance penalty.

For both the initialized and direct function variables, the behavior of the sinusoidal signal generation primitive is described in terms of the magnitude, frequency, and phase parameters, i.e.,

$$s(n) = magQ15 \cdot \cos(2\pi \cdot freqQ15 \cdot n + phaseQ15)$$

where the parameter *magQ15* specifies the magnitude (Q15), the parameter *freqQ15* specifies the sinusoidal frequency normalized by the sample frequency (Q15), the parameter *phaseQ15* specifies the phase, and the parameter *n* represents the integer index of discrete time (sample index).

Both the initialized and direct usage models are straightforward. The direct primitive requires a single call to the primitive with a complete set of parameters. The direct primitive returns a vector of sinusoidal samples synthesized according to the input parameters. On the other hand, for the initialized primitive, the user must allocate, initialize, maintain, and ultimately free an external state variable. Therefore, the initialized generator requires the combined use of three primitives. In particular, the application code making use of an initialized sinusoidal signal generation primitive must perform the following sequence of operations:

1. Call the primitive `ippsToneGetStateSizeQ15_16s()` for API alignment to determine the number of bytes required for the state variable structure `IppToneState_16s`.
2. Invoke the memory allocation procedure(s) appropriate for the particular target operating system (OS) to request a buffer of the size returned by the size function in step 1, and then set the signal generation state buffer pointer such that it references the newly allocated block of memory.
3. Call the primitive `ippsToneInitQ15_16s()` to configure the newly allocated state buffer with the desired synthesis parameters.
4. Call the primitive `ippsToneQ15_16s()` to synthesize the desired sequence of output samples.
5. Free the memory allocated to the state buffer once signal synthesis has been completed and the state variable is no longer required.

Complete details for both the initialized and direct variables of the sinusoidal synthesis primitives are given next. In addition, example 'C' language source listings are given at the end of the chapter that demonstrate the usage of signal generation primitives.

ToneGetStateSizeQ15_16s

Prototype

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

Description

Initialized tone generator state variable size – computes the size, in bytes, of the structure `IppToneState_16s`.

Input Arguments

None

Output Arguments

`pToneStateSize` – pointer to an integer that indicates the size of the structure `IppToneState_16s`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

ToneInitQ15_16s

Prototype

```
IppStatus ippsToneInitQ15_16s(IppToneState_16s * pToneState, Ipp16s
   magQ15, Ipp16s freqQ15, Ipp32s phaseQ15);
```

Description

Given a set of sinusoidal parameters (magnitude, frequency, and phase), initializes the state variable structure `IppToneState_16s`.

Input Arguments

- `magQ15` – desired peak sinusoidal magnitude, in Q0.15 format. The value must be non-negative, i.e., $0 \le magQ15 < 32768$. For Q0.15 in conjunction with the underlying type `Ipp16s`, this constraint corresponds to a peak magnitude range of $0 \le peak\ magnitude < 1$, and an output signal amplitude range of $-1 < amplitude < 1$.
- `freqQ15` – desired sinusoidal frequency, f, normalized by the sample frequency, f_s, represented in Q0.15 format. In order to satisfy the Nyquist criterion for a sampled sinusoid, it is required that $0 \le freqQ15 \le 16383$. For Q0.15 in conjunction with the underlying type `Ipp16s`, this constraint corresponds to a range for the normalized digital frequency, θ, of $0 \le \theta < \pi$ radians, where $\theta = 2\pi f / f_s$.
- `phaseQ15` – desired unwrapped sinusoidal phase in Q16.15 format, with the value constrained such that $0 < phaseQ15 \le 205886$. For Q16.15 (Q15 in conjunction with the underlying type `Ipp32s`), this corresponds to a range for the unwrapped phase, ϕ, of $0 < \phi < 2\pi$ radians.
- `pToneState` – pointer to the uninitialized state variable structure `IppToneState_16s`

Output Arguments

`pToneState` – pointer to the initialized state variable structure `IppToneState_16s`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

ToneQ15_16s

Prototype

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

Description

Initialized version of the sinusoidal sequence generator – synthesizes a sequence of sinusoidal samples in accordance with the parameters given in the `IppToneState_16s` state variable structure.

Input Arguments

- `pToneState` – pointer to the initialized state variable structure `IppToneState_16s`
- `len` – number of sinusoidal samples to synthesize in the output vector

Output Arguments

`pDst` – Pointer to the vector that contains the sinusoidal output sequence. Given the Q15 representation of the sinusoidal synthesis parameters, for the amplitude range $0 \le magQ15 < 32768$, the corresponding output amplitudes will be in the range $-32767 \le pDst[i] < 32768$.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

ToneQ15_Direct_16s

Prototype

```
IppStatus ippsToneQ15_Direct_16s(Ipp16s * pDst, int len, Ipp16s magQ15,
   Ipp16s freqQ15, Ipp32s phaseQ15);
```

Description

Synthesizes a sequence of sinusoidal samples directly given a set of desired sinusoidal parameters (magnitude, frequency, and phase).

Input Arguments

- `len` – number of sinusoidal samples to synthesize in the output vector

- magQ15 – desired peak sinusoidal magnitude, in Q0.15 format. The value must be non-negative, i.e., $0 \le magQ15 < 32768$. For Q15 in conjunction with the underlying type Ipp16s, this constraint corresponds to a peak magnitude range of $0 \le peak\ amplitude < 1$, and an output signal amplitude range of $-1 < amplitude < 1$.
- freqQ15 – desired sinusoidal frequency, f, normalized by the sample frequency, f_s, represented in Q0.15 format. In order to satisfy the Nyquist criterion for a sampled sinusoid, it is required that $0 \le freqQ15 \le 16383$. For Q15 in conjunction with the underlying type Ipp16s, this constraint corresponds to a range for a normalized digital frequency, θ, of $0 \le \theta < \pi$ radians, where $\theta = 2\pi f / f_s$.
- phaseQ15 – desired unwrapped sinusoidal phase in Q16.15 format, with the value constrained such that $0 \le phaseQ15 < 205887$. For Q16.15 (Q15 in conjunction with the underlying type Ipp32s), this corresponds to a range for the unwrapped phase, ϕ, of $0 < \phi < 2\pi$ radians.

Output Arguments

pDst – pointer to the vector that contains the sinusoidal output sequence

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments

Triangular Signals

This section describes the primitives available for generation of triangular sequences, including both the initialized and direct variables. While the initialized and direct primitive variables are functionally identical from an input/output point of view, the initialized generator internally achieves increased execution efficiency relative to the direct primitive by exploiting an iterative computational technique. The increased efficiency, however, comes at the expense of two overhead function calls into the library during initialization time. On the other hand, the direct version offers a simplified API that consists of only a single call into the library. By employing a computational scheme based on linear interpolation, however, the direct variable incurs a slight performance penalty.

For both the initialized and direct function variables, the triangular signal generation primitive behavior is defined in terms of a parametric, periodic triangular kernel, i.e.,

$$pDst[i] = magQ15 \cdot T_h(\theta(i)), \quad i = 0, 1, \ldots, len - 1$$

where the kernel is defined as

$$T_h(\theta(i)) = \begin{cases} -\frac{2}{h}\left(\theta(i) - \frac{h}{2}\right), & 0 \le \theta(i) \le h \\ \frac{2}{2\pi - h}\left(\theta(i) - \frac{2\pi + h}{2}\right), & h < \theta(i) \le 2\pi \end{cases}$$

the parameter $magQ15$ specifies the magnitude (Q15), the parameter i represents the integer index of discrete time (sample index), and the asymmetry, h, is given in terms of the Q0.15 parameter $asymQ15$ as follows

$$h = asymQ15 + \pi$$

Periodic extension of the triangular kernel is controlled by the 2π-periodic sequence, $\theta(i)$,defined as

$$\theta(i) = \omega(i) - 2\pi K$$

where the Q0.15 parameters $freqQ15$ and $phaseQ15$, respectively, control fundamental frequency and phase, i.e.,

$$\omega(i) = 2\pi \cdot freqQ15 \cdot i + phaseQ15$$

and the integer K, defined as

$$K = \left\lfloor \frac{\omega(i)}{2\pi} \right\rfloor$$

forces a periodicity on the sequence $\theta(i)$. Here, the operation $\lfloor \circ \rfloor$ denotes rounding towards zero. While the foregoing expressions describe in detail the behavior of the triangular primitive, the API is straightforward, and requires only that the user specify the triangular sequence in terms of the magnitude, frequency, phase, and asymmetry parameters. Valid ranges for each of the parameters are given below under the detailed function descriptions.

As far as usage models are concerned, the direct primitive requires a single call into the library with a complete set of parameters. The direct primitive returns a vector of triangular samples synthesized according to the input parameters. On the other hand, for the initialized primitive, the user must allocate, initialize, maintain, and ultimately free an external state variable. Therefore,

the initialized generator requires the combined use of three primitives. In particular, the application code making use of an initialized triangular signal generation primitive must perform the following sequence of operations:

1. Call the primitive `ippsTriangleGetStateSizeQ15_16s()` for API alignment to determine the number of bytes required for the state variable structure `IppTriangleState_16s`.
2. Invoke the memory allocation procedure(s) appropriate for the particular target operating system (OS) to request a buffer of the size returned by the size function in step 1, and then set the signal generation state buffer pointer such that it references the newly allocated block of memory.
3. Call the primitive `ippsTriangleInitQ15_16s()` to configure the newly allocated state buffer with the desired synthesis parameters.
4. Call the primitive `ippsTriangleQ15_16s()` to synthesize the desired sequence of output samples.
5. Free the memory allocated to the state buffer once signal synthesis has been completed and the state variable is no longer required.

Complete details for both the initialized and direct variables of the triangular synthesis primitives are given next. In addition, example 'C' language source listings are given at the end of the chapter that demonstrate the usage of signal generation primitives.

TriangleGetStateSizeQ15_16s

Prototype

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

Description

Triangular generator state variable size for the initialized version of the primitive – computes the size, in bytes, of the structure `IppTriangleState_16s`

Input Arguments

none

Output Arguments

`pTriangleStateSize` – pointer to an integer that indicates the size of the state variable structure `IppTriangleState_16s`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

TriangleInitQ15_16s

Prototype

```
IppStatus ippsTriangleInitQ15_16s(IppTriangleState_16s * pTriangleState,
   Ipp16s magQ15, Ipp16s freqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

Description

Given a set of triangular sequence parameters (magnitude, fundamental frequency, phase, and asymmetry), initializes the state variable structure `IppTriangleState_16s`.

Input Arguments

- `magQ15` – desired peak magnitude, in Q0.15 format. The value must be non-negative, i.e., $0 \le magQ15 < 32768$. For Q15 in conjunction with the underlying type `Ipp16s`, this constraint corresponds to a peak magnitude range of $0 \le peak\ magnitude < 1$, and an output signal amplitude range of $-1 < amplitude < 1$.
- `freqQ15` – desired fundamental frequency, f_0, for the triangular sequence, normalized by the sample frequency, f_s, and represented in Q0.15 format. In order to satisfy the Nyquist criterion, it is required that $0 \le freqQ15 < 16384$. For Q15 in conjunction with the underlying type `Ipp16s`, this constraint corresponds to a range for the normalized fundamental digital frequency, θ_0, of $0 \le \theta_0 < \pi$ radians, where $\theta_0 = 2\pi f_0 / f_s$.
- `phaseQ15` – desired phase in Q16.15 format, with the value constrained such that $0 < phaseQ15 \le 205886$. For Q16.15 (Q15 in conjunction with the underlying type `Ipp32s`), this corresponds to a range for the phase, ϕ, of $0 < \phi < 2\pi$ radians.

- `asymQ15` – asymmetry coefficient in Q16.15 format, with the value constrained such that $-102943 \le asymQ15 \le 102943$. For Q16.15 (Q15 in conjunction with the underlying type `Ipp32s`), this corresponds to a range on the internal asymmetry parameter, h, of $-\pi < h < \pi$, radians.
- `pTriangleState` – pointer to the uninitialized state variable structure `IppTriangleState_16s`

Output Arguments

`pTriangleState` – pointer to the initialized state variable structure `IppTriangleState_16s`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

TriangleQ15_16s

Prototype

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

Description

Initialized version of the triangular sequence generator – synthesizes a sequence of triangular samples in accordance with the parameters given in the `IppTriangleState_16s` state variable structure.

Input Arguments

- `pTriangleState` – pointer to the initialized state variable structure `IppTriangleState_16s`
- `len` – number of triangular samples to synthesize in the output vector

Output Arguments

`pDst` – pointer to the vector that contains the triangular output sequence. Given the Q15 representation of the triangular synthesis parameters, for the magnitude range $0 \le magQ15 < 32768$, the corresponding output amplitudes will be in the range $-32767 \le pDst[i] < 32768$.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

TriangleQ15_Direct_16s

Prototype

```
IppStatus ippsTriangleQ15_Direct_16s(Ipp16s * pDst, int len, Ipp16s
   magQ15, Ipp16s freqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

Description

Direct version of the triangular sequence generator – synthesizes a sequence of triangular samples directly given a set of synthesis parameters (magnitude, fundamental frequency, and phase).

Input Arguments

- `len` – the number of samples to generate
- `magQ15` – desired peak magnitude, in Q0.15 format. The value must be non-negative, i.e., $0 \le magQ15 < 32768$. For Q15 in conjunction with the underlying type `Ipp16s`, this constraint corresponds to a peak magnitude range of $0 \le peak\ magnitude < 1$, and an output signal amplitude range of $-1 < amplitude < 1$.
- `freqQ15` – desired fundamental frequency, f_0, for the triangular sequence, normalized by the sample frequency, f_s, and represented in Q0.15 format. In order to satisfy the Nyquist criterion, it is required that $0 \le freqQ15 \le 16383$. For Q15 in conjunction with the underlying type `Ipp16s`, this constraint corresponds to a range for the normalized fundamental digital frequency, θ_0, of $0 \le \theta_0 < 2\pi$ radians, where $\theta_0 = 2\pi f_0 / f_s$.
- `phaseQ15` – desired phase in Q16.15 format, with the value constrained such that $0 < phaseQ15 \le 205886$. For Q16.15 (Q15 in conjunction with the underlying type `Ipp32s`), this corresponds to a range for the phase, ϕ, of $0 < \phi < 2\pi$ radians.
- `asymQ15` – asymmetry coefficient in Q16.15 format, with the value constrained such that $-102943 \le asymQ15 \le 102943$. For Q16.15 (Q15 in conjunction with the underlying type `Ipp32s`), this corresponds to a range on the internal asymmetry parameter, h, of $-\pi < h < \pi$, radians.

Output Arguments

`pDst` – Pointer to the vector that contains the triangular output sequence. Given the Q15 representation of the sinusoidal synthesis parameters, for the magnitude range $0 \le magQ15 < 32768$, the corresponding output amplitudes will be in the range $-32767 \le pDst[i] < 32768$.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Uniformly Distributed Pseudo-Random Signals

This section describes the set of three primitives that comprises a uniformly distributed pseudo-random sequence generator. The output samples are chosen from a uniform distribution (i.e., all possible sample values occur with equal likelihood) on the interval $[low, high-1]$, where `low` and `high` are primitive parameters. The samples of the pseudo-random number sequence are determined by the generator state, which can be controlled using the `seed` parameter. This means that particular sequences are repeatable.

The generator operates with an "initialized" usage model similar to the initialized sinusoidal and triangular function generators. The user must allocate, initialize, maintain, and ultimately free an external state variable. In particular, the application code must perform the following sequence of operations:

1. Call the primitive `ippsRandUniformGetSize_16s()` to determine the number of bytes required for the state variable structure `IppRandUniformState_16s`.
2. Invoke the memory allocation procedure(s) appropriate for the particular target operating system (OS) to request a buffer of the size returned by the size function in step 1, and then set the signal generation state buffer pointer such that it references the newly allocated block of memory.
3. Call the primitive `ippsRandUniformInit_16s()` to configure the newly allocated state buffer with the desired synthesis parameters.
4. Call the primitive `ippsRandUniform_16s()` to synthesize the desired sequence of output samples.
5. Free the memory allocated to the state buffer once signal synthesis has been completed and the state variable is no longer required.

Complete details for the uniform random generator are given next. Example 'C' language source listings are given in the "Signal Generation Usage Examples" section.

RandUniformGetSize_16s

Prototype

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

Description

Uniform sequence generator state variable size – computes the size, in bytes, of the state variable structure `IppRandUniformState_16s`.

Input Arguments

none

Output Arguments

`pRandUniformStateSize` – pointer to an integer that indicates the size of the structure `IppRandUniformState_16s`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

RandUniformInit_16s

Prototype

```
IppStatus ippsRandUniformInit_16s(IppRandUniformState_16s *
   pRandUniformState, Ipp16s low, Ipp16s high, Ipp16s seed);
```

Description

Uniform sequence generator configuration – initializes the state variable structure `ippsRandUniformState_16s` such that a call to the generator primitive will synthesize a pseudo-random sequence starting from the state specified by the seed value and characterized by the specified distribution boundaries.

Input Arguments

- `seed` – pseudo-random sequence seed value; places generator into a known state.
- `low` – lower bound of the interval from which to select the uniformly distributed samples
- `high` – upper bound+1 of the interval from which to select the uniformly distributed samples
- `pRandUniformState` – pointer to the uninitialized state variable structure `IppRandUniformState_16s`

Output Arguments

`pRandUniformState` – pointer to the initialized state variable structure `IppRandUniformState_16s`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

RandUniform_16s

Prototype

```
IppStatus ippsRandUniform_16s(Ipp16s * pDst, int len,
   IppRandUniformState_16s * pRandUniformState);
```

Description

Uniform sequence generator – synthesizes a pseudo-random sequence, the samples of which are chosen from a uniform distribution on the interval $[low, high-1]$. The initial generator state is controlled by value of the parameter seed.

Input Arguments

- `len` – the number of samples to generate
- `pRandUniformState` – pointer to the initialized state variable structure `IppRandUniformState_16s`

Output Arguments

`pDst` – pointer to a vector that contains the pseudo-random output sequence

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Normally Distributed Pseudo-Random Signals

This section describes the set of three primitives that comprises a normally distributed pseudo-random sequence generator. In general, the probability density function (pdf), $p(x)$, associated with a Gaussian random variable, x, is given by

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2},$$

where the parameters μ and σ^2, respectively, represent the mean and variance of the distribution. For the primitives described in this section, the output samples are chosen from a Gaussian distribution with a mean of *mean* and variance of $stdDev^2$, where `mean` and `stdDev` are primitive parameters. The sample values of the pseudo-random number sequence are determined by the generator state, which can be controlled using the `seed` parameter. This means that particular sequences are repeatable.

The generator operates with an "initialized" usage model similar to the initialized sinusoidal and triangular function generators. The user must allocate, initialize, maintain, and ultimately free an external state variable. In particular, the application code must perform the following sequence of operations:

1. Call the primitive `ippsRandGaussGetSize_16s()` to determine the number of bytes required for the state variable structure `IppRandGaussState_16s`.
2. Invoke the memory allocation procedure(s) appropriate for the particular target operating system (OS) to request a buffer of the size returned by the size function in step 1, and then set the signal generation state buffer pointer such that it references the newly allocated block of memory.
3. Call the primitive `ippsRandGaussInit_16s()` to configure the newly allocated state buffer with the desired synthesis parameters.

4. Call the primitive `ippsRandGauss_16s()` to synthesize the desired sequence of output samples.
5. Free the memory allocated to the state buffer once signal synthesis has been completed and the state variable is no longer required.

Complete details for the uniform random generator are given next. Example 'C' language source listings are given at the end of the chapter that demonstrate the usage of signal generation primitives.

RandGaussGetSize_16s

Prototype

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

Description

Gaussian sequence generator state variable size – computes the size, in bytes, of the state variable structure `IppRandGaussState_16s`.

Input Arguments

None

Output Arguments

`pRandGaussStateSize` – pointer to an integer that indicates the size of the structure `IppRandGaussState_16s`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

RandGaussInit_16s

Prototype

```
IppStatus ippsRandGaussInit_16s(IppRandGaussState_16s * pRandGaussState,
   Ipp16s mean, Ipp16s stdDev, Ipp32u seed);
```

Description

Gaussian sequence generator configuration – initializes the state variable structure `ippsRandGaussState_16s` such that a call to the generator primitive will synthesize a pseudo-random sequence starting from the state specified by the seed value and characterized by the specified distribution parameters.

Input Arguments

- `seed` – pseudo-random sequence seed value; places generator into a known state
- `mean` – arithmetic mean of the Gaussian output sequence
- `stdDev` – standard deviation of the Gaussian output sequence
- `pRandGaussState` – pointer to the uninitialized state variable structure `IppRandGaussState_16s`

Output Arguments

`pRandGaussState` – pointer to the initialized state variable structure `IppRandGaussState_16s`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

RandGauss_16s

Prototype

```
IppStatus ippsRandGauss_16s(Ipp16s * pDst, int len,
   IppRandGaussState_16s * pRandGaussState);
```

Description

Gaussian sequence generator – synthesizes a pseudo-random sequence, the samples of which are chosen from a Gaussian distribution having mean and variance, respectively, of *mean* and $stdDev^2$. The initial generator state is controlled by value of the parameter seed.

Input Arguments

- `len` – the number of samples to generate
- `pRandGaussState` – pointer to the initialized state variable structure `IppRandGaussState_16s`

Output Arguments

`pDst` – pointer to a vector that contains the pseudo-random output sequence

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Signal Generation Usage Examples

This section provides 'C' language source listings that illustrate the usage of the various signal generation primitives.

Uniformly Distributed Pseudo-Random Sequence

The source code example below illustrates the usage of the `ippsRandUniform_16s` primitive. In the example, a vector containing 1024 samples is synthesized, and its samples are chosen from a uniform distribution on the semi-open interval [-1, 255).

The example, although specific to the uniform pseudo-random primitive, also illustrates the generic calling sequence required for the other initialized signal generation primitives. In particular, it shows that the application must:

1. Call the size function (for example, `ippsRandUniformGetSize_16s`) in order to determine the number of bytes required for the state variable structure.
2. Invoke a memory allocation procedure(s) appropriate for the particular target operating system (OS) to request the number of bytes returned by the size function in step 1, and then set the signal generation state variable pointer such that it points to the newly allocated block of memory.
3. Call the init function (for example, `ippsRandUniformInit_16s`) to configure the newly allocated state variable structure with the desired parameters.
4. Call the sample generation function (for example, `ippsRandUniform_16s`) to generate the desired output signal.
5. Free the memory allocated to the state variable structure once signal generation has been completed and the state variable is no longer required.

Example 3-1 ippsRandUniform_16s Usage

```
include <stdio.h>
#include <malloc.h>
#include "ippdefs.h"
#include "ippSP.h"

int main()
{
    Ipp16s * pDst;
    Ipp16s low=-1, high=255;
   Ipp16s seed=0;
    IppRandUniformState_16s *pRandUniformState;
    int    RandUniformStateSize;
    int    i;
    pDst  =  (Ipp16s *)malloc(1024 * sizeof(Ipp16s));

    /* Calculate the memory size needed */
    ippsRandUniformGetSize_16s(&RandUniformStateSize);

    /* Allocate memory for the corresponding structure */
    pRandUniformState = (IppRandUniformState_16s *)malloc(RandUniformStateSize);

    /* Do structure initializing */
    ippsRandUniformInit_16s(pRandUniformState,low,high,seed);

    /* Call sample-generating functions to generate the corresponding samples */
    ippsRandUniform_16s(pDst,1024,pRandUniformState);

    free(pRandUniformState);
    free(pDst);

    return(0);
}
```

Filtering 4

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) that are concerned with digital filtering. The available filter types include the following:

- Finite Impulse Response (FIR)
- Infinite Impulse Response (IIR)
- Biquad IIR
- Least Mean Square (LMS) adaptive FIR

In the interest of simplicity and consistency, the mathematical expressions given in this chapter to describe the behavior of each filter primitive represent the particular case of the non-in-place and non-scaled variable (the so-called "default" version). The user should be aware that the behavior of any scaled and/or in-place variables can be understood easily by applying to the default behavioral specification the generic in-place and scaled function behavioral rules that are given in Chapter 1. Moreover, several of the filters described in this chapter make use of the Qm.n integer fixed-point representation of floating-point parameters. Complete details on the Qm.n format are given in Chapter 1.

The rest of this chapter is organized as follows. First, the "FIR Filters" section describes FIR filters. Next, the "IIR Filters" section addresses IIR filters. Section "Biquad IIR Filters" is on biquad IIR filters. For each of the FIR, IIR, and biquad IIR primitives, both single sample and block versions are available. The "LMS FIR Adaptive Filter" section describes LMS adaptive FIR filters. Finally, the "Filtering Usage Examples" section provides example C language source listings that demonstrate the usage of the Intel® IPP digital filtering primitives.

FIR Filters

This section describes the FIR filtering primitives, including both block and single sample variables. An FIR filter is a discrete-time linear system for which the value of the current output sample can be determined by computing a weighted sum of the current and past input samples. In particular, the operation of an FIR filter can be described in terms of the time-domain difference equation

$$y(n) = \sum_{k=0}^{K} b_k x(n-k)$$

where $x(n)$ is the input sequence, $y(n)$ is the output sequence, b_k are the filter coefficients (aka "taps"), K is the filter order, and n is the discrete-time (sample) index. The impulse response associated with this system, $h(n)$, can be obtained by exciting the system with a digital impulse. For example:

$$x(n) = \delta(n) = \begin{cases} 1, & n = 0 \\ 0, & otherwise \end{cases}$$

such that

$$h(n) = \sum_{k=0}^{K} b_k \delta(n-k)$$

Clearly, the impulse response of the FIR filter is given by the filter taps. By computing the z transform of the impulse response, it can be easily shown that the system transfer function for the FIR filter is given by

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{k=0}^{K} b_k z^{-k}$$

The foregoing expressions describe FIR filter behavior in general. Characteristics unique to the Intel® IPP FIR primitives are described next. For both the block and single-sample variables, the floating point filter coefficients, b_k, are represented using Q15 parameters. That is:

$$pTapsQ15(k) = b_k \cdot 32768, \quad 0 \le k < tapsLen$$

Because the underlying type is `Ipp16s`, the filter coefficients must be normalized such that $|b_k| \le 1$ prior to the Q0.15 scaling. In addition to Q0.15 coefficient representations, both the block and single-sample FIR primitives require external state buffers (filter memories). This API architecture supports sequential filtering of contiguous samples using repeated calls to the FIR primitives without loss of internal filter state. Transient filter responses are avoided at block or single-sample boundaries by providing a state initialization mechanism for the filter. If the filter is initialized at the start of each new block to the state that was reached at the end of the previous block, then a steady-state filter response is maintained when filtering a long data record on a block-by-block or even a sample-by-sample basis. Users are responsible for filter memory management, including allocation, initialization, and, de-allocation. Application code making use of the FIR primitives should adhere to the following usage model:

1. *Initialization:* prior to calling the primitive for the first time, invoke the memory allocation procedure(s) appropriate for the particular target operating system (OS) to request a buffer of the size $2K+2$ (twice the length of the coefficient vector). Next, set the filter memory pointer, `pDelayLine`, such that it references the newly allocated block of memory, and initialize the contents of the buffer to zeros. Set to zero the delay line index value, `pDelayLineIndex`. Finally, load the tap vector, `pTapsQ15`, with appropriately scaled Q0.15 filter coefficients.
2. *Filtering:* after initialization has been completed and the data to be filtered has been loaded into the input vector, the application should call the desired primitive: (`ippsFIR_Direct_16s`, `ippsFIROne_Direct_16s` or one of the variables). If contiguous samples from a long sequence are processed either in blocks or sample-by-sample using repeated primitive calls, the application should not modify the delay line memory or the delay line index in between successive calls to the primitive, unless a transient response is desired in the output. Once filtering has been completed, the user is responsible for de-allocating the delay line buffer memory.

   Complete details for both the single sample and block FIR filtering primitives are given next. In addition, example 'C' language source listings are given at the end of the chapter that demonstrate their usage.

FIR_Direct_16s

```
IppStatus ippsFIR_Direct_16s(const Ipp16s * pSrc, Ipp16s * pDst, int
   sampLen, const Ipp16s * pTapsQ15, int tapsLen, Ipp16s * pDelayLine,
   int * pDelayLineIndex);
IppStatus ippsFIR_Direct_16s_I(Ipp16s * pSrcDst, int sampLen, const
   Ipp16s * pTapsQ15, int tapsLen, Ipp16s * pDelayLine, int *
   pDelayLineIndex);
IppStatus ippsFIR_Direct_16s_Sfs(const Ipp16s * pSrc, Ipp16s * pDst, int
   sampLen, const Ipp16s * pTapsQ15, int tapsLen, Ipp16s * pDelayLine,
   int * pDelayLineIndex, int scaleFactor);
IppStatus ippsFIR_Direct_16s_ISfs(Ipp16s * pSrcDst, int sampLen, const
   Ipp16s * pTapsQ15, int tapsLen, Ipp16s * pDelayLine, int *
   pDelayLineIndex, int scaleFactor);
```

Description

Block FIR – applies the FIR filter defined by the coefficient vector `pTapsQ15` to a vector of input data.

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the vector of input samples to which the filter is applied
- `sampLen` – the number of samples contained in both the input and output vectors
- `pTapsQ15` – pointer to the vector that contains the filter coefficients, represented in Q0.15 format (see Chapter 1). Given that $-32768 \le pTapsQ15(k) < 32768, \quad 0 \le k < tapsLen$, the range on the actual filter coefficients is $-1 \le b_k < 1$, and therefore coefficient normalization may be required during the filter design process.
- `tapsLen` – the number of taps, or, equivalently, the filter order + 1
- `pDelayLine` – pointer to the $2 \cdot tapsLen$ -element filter memory buffer (state). The user is responsible for allocation, initialization, and de-allocation. The filter memory elements are initialized to zero in most applications.
- `pDelayLineIndex` – pointer to the filter memory index that is maintained internally by the primitive. The user should initialize the value of this index to zero.
- `scaleFactor` – saturation fixed scalefactor (only for the scaled primitive)

Output Arguments

`pDst`, `pSrcDst` – pointer to the vector of filtered output samples

Returns

`ippStsNoErr` – no error

`ippStsBadArgErr` – bad arguments

FIROne_Direct_16s

Prototype

```
IppStatus ippsFIROne_Direct_16s(Ipp16s val, Ipp16s * pResult, const
   Ipp16s * pTapsQ15, int tapsLen, Ipp16s * pDelayLine, int *
   pDelayLineIndex);
IppStatus ippsFIROne_Direct_16s_I(Ipp16s * pValResult, const Ipp16s *
   pTapsQ15, int tapsLen, Ipp16s * pDelayLine, int * pDelayLineIndex);
IppStatus ippsFIROne_Direct_16s_Sfs(Ipp16s val, Ipp16s * pResult, const
   Ipp16s * pTapsQ15, int tapsLen, Ipp16s * pDelayLine, int *
   pDelayLineIndex, int scaleFactor);
IppStatus ippsFIROne_Direct_16s_ISfs(Ipp16s * pValResult, const Ipp16s *
   pTapsQ15, int tapsLen, Ipp16s * pDelayLine, int * pDelayLineIndex, int
   scaleFactor);
```

Description

Single-sample FIR – applies the FIR filter defined by the coefficient vector `pTapsQ15` to a single sample of input data.

Input Arguments

- `val`, `pValResult` – the single input sample to which the filter is applied. A pointer is used for the in-place version.
- `pTapsQ15` – pointer to the vector that contains the filter coefficients, represented in Q0.15 format (see Chapter 1). Given that $-32768 \le pTapsQ15(k) < 32768, \quad 0 \le k < tapsLen$, the range on the actual filter coefficients is $-1 \le b_k < 1$, and therefore coefficient normalization may be required during the filter design process.
- `tapsLen` – the number of taps, or, equivalently, the filter order + 1

- pDelayLine – pointer to the $2 \cdot tapsLen$ -element filter memory buffer (state). The user is responsible for allocation, initialization, and de-allocation. The filter memory elements are initialized to zero in most applications.
- pDelayLineIndex – pointer to the filter memory index that is maintained internally by the primitive. The user should initialize the value of this index to zero.
- scalefactor – saturation fixed scalefactor (only for the scaled primitive)

Output Arguments

pResult, pValResult – pointer to the filtered output sample

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments

IIR Filters

This section describes the IIR filtering primitives, including both block and single sample variables. An IIR filter is a discrete-time linear system for which the value of the current output sample can be determined by computing a weighted sum of the current input sample, past input samples, and past output samples. In particular, the operation of an IIR filter can be described in terms of the time-domain difference equation:

$$y(n) = \sum_{k=0}^{K} b_k x(n-k) - \sum_{m=1}^{M} a_m y(n-m)$$

where $x(n)$ is the input sequence, $y(n)$ is the output sequence, n is the discrete-time (sample) index, and a_m and b_k are the filter coefficients (aka "taps"). The impulse response associated with this system, $h(n)$, can be obtained by exciting the system with a digital impulse. That is:

$$x(n) = \delta(n) = \begin{cases} 1, & n = 0 \\ 0, & otherwise \end{cases}$$

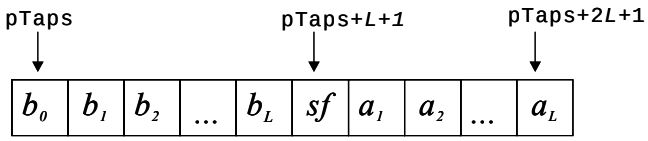
such that

$$h(n) = \sum_{k=0}^{K} b_k \delta(n-k) - \sum_{m=1}^{M} a_m h(n-m)$$

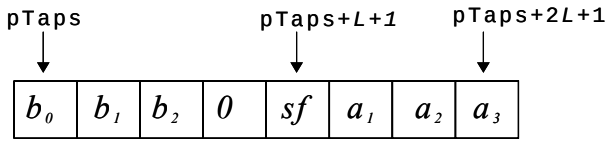
Clearly, the feedback terms cause the impulse response to have an infinite time extent. By computing the z transform of $h(n)$, it can be shown easily that the system transfer function for the IIR filter is given by

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{K} b_k z^{-k}}{1 + \sum_{m=1}^{M} a_m z^{-m}}$$

The foregoing expressions describe IIR filter behavior in general. Characteristics unique to the Intel® IPP IIR primitives are described next. For both the block and single-sample variables, the floating-point filter coefficients b_k and a_m are represented in a combined coefficient vector that is pointed to by the parameter `pTaps` and is organized as follows:



The combined coefficient vector contains $2L+2$ elements, where $L = max\{K, M\}$. Therefore, if $K \neq M$ for a particular filter design, the user must pad with zeros the smaller set of coefficients such that the organization of the combined coefficient vector matches the figure. For example, if $K = 2$ and $M = 3$, then the combined coefficient vector would be arranged as follows:



The specific Q-format used to represent the elements of the IIR coefficient vector is controlled by the scaling coefficient denoted in the above figure by sf. In particular, the actual filter coefficients are related to the elements of the coefficient vector in the following way

$$b_k = pTaps(k) \cdot 2^{-sf}, \quad 0 \le k \le K$$

and

$$a_m = pTaps(m + L + 2) \cdot 2^{-sf}, \quad 0 \le m \le M$$

where $sf = pTaps(L + 1)$, and $sf \ge 0$.

In addition to the user-controlled coefficient scaling, both the block and single-sample IIR primitives require external state buffers (filter memories). External memory makes it is possible to perform sequential filtering of contiguous samples using repeated calls to the IIR primitives without loss of the internal filter state. Thus, a steady-state filter response is maintained when filtering long data records on a block-by-block or even a sample-by-sample basis by avoiding transient responses at the block or single-sample boundaries. Users are responsible for filter memory management, including allocation, initialization, and de-allocation. As a result, application code making use of the IIR primitives should adhere to the following usage model:

1. *Initialization:* prior to calling the primitive for the first time, invoke the memory allocation procedure(s) appropriate for the particular target operating system (OS) to request a buffer of size $L - 1$. Next, set the filter memory pointer, `pDelayLine`, such that it references the newly allocated block of memory, and initialize the contents of the buffer to zeros. Finally, load the tap vector, `pTaps`, with appropriately scaled filter coefficients.
2. *Filtering:* after initialization has been completed and the data to be filtered has been loaded into the input vector, the application should call the desired primitive: (`ippsIIR_Direct_16s`, `ippsIIR_DirectOne_16s`, or one of the variables). If contiguous samples from a long sequence are processed either in blocks or sample-by-sample using repeated primitive calls, the application should not modify the delay line memory in between successive calls to the primitive, unless a transient response is desired in the output. Once filtering has been completed, the user is responsible for de-allocating the delay line buffer memory.

Complete details for both the single sample and block IIR filtering primitives are given next. In addition, example 'C' language source listings are given at the end of the chapter that demonstrate their usage.

IIR_Direct_16s

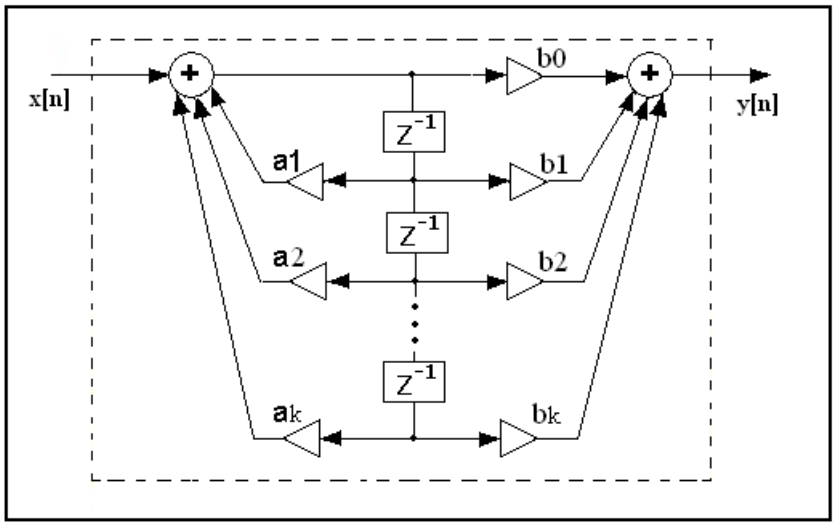
Prototype

```
IppStatus ippsIIR_Direct_16s(const Ipp16s * pSrc, Ipp16s * pDst, int len,
   const Ipp16s * pTaps, int order, Ipp32s * pDelayLine);
IppStatus ippsIIR_Direct_16s_I(Ipp16s * pSrcDst, int len, const Ipp16s *
   pTaps, int order, Ipp32s * pDelayLine);
```

Description

Block IIR – applies the direct form II IIR filter defined by the coefficient vector `pTaps` to a vector of input data. The direct form II IIR filter has the structure shown below, where x and y are, respectively, the vectors of input and output data. See Figure 4-1.

Figure 4-1 Data Flow Diagram of a Typical IIR Direct Form II Filter



Input Arguments

- `pSrc`, `pSrcDst` – pointer to the vector of input samples to which the filter is applied

- `len` – the number of samples contained in both the input and output vectors
- `pTaps` – pointer to the $2L+2$ -element vector that contains the combined numerator and denominator filter coefficients from the system transfer function, $H(z)$. Coefficient scaling and coefficient vector organization should follow the conventions described above. The value of the coefficient scalefactor exponent must be non-negative ($sf \geq 0$).
- `order` – the maximum of the degrees of the numerator and denominator coefficient polynomials from the system transfer function, $H(z)$, That is: $order = max(K, M) - 1 = L - 1$.
- `pDelayLine` – pointer to the L -element filter memory buffer (state). The user is responsible for allocation, initialization, and deallocation. The filter memory elements are initialized to zero in most applications.

Output Arguments

`pDst`, `pSrcDst` – pointer to the vector of filtered output samples

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

IIROne_Direct_16s

```
IppStatus ippsIIROne_Direct_16s(Ipp16s val, Ipp16s * pResult, const
   Ipp16s * pTaps, int order, Ipp32s * pDelayLine);
IppStatus ippsIIROne_Direct_16s_I(Ipp16s * pValResult, const Ipp16s *
   pTaps, int order, Ipp32s * pDelayLine);
```

Description

Single sample IIR – applies the direct form II IIR filter defined by the coefficient vector `pTaps` to a single sample of input data. The direct form II IIR filter has the structure shown below, where x and y are, respectively, the input and output samples. See Figure 4-1.

Input Arguments

- `val`, `pValResult` – the single input sample to which the filter is applied. A pointer is used for the in-place version.

- `pTaps` – pointer to the $2L+2$-element vector that contains the combined numerator and denominator filter coefficients from the system transfer function, $H(z)$. Coefficient scaling and coefficient vector organization should follow the conventions described above. The value of the coefficient scalefactor exponent must be non-negative ($sf \geq 0$).
- `order` – the maximum of the degrees of the numerator and denominator coefficient polynomials from the system transfer function, $H(z)$. That is: $order = max(K, M) - 1 = L - 1$.
- `pDelayLine` – pointer to the L-element filter memory buffer (state). The user is responsible for allocation, intialization, and deallocation. The filter memory elements are initialized to zero in most applications.

Output Arguments

`pResult`, `pValResult` – pointer to the filtered output sample

Returns

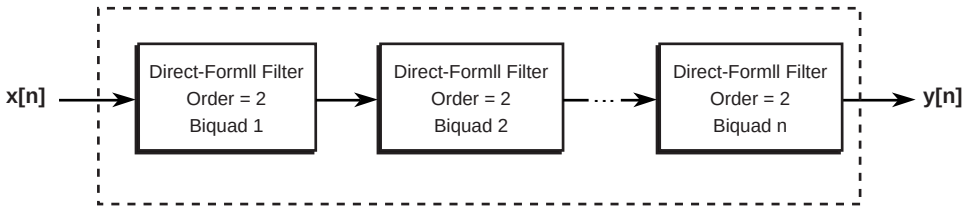
- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Biquad IIR Filters

IIR filters with quantized coefficients are susceptible to stability problems, particularly in fixed-point systems. Therefore, single IIR filters of arbitrary order are often decomposed into equivalent cascades of 2nd-order IIR sections known as biquads. To implement a general IIR filter using a biquad cascade, the filter designer must factor the system transfer function polynomial, $H(z)$, into a cascade of 2nd-order rational polynomials. Although the biquad cascade is

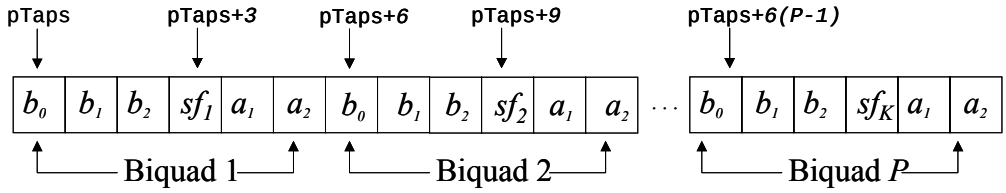
analytically identical to the single filter of higher order, the stability of the biquad filter realization tends to be less sensitive to quantization errors. Intel® IPP biquad IIR primitives are implemented using the direct form II cascade structure shown in Figure 4-2.

Figure 4-2 Direct Form II Cascade Structure



Each of the biquad stages shown in Figure 4-2 implements a direct form II IIR section that conforms to the behavioral description given in the "IIR Filters" section for general IIR filters, and therefore the notation used throughout the remainder of this section is the same as the notation used in the "IIR Filters" section. For both the block and single-sample variables, the floating point filter coefficients b_k and a_m for all of the biquad stages are represented in a combined coefficient vector that is pointed to by the parameter `pTaps` and is organized as follows:

Figure 4-3 Combined Coefficient Vector Organization



The combined coefficient vector contains $6P$ elements, where P is the number of biquad stages in the cascade structure. As with the coefficient vector for the standard IIR primitive, if $K \neq M$ for any constituent filter, then the user must pad with zeros the smaller set of coefficients such that the organization of the combined coefficient vector matches the figure. The specific Q-format used to

represent the elements of the p^{th} biquad section is controlled by the scaling coefficient denoted in the above figure by sf_p, $1 \le p \le P$, where $sf_p \ge 0$. In particular, the actual filter coefficients for the p^{th} biquad section are related to the elements of the coefficient vector in the following way

$$b_k = pTaps(6(p-1)+k) \cdot 2^{-sf_p}, \quad 0 \le k \le K_p$$

and

$$a_m = pTaps(6(p-1)+m+4) \cdot 2^{-sf_p}, \quad 0 \le m \le M_p$$

where $sf_p = pTaps(6(p-1)+3$, K_p is the order of the p^{th} biquad numerator polynomial, and M_p is the order of the p^{th} biquad denominator polynomial.

In addition to the user-controlled coefficient scaling, both the block and single-sample biquad IIR primitives require external state buffers (filter memories). External memory makes it is possible to perform sequential filtering of contiguous samples using repeated calls to the biquad IIR primitives without loss of the internal filter state. Thus, a steady-state filter response is maintained when filtering long data records on a block-by-block or even a sample-by-sample basis by avoiding transient responses at the block or single-sample boundaries. Users are responsible for filter memory management, including allocation, initialization, and de-allocation. As a result, application code making use of the biquad IIR primitives should adhere to the following usage model:

1. *Initialization:* prior to calling the primitive for the first time, invoke the memory allocation procedure(s) appropriate for the particular target operating system (OS) to request a buffer of size $2P$, where P is the number of biquad sections. Next, set the filter memory pointer, `pDelayLine`, such that it references the newly allocated block of memory, and initialize the contents of the buffer to zeros. Finally, load the tap vector, `pTaps`, with appropriately scaled filter coefficients.
2. *Filtering:* after initialization has been completed and the data to be filtered has been loaded into the input vector, the application should call the desired primitive: (`ippsIIR_BiQuadDirect_16s`, `ippsIIROne_BiQuadDirect_16s`, or one of the variables). If contiguous samples from a long sequence are processed either in blocks or sample-by-sample using repeated primitive calls, the application should not modify the delay line memory in between successive calls to the primitive, unless a transient response is desired in the output. Once filtering has been completed, the user is responsible for de-allocating the delay line buffer memory.

Complete details for both the single sample and block biquad IIR filtering primitives are given next. In addition, example 'C' language source listings are given at the end of the chapter that demonstrate their usage.

IIR_BiQuadDirect_16s

Prototype

```
IppStatus ippsIIR_BiQuadDirect_16s(const Ipp16s * pSrc, Ipp16s * pDst,
   int len, const Ipp16s * pTaps, int numBiquad, Ipp32s * pDelayLine);
IppStatus ippsIIR_BiQuadDirect_16s_I(Ipp16s * pSrcDst, int len, const
   Ipp16s * pTaps, int numBiquad, Ipp32s * pDelayLine);
```

Description

Block biquad IIR applies the direct form II biquad IIR cascade defined by the coefficient vector `pTaps` to a vector of input data.

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the vector of input samples to which the filter is applied
- `len` – the number of samples contained in both the input and output vectors
- `pTaps` – pointer to the $6P$ -element vector that contains the combined numerator and denominator filter coefficients from the biquad cascade. Coefficient scaling and coefficient vector organization should follow the conventions described above. The value of the coefficient scalefactor exponent must be non-negative. ($sf_p \geq 0$.).
- `numBiquad` – the number of biquads contained in the IIR filter cascade: (P)
- `pDelayLine` – pointer to the $2P$ -element filter memory buffer (state). The user is responsible for allocation, intialization, and de-allocation. The filter memory elements are initialized to zero in most applications.

Output Arguments

`pDst`, `pSrcDst` – pointer to the vector of filtered output samples

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

IIROne_BiQuad_16s

Prototype

```
IppStatus ippsIIROne_BiQuadDirect_16s(Ipp16s val, Ipp16s * pResult,
   const Ipp16s * pTaps, int numBiquad, Ipp32s * pDelayLine);
IppStatus ippsIIROne_BiQuadDirect_16s_I(Ipp16s * pValResult, const
   Ipp16s * pTaps, int numBiquad, Ipp32s * pDelayLine);
```

Description

Single-sample biquad IIR – applies the direct form II biquad IIR cascade defined by the coefficient vector `pTaps` to a single sample of input data.

Input Arguments

- `val`, `pValResult` – the single input sample to which the filter is applied. A pointer is used for the in-place version.
- `pTaps` – pointer to the $6P$ -element vector that contains the combined numerator and denominator filter coefficients from the biquad cascade. Coefficient scaling and coefficient vector organization should follow the conventions described above. The value of the coefficient scalefactor exponent must be non-negative: ($sf_p \geq 0$).
- `numBiquad` – the number of biquads contained in the IIR filter cascade: (P)
- `pDelayLine` – pointer to the $2P$ -element filter memory buffer (state). The user is responsible for allocation, intialization, and deallocation. The filter memory elements are initialized to zero in most applications.

Output Arguments

`pResult`, `pValResult` – pointer to the filtered output sample

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

LMS FIR Adaptive Filter

This section describes the Least-Mean-Square (LMS) FIR adaptive filter primitives. The FIR adaptive filter is a time-varying linear system for which the taps are adjusted to minimize a measure of the error between the actual output and some desired output. The iterative LMS algorithm minimizes the error sequence in the mean square sense by updating the taps for each new sample that enters the process such that the error is changed in the direction of the negative gradient. During the k^{th} iteration, the output of the FIR filter, $y(n)$, is described in terms of the usual time-domain difference equation, i.e,

$$y(n) = \sum_{m=0}^{M} b_k(m)x(n-m)$$

where $x(n)$ is the input sequence, $y(n)$ is the output sequence, $b_k(m)$ is the k^{th} estimate of the desired taps, M is the filter order, and n is the discrete-time (sample) index. Given a desired output, $d(n)$, the error sequence on the k^{th} iteration is therefore defined as

$$e(n) = d(n) - \sum_{m=0}^{M} b_k(m)x(n-m)$$

For a properly chosen adaptation gain, μ, it can be shown that the expected value of the squared error, $E\{e^2(n)\}$, converges to the minimum mean square bound when the taps, $b_k(m)$,are updated using the LMS gradient algorithm. That is:

$$\begin{bmatrix} b_{k+1}(0) \\ b_{k+1}(1) \\ b_{k+1}(2) \\ \cdots \\ b_{k+1}(M) \end{bmatrix} = \begin{bmatrix} b_k(0) \\ b_k(1) \\ b_k(2) \\ \cdots \\ b_k(M) \end{bmatrix} - 2\mu e(n) \begin{bmatrix} x(n) \\ x(n-1) \\ x(n-2) \\ \cdots \\ x(n-M) \end{bmatrix}$$

The gradient is estimated in terms of the error and the input data. The adaptation gain (or step size) parameter, μ , controls the rate of adaptation. The adaptation gain must be chosen to achieve a reasonable trade-off between rate of convergence and steady-state error.

The foregoing expressions describe LMS FIR adaptive filter behavior in general. For the Intel® IPP LMS adaptive FIR primitives, the floating point filter coefficients, $b_k(m)$, are represented using Q0.15 parameters. That is:

$$pTapsQ15(m) = b_k(m) \cdot 32768, \quad 0 \le m < tapsLen$$

as is the adaptation gain, μ. That is:

$$stepQ15 = \mu \cdot 32768$$

To maximize dynamic range, both the taps and the step size are represented in terms of Q0.15 in conjunction with the underlying type `Ipp32s`: (Q16.15). In addition, external state buffers (filter memories) are required. Users are responsible for filter memory management, including allocation, initialization, and de-allocation. Application code making use of the LMS adaptive FIR primitives should adhere to the following usage model:

1. *Initialization:* prior to calling the primitive for the first time, invoke the memory allocation procedure(s) appropriate for the particular target operating system (OS) to request a buffer of the size $2 \cdot tapsLen = 2M + 2$. Next, set the filter memory pointer, `pDelayLine`, such that it references the newly allocated block of memory, and initialize the contents of the buffer to zeros. Finally, set to zero the delay line index value, `pDelayLineIndex`.
2. *Filtering and LMS adaptation:* after initialization has been completed and the desired inputs and outputs have been selected, the application should call `ippsFIRLMSOne_Direct_16s` or `ippsFIRLMSOne_DirectOne_16s` for each iteration of the LMS adaptation. The user should update the input and desired output on each iteration, but should not modify either the delay line memory or the delay line index. Once filtering and adaptation have been completed, the user is responsible for de-allocating the delay line buffer memory.

Complete details for both of the LMS adaptive FIR filtering primitives are given next. In addition, example 'C' language source listings are given at the end of the chapter that demonstrate their usage.

FIRLMSOne_Direct_16s

Prototype

```
IppStatus ippsFIRLMSOne_Direct_16s(Ipp16s val, Ipp16s valDesire, Ipp16s
   * pResult, Ipp32s * pTapsQ15, int tapsLen, Ipp32s stepQ15, Ipp16s *
   pDelayLine, int * pDelayLineIndex);
IppStatus ippsFIRLMSOne_Direct_16s_I(Ipp16s * pValResult, Ipp16s
   valDesire, Ipp32s * pTapsQ15, int tapsLen, Ipp32s stepQ15, Ipp16s *
   pDelayLine, int * pDelayLineIndex);
```

Description

Adapts FIR filter coefficients using one iteration of the classical LMS algorithm; processes a single input sample, updates the taps, and generates a single output sample.

Input Arguments

- `val`, `pValResult` – the single input sample to which the filter is applied. A pointer is used for the in-place version.
- `valDesire` – desired output in response to the given input
- `pTapsQ15` – pointer to the vector that contains the estimate of the desired filter coefficients obtained during the most recent LMS iteration ($b_k(m), \quad 0 \le m \le M$). The coefficients are represented in Q16.15 format (see Chapter 1). Prior to the first LMS iteration, the taps could be initialized either to a seed value or to zero. In some cases, the rate of convergence can be improved by initializing the coefficients with an approximation to the desired coefficients rather than a vector of zeros.
- `tapsLen` – the number of taps to be updated by the LMS algorithm, or, equivalently, the filter order + 1
- `stepQ15` – the adaptation gain, μ, represented in Q16.15 format (see Chapter 1). To achieve steady-state convergence, the gain should be greater than 0.
- `pDelayLine` – pointer to the $2 \cdot tapsLen$ -element filter memory buffer. The user is responsible for allocation, initialization, and deallocation. The filter memory elements should be initialized to zero prior to the start of an LMS adaptation cycle.
- `pDelayLineIndex` – pointer to the filter memory index that is maintained internally by the primitive. The user should initialize the value of this index to zero. The most recent sample contained in the filter memory is addressed by `pDelayLine+(*pDelayLineIndex)`.

Output Arguments

- `pResult`, `pValResult` – pointer to the actual output that is generated in response to the current input
- `pTapsQ15` – pointer to the vector that contains the updated filter coefficients ($b_{k+1}(m), \quad 0 \le m \le M$), represented in Q16.15 format (see Chapter 1).

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Filtering Usage Examples

This section provides 'C' language source listings that illustrate the usage of the FIR, IIR, biquad IIR, and LMS adaptive filtering primitives.

FIR

The example below illustrates the usage of the scaled FIR filtering primitive, `ippsFIR_Direct_16s_Sfs`. The example code implements a linear-phase, lowpass, 19-th order FIR filter having the coefficients

$$b_k = \{0.08, 0.10492407, 0.17699537, 0.28840385, \ldots, 0.08\}$$

Given that

$$\sum_{k=0}^{19} |b_k| = 10.34$$

the dynamic range on the output, $y(n)$, is $-10.34 \le y(n) < 10.34$, which means that a Q4.11 output representation is required to avoid saturation in the 16-bit output word. Therefore, the scaled FIR primitive is used to accommodate the dynamic range on the output with a scalefactor value of 4.

Example 4-1 ippsFIR_Direct_16s_Sfs Usage

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "ippdefs.h"
#include "ippSP.h"
#define tapsLen 20
#define N 40
#define scaleFactor 4

int main()
{
    int     i;
    Ipp16s  pSrc[N], pDst[N];
    Ipp32s  pDelayLine[tapsLen*2];
    int     delayLineIndex;
    Ipp16s  pTaps[tapsLen];
    float b[tapsLen] =
    { 0.080000000, 0.104924069, 0.176995366, 0.288403847, 0.427076676,
    0.577986499, 0.724779895, 0.851549523, 0.944557926, 0.993726200,
    0.993726200, 0.944557926, 0.851549523, 0.724779895, 0.577986499,
    0.427076676, 0.288403847, 0.176995366, 0.104924069, 0.080000000 };

    /* scale the filter taps to Q15 */
    for ( i = 0; i < tapsLen; i ++ ) {
    b[i] *= (1<<14);
    pTaps[i] = (b[i] > 32767)?32767 : b[i];
    }
    /* random input signal */
    srand(200);
    for ( i = 0; i < N; i ++ )
```

continued

Example 4-1 ippsFIR_Direct_16s_Sfs Usage (continued)

```
    pSrc[i] = rand() - 32768/2;
    delayLineIndex = 0;
    for ( i = 0; i < tapsLen*2; i ++ )
    pDelayLine[i] = 0;
    ippsFIR_Direct_16s_Sfs(pSrc,pDst,N,pTaps,tapsLen,pDelayLine,
        &delayLineIndex, scaleFactor);
    /* display out signal vector */
    for ( i = 0; i < N; i ++ ) {
    printf("%8d", pDst[i]);
    if ( (i+1)%5 == 0 ) {
        printf("\n");
    }
    }
    return(0);
}
```

IIR

The example below illustrates the usage of the `ippsIIR_Direct_16s` primitive.

Example 4-2 ippsIIR_Direct_16s Usage

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"
#define tapsLen 4
#define N 40

int main()
{
   int    i;
   Ipp16s pSrc[N], pDst[N];

   /* here, the scaleFactor is 15 */
   Ipp16s pTapsIIR[(tapsLen+1)*2] = {
```

continued

Example 4-2 ippsIIR_Direct_16s Usage (continued)

```
    7922, 16348, 22394, 16348,  7922,    15,
    6338, 29356,  1841,  4222
   };
   Ipp32s pDelayLineIIR[tapsLen];

   for ( i = 0; i < tapsLen; i ++ ) {
    pDelayLineIIR[i] = 0;
   }

   printf("\nTesting <ippsIIR_Direct_16s>: \n");
   for ( i = 0; i < N; i ++ ) {
    pSrc[i] = i;
   }

   ippsIIR_Direct_16s(pSrc, pDst, N, pTapsIIR,
   tapsLen, pDelayLineIIR);
   for ( i = 0; i < N; i ++ ) {
    printf("%8d", pDst[i]);
    if ( i%5 == 0 && i != 0 ) {
        printf("\n");
    }
   }
   return(0);
}
#undef N
#undef tapsLen
```

Biquad IIR

The example below illustrates the usage of the `ippsIIROne_BiQuadDirect_16s_I` primitive.

Example 4-3 ippsIIROne_BiquadDirect_16s_I Usage

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"
#define numBiQuad 4
#define N 40

int main()
{
     int    i;

    Ipp16s pValResult;
    Ipp16s pTapsIIR[numBiQuad*6] = {
        3178,  4488,  3178, 14,  -922,  1766,
        7569,  2155,  7569, 14,  -225,  7572,
        9458,   513,  9458, 14,    11,  9542,
        9895,   159,  9895, 14,    55,  9934
    };

    Ipp32s pDelayLineIIR[numBiQuad*2];

    for ( i = 0; i < numBiQuad*2; i ++ ) {
        pDelayLineIIR[i] = 0;
    }
    printf("\nTesting <ippsIIROne_BiQuadDirect_16s_I>:\n");
    for ( i = 0; i < N; i ++ ) {
        pValResult = i;
        ippsIIROne_BiQuadDirect_16s_I(&pValResult, pTapsIIR,
             numBiQuad, pDelayLineIIR);
        printf("%8d", pValResult);
```

continued

Example 4-3 ippsIIROne_BiquadDirect_16s_I Usage (continued)

```
            if ( i != 0 && i%5 == 0 ) {
               printf("\n");
            }
        }

        return(0);
}
#undef numBiQuad
#undef N
```

LMS FIR

The example below illustrates the usage of the `ippsFIRLMSOne_Direct_16s` primitive.

Example 4-4 ippsFIRLMSOne_Direct_16s Usage

```
#include <stdio.h>
#include "ippdefs.h"
#include "ippSP.h"
#define N 80
#define tapsLen 16

int main()
{
    int      i;
    Ipp16sval, pResult, valDesire;
    int    pDelayLineIndex;
    Ipp32sstepQ15;
    Ipp32spTapsLMS[tapsLen];
    Ipp16spDelayLine[tapsLen*2];
    stepQ15 = 1024;  /* 0.03125 in Q16.15 */
    valDesire = 100;

    for ( i = 0; i < tapsLen; i ++ ) {
```

continued

Example 4-4 ippsFIRLMSOne_Direct_16s Usage (continued)

```
        pTapsLMS[i] = 0;
    }

    /* Initialize contents of Delay Line to ZERO */
    /* the first time filter is called */
    pDelayLineIndex = 0;
    for ( i = 0; i < tapsLen*2; i ++ ) {
        pDelayLine[i] = 0;
    }
    printf("\nTesting <ippsFIRLMSOne_Direct_16s>: \n");
    printf("Desired value is %d\n", valDesire);
    printf("The calculated value is \n");
    for ( i = 0; i < N; i ++ ) {
        val = 100 + i%12;
    ippsFIRLMSOne_Direct_16s(val, valDesire, &pResult,
        pTapsLMS, tapsLen, stepQ15, pDelayLine, &pDelayLineIndex);


        printf("%8d", pResult);
        if ( i != 0 && i%5 == 4 ) {
        printf("\n");
       }
    }

    return (0);
}
/* EOF */
```

Windowing 5

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) that are available for data windowing. The set of primitives implements the windows that are most popular for general and multimedia signal processing applications. Available non-parametric windows include the Bartlett, Hann, Hamming, standard Blackman, and optimal Blackman. Available parametric windows include Blackman, which provide users with significant flexibility.

In the context of signal processing, windowing denotes the operation in which a predefined or parametric sequence with a particular set of desirable characteristics is applied via pointwise vector multiplication to a signal vector. This windowing technique typically improves the characteristics or performance of some subsequent analysis. In most cases, the idea is to exploit the duality between time-domain multiplication and frequency-domain convolution. In spectral analysis applications, for example, tapered windows are most often used to reduce the spectral leakage associated with the relatively large magnitude sidelobes of non-tapered windows. On the other hand, non-tapered windows might be used in some cases to improve frequency selectivity at the expense of increased side lobe energy.

In the interest of simplicity and consistency, the mathematical expressions given in this chapter that describe the behavior of each window represent the particular case of the non-in-place and non-scaled function variable (the so-called “default” version). The behavior of any scaled and/or in-place variable can be understood easily by applying to the default behavioral specification the generic in-place and scaled function behavioral rules that are given in Chapter 1.

The rest of this chapter is organized as follows. Non-parametric windows are described in the “Non-Parametric Windows” section, and parametric windows are described in the “Parametric Windows” section. The “Window Usage Examples” section provides example 'C' language source listings that demonstrate the usage of the Intel® IPP windowing primitives.

Non-Parametric Windows

This section describes the non-parametric windowing primitives, including the Bartlett, Hann, Hamming, standard Blackman, and optimal Blackman windows. In general, for an N-point non-parametric window sequence, $w(n)$, and an N-point data vector, $v(n)$, the windowing operation comprises a pointwise multiply of the window against the input, i.e.,

$$v_w(n) = w(n) \cdot v(n), \quad 0 \le n < N$$

where the N-point vector $v_w(n)$ is the windowed output sequence. In the particular case of the Intel® IPP windowing primitives, the user must also be aware of a few other behavioral characteristics. First, the windowing primitives are computed in-place, such that the primitive replaces the non-windowed elements of the input vector with the windowed output elements. Second, the detailed descriptions on each window type give certain restrictions on maximum window lengths that must be observed. Finally, internal scaling is data-dependent. The primitives scale the samples of the window sequence such that the magnitude of the peak window element is set equal to the magnitude of the largest input element. Therefore, in order to maximize precision, it is recommended that users apply block normalization to the input vector prior to application of the window, i.e., the input vector should be scaled such that its largest element has a Q15 magnitude on the interval $[0.5, 1)$. For example, if the N-point input vector $v(i)$ contains elements of the type `Ipp16s`, then the input data should be scaled such that $0x4000 \le max\{|v(i)|\} \le 0x7fff, \ 0 \le i < N$. Complete details on the non-parametric windowing primitives are given next. In addition, example 'C' language source listings that demonstrate their usage are given in the "Window Usage Examples" section at the end of this chapter.

WinBartlett_16s_I

Prototype

```
IppStatus ippsWinBartlett_16s_I(Ipp16s * pSrcDst, int len);
```

Description

Applies a Bartlett (triangular) window, $w_\Delta(n)$, defined as:

to a `len`-element input vector. Uses automatic internal scaling as described in the "Non-Parametric Windows" section.

$$w_{\Delta}(n) = \begin{cases} \frac{2n}{len-1}, & 0 \le n \le \frac{len-1}{2} \\ \left(2 - \frac{2n}{len-1}\right), & \frac{len-1}{2} < n \le len-1 \end{cases}$$

Input Arguments

- `pSrcDst` – pointer the vector that contains the non-windowed input data; block normalization is recommended prior to application of the window (see discussion, the "Non-Parametric Windows" section)
- `len` – number of elements contained in the input/output vector, valid range: $2 < len < 32768$
- `scaleFactor` – the value that the output should be scaled, must be greater than –16 and less than -16

Output Arguments

`pSrcDst` – pointer to the vector that contains the windowed output data

Returns

`ippStsNoErr` – no error

`ippStsBadArgErr` – bad arguments

WinHann_16s_I

Prototype

```
IppStatus ippsWinHann_16s_I(Ipp16s * pSrcDst, int len);
```

Description

Applies a Hann window, $w_{Hann}(n)$, defined as

$$w_{Hann}(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{len-1}\right) \qquad 0 \le n \le len-1$$

to a `len`-element input vector. Uses automatic internal scaling as described in the "Non-Parametric Windows" section.

Input Arguments

- `pSrcDst` – pointer the vector that contains the non-windowed input data; block normalization is recommended prior to application of the window (see discussion, "Non-Parametric Windows" section)
- `len` – number of elements contained in the input/output vector, valid range: $2 < len < 32768$
- scaleFactor – the value that the output should be scaled, must be greater than –16 and less than –16

Output Arguments

- `pSrcDst` – pointer to the vector that contains the windowed output data

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

WinHamming_16s_I

Prototype

```
IppStatus ippsWinHamming_16s_I(Ipp16s * pSrcDst, int len);
```

Description

Applies a Hamming window, $w_{Hamming}(n)$, defined as

$$w_{Hamming}(n) = 0.54 - 0.46 \times \cos\left(\frac{2\pi n}{len-1}\right) \qquad 0 \le n \le len-1$$

to a `len`-element input vector. Uses automatic internal scaling as described in the "Non-Parametric Windows" section.

Input Arguments

- `pSrcDst` – pointer the vector that contains the non-windowed input data; block normalization is recommended prior to application of the window (see discussion, "Non-Parametric Windows" section)
- `len` – number of elements contained in the input/output vector, valid range: $2 < len < 32768$

Output Arguments

`pSrcDst` – pointer to the vector that contains the windowed output data

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

WinBlackmanStd_16s_I

Prototype

```
IppStatus ippsWinBlackmanStd_16s_I(Ipp16s * pSrcDst, int len);
```

Description

Applies a standard Blackman window, $w_{Blk}(n)$, defined as

$$w_{Blk}(n) = 0.42 - 0.5 \cdot \cos\left(\frac{2\pi n}{len-1}\right) + 0.08 \cdot \cos\left(\frac{4\pi n}{len-1}\right) \qquad 0 \le n \le len-1$$

to a `len`-element input vector. Uses automatic internal scaling as described in the "Non-Parametric Windows" section.

Input Arguments

- `pSrcDst` – pointer the vector that contains the non-windowed input data; block normalization is recommended prior to application of the window (see discussion, "Non-Parametric Windows" section)
- `len` – number of elements contained in the input/output vector, valid range: $2 < len < 32768$

Output Arguments

`pSrcDst` – pointer to the vector that contains the windowed output data

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

WinBlackmanOpt_16s_I

Prototype

```
IppStatus ippsWinBlackmanOpt_16s_I(Ipp16s * pSrcDst, int len);
```

Description

Applies an optimal Blackman window, $w_{opt}(n)$, defined as

$$w_{opt}(n) = \frac{alpha+1}{2} - 0.5 \times \cos\left(\frac{2\pi n}{len-1}\right) - \frac{alpha}{2}\cos\left(\frac{4\pi n}{len-1}\right), \quad 0 \le n \le len-1$$

to a `len`-element input vector, where the optimal value for the parameter `alpha`, given by

$$alpha = -\left(\frac{\sin\frac{2\pi}{len-1}}{\sin\frac{2\pi}{len-1}}\right)$$

is an internal constant. The primitive uses automatic internal scaling as described in the "Non-Parametric Windows" section.

Input Arguments

- `pSrcDst` – pointer the vector that contains the non-windowed input data; block normalization is recommended prior to application of the window (see discussion, "Non-Parametric Windows" section)
- `len` – number of elements contained in the input/output vector, valid range: $2 < len < 32768$

Output Arguments

`pSrcDst` – pointer to the vector that contains the windowed output data

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Parametric Windows

This section describes the parametric windowing primitives, including Blackman. In general, for an N-point non-parametric window sequence, $w(\alpha, n)$, and an N-point data vector, $v(n)$, the windowing operation comprises a pointwise multiply of the window against the input, i.e.,

$$v_w(\alpha, n) = w(\alpha, n) \cdot v(n), \quad 0 \le n < N$$

where the N-point vector $v_w(\alpha, n)$ is the windowed output sequence. In the particular case of the Intel® IPP windowing primitives, the user must also be aware of a few other behavioral characteristics. First, the computation is performed in-place, such that the primitive replaces the non-windowed elements of the input vector with the windowed output elements. Second, the detailed descriptions on each window type gives certain restrictions on the window parameters as well as the maximum window lengths that must be observed. Finally, internal scaling is data-dependent. The primitives scale the samples of the window sequence such that the magnitude of the peak window element is set equal to the magnitude of the largest input element. Therefore, in order to maximize precision, it is recommended that users apply block normalization to the input vector prior to application of the window, i.e., the input vector should be scaled such that its largest element has a Q15 magnitude on the interval $[0.5, 1)$. For example, if the N-point input vector $v(i)$ contains elements of the type `Ipp16s`, then the input data should be scaled such that $0x4000 \le max\{|v(i)|\} \le 0x7fff, \ 0 \le i < N$. Complete details on the parametric windowing primitives are given next. In addition, example 'C' language source listings that demonstrate their usage are given in the "Window Usage Examples" section at the end of this chapter.

WinBlackmanQ15_16s_I

Prototype

```
IppStatus ippsWinBlackmanQ15_16s_I(Ipp16s * pSrcDst, int len, Ipp32s
   alphaQ15);
IppStatus ippsWinBlackmanQ15_16s_ISfs(Ipp16s * pSrcDst, int len, Ipp32s
   alphaQ15, int scaleFactor);
```

Description

Applies a parametric Blackman window, $w_B(alpha, n)$, defined as

$$w_B(alpha,n) = \frac{alpha+1}{2} - 0.5 \times \cos\left(\frac{2\pi n}{len-1}\right) - \frac{alpha}{2}\cos\left(\frac{4\pi n}{len-1}\right), \quad 0 \le n \le len-1$$

to a `len`-element input vector, where the user is able to specify a value for the parameter $alpha$ via the Q15 parameter `alphaQ15`, i.e.,

$$alpha = alphaQ15 \cdot 2^{-15}$$

In other words, the parameter `alphaQ15` provides a mechanism for representing the floating-point parameter $alpha$ using an integer data type. Note that the unscaled and in-place primitive variable uses the automatic internal scaling technique described in section "Non-Parametric Windows".

Input Arguments

- `pSrcDst` – pointer the vector that contains the non-windowed input data; block normalization is recommended prior to application of the window (see discussion, "Non-Parametric Windows" section)
- `len` – number of elements contained in the input/output vector, valid range: $2 < len < 32768$

- `alphaQ15` – window parameter *`alpha`* in Q16.15 format. For Q16.15 (Q15 in conjunction with the underlying type `Ipp32s`), this corresponds to a range for the parameter *`alpha`* of $-65536 \le alpha < 65536$, with a precision of 2^{-15}.
- `scaleFactor` – saturation fixed scalefactor (only for the scaled primitive)

Output Arguments

`pSrcDst` – pointer to the vector that contains the windowed output data

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

Window Usage Examples

This section provides 'C' language source listings that illustrate the usage of the windowing primitives.

Hamming Window

The example below illustrates the usage of the `ippsWinHamming_16s_I` primitive. In the example, a Hamming window is applied to a signal vector containing 256 elements.

Example 5-1 ippsWinHamming_16s_I Usage

```
#include <stdio.h>
#include <malloc.h>
#include "ippdefs.h"
#include "ippSP.h"

int main()
{
    Ipp16s  * x;
    int     i, len = 256;

    /* allocate memory for signal vector */
    x = (Ipp16s *)malloc(len * sizeof(Ipp16s));
   for ( i = 0; i < len; i ++ ) {
       x[i] = i - (len>>1);
   }

   /* call windowing primitive */
```

continued

Example 5-1 ippsWinHamming_16s_I Usage (continued)

```
    ippsWinHamming_16s_I(x, len);

    /* output signal */
     for ( i = 0; i < len; i ++ ) {
          printf("%10d", x[i]);
     if ( (i+1)%6 == 0 ) {
               printf("\n");
     }
          else {
               printf(",");
     }
     }

     free(x);

     return( 0 );
}
```

Convolution 6

This chapter describes discrete convolution functions for the Intel® Integrated Performance Primitives (Intel® IPP) on the Intel® Integrated Performance Primitives on Intel® PCA Processors with Intel® Wireless MMX™ Technology (PCA processors with MMX™). Convolution is the linear operator that defines the output of a Linear Time-Invariant (LTI) system in response to an input. In particular, the output of a discrete-time LTI system in response to an arbitrary input sequence is given by the convolution of the input sequence with the system impulse response sequence. Intel® IPP offers primitives that perform discrete-time convolution for one-dimensional (1D) sequences.

In the interest of simplicity and consistency, the mathematical expressions given in this chapter to describe the behavior of each convolution primitive represent the particular case of the non-in-place and non-scaled function variable (the so-called "default" function version). The user should be aware that the behavior of any scaled and/or in-place variables can be understood easily by applying to the default behavioral specification the generic in-place and scaled function behavioral rules that are given in Chapter 1.

The remainder of this chapter is organized as follows. The first section, "One-Dimensional (1D) Convolution" describes the 1D convolution primitives. The last section, "Convolution Usage Examples" provides example 'C' language source listings that demonstrate the usage of the Intel® IPP convolution primitives.

One-Dimensional (1D) Convolution

A single primitive is provided for 1D convolution, namely `ippsConv_16s`. Details for each of these primitives are given next. The "default" function behavior and arguments are described for each primitive. Scaled and in-place primitive variables can be understood easily by applying the behavioral rules given in Chapter 1.

Conv_16s

Prototype

```
IppStatus ippsConv_16s(const Ipp16s * pSrc1, int src1Len, const Ipp16s *
   pSrc2, int src2Len, Ipp16s * pDst);
IppStatus ippsConv_16s_Sfs(const Ipp16s * pSrc1, int src1Len, const
   Ipp16s * pSrc2, int src2Len, Ipp16s * pDst, int scaleFactor);
```

Description

Performs a discrete convolution of two vectors, `*pSrc1` and `*pSrc2`, that have, respectively, `src1Len` and `src2Len` elements, i.e.,

$$pDst[i] = \sum_{k=k_{min}}^{k_{max}} pSrc2[k] \cdot pSrc1[i-k], \quad 0 \le i < src1Len + src2Len - 1$$

where the lower and upper summation limits, respectively, are obtained from the vector lengths as follows:

$$k_{min} = max(0, i - src1Len + 1)$$

and

$$k_{max} = min(i, src2Len - 1)$$

NOTE. *If the input vectors contain appropriately arranged coefficients of two polynomials, then the convolution is equivalent to polynomial multiplication.*

Input Arguments

- `pSrc1`, `pSrc2` – pointers to the input vectors
- `src1Len` and `src2Len` – the number of samples contained in each of the input vectors

Output Arguments

`pDst` – pointer to the convolution output vector

Convolution Usage Examples

This section provides 'C' language source listings that illustrate the usage of the 1D convolution primitive.

One-Dimensional Convolution

The source code example below illustrates the usage of `ippsConv_16s` primitive.

Example 6-1 ippsConv_16s Usage

```
#include <stdio.h>
#include ¡°ippdefs.h¡±
#include ¡°ippdsp.h¡±
#define src1Len  10
#define src2Len  11

int main()
{
    int    i;

    /* variables in 1D convolution */
    int    dstLen;
    Ipp16s pSrc1[src1Len];
    Ipp16s pSrc2[src2Len];
    Ipp16s pDst[src1Len+src2Len-1];
```

continued

Example 6-1 ippsConv_16s Usage (continued)

```
        /* initialize source array */
    for( i = 0; i < src1Len; i ++ ) {
        pSrc1[i] = i;
    }
    for( i = 0; i < src2Len; i ++ ) {
        pSrc2[i] = 1;
}

    /* call 1D convolution */
    ippsConv_16s(pSrc1, src1Len, pSrc2, src2Len, pDst);

    /* display result */
    printf(¡°Test Result \n¡±);
    for ( i = 0; i < (src1Len+src2Len-1); i ++ ) {
    printf("%5d", pDst[i]);
    if ( i!=0 && i%5==0 ) {
        printf("\n");
    }
    }

    return(0);
}
#undef src1Len
#undef src2Len
```

Transforms 7

This chapter describes the Intel Integrated Performance Primitives (Intel® IPP) that are available for computing block transforms. The set of transform primitives includes both forward and inverse Discrete Fourier Transforms (DFTs) of several radix-2 block lengths. These primitives have been implemented using Fast Fourier Transform (FFT) algorithms.

In the interest of simplicity and consistency, the mathematical expressions given in this chapter describe the behavior of each transform.

This chapter is organized as follows. FFT APIs are described in the "Variable-length FFT APIs" section, a list of callback functions that must be implemented are described in "Callback Function APIs", and "Usage Example of FFT Functions" which provides example 'C' language source listings that demonstrate the usage of the Intel® IPP FFT primitives.

Variable-length FFT APIs

The variable-length FFT APIs include an initialization function, a forward FFT function, an inverse FFT function, and a helper function to calculate the necessary buffer size. The maximum FFT length is 4096.

The structure `IppsFFTSpec_C_16s` is used to provide help information for the variable-length FFT. Its definition is:

```
typedef void IppsFFTSpec_C_16s;
```

User can simply call initialization function to initialize it.

FFTGetBufSize_C_16sc

Prototype

```
IppStatus ippsFFTGetBufSize_C_16sc(const IppsFFTSpec_C_16sc* pFFTSpec, int*
   pSize);
```

Description

This function computes the size of the buffer used in the `ippsFFTFwd_CToC_16sc_Sfs` and `ippsFFTInv_CToC_16sc_Sfs` functions.

Input Arguments

`pFFTSpec` – pointer to the initialized tables of FFT coefficients

Output Arguments

`pSize` – pointer that hold the size of the buffer which the user allocates before calling the `ippsFFTFwd_CToC_16sc_Sfs` and `ippsFFTInv_CToC_16sc_Sfs` functions. The sizes are in bytes.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned under the following conditions:
  - a pointer was NULL

FFTFree_C_16sc

Prototype

```
IppStatus ippsFFTFree_C_16sc (const IppsFFTSpec_C_16sc *pFFTSpec);
```

Description

This function frees the buffer used in the `ippsFFTFwd_CToC_16sc_Sfs` and `ippsFFTInv_CToC_16sc_Sfs` functions.

Input Arguments

`pFFTSpec` – pointer to the initialized FFT structure

Output Arguments

None.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned under the following conditions:
  - a pointer was NULL

FFTInitAlloc_C_16s

Prototype

```
IppStatus ippsFFTInitAlloc_C_16s (IppsFFTSpec_C_16sc **pFFTSpec, int
   order, int flag, IppHintAlgorithm hint);
```

Description

This function initiates the specification for 2^`order` complex FFT, whose maximum length is 4096 and minimum length is 1.

This function calls `ippsMalloc` which the user is responsible for implementing.

Input Arguments

- `order` – the base-2 logarithm of the FFT length; 0<order<=12
- `flag` – reserved. Only zero is supported.
- `hint` – reserved. Only zero is supported.

Output Arguments

`pFFTSpec` – pointer to the initialized FFT specification structure

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - `pFFTSpec` was NULL
  - `order < 0` or `order > 12`

NOTE. *The memory of each table in the specification structure should be allocated beforehand. Please refer to the structure definition of IppsFFTSpec_C_16s for their length.*

FFTFwd_CtoC_16sc_Sfs

Prototype

```
IppStatus ippsFFTFwd_CtoC_16sc_Sfs (const Ipp16sc *pSrc, Ipp16sc *pDst, const
    IppsFFTSpec_C_16sc *pFFTSpec, int scaleFactor,Ipp8u *pBuffer);
```

Description

This function computes a forward FFT of a complex signal in the length of 2^`order`, with a maximum length of 4096 and a minimum length of 1.

The DFT formula used in this function for complex forward FFT is:

$$X[k] = 2^{-scaleFactor} \sum_{n=0}^{2^{order}-1} x[n] e^{-j\frac{2\pi}{2^{order}}nk}, k = 0,1,..., 2^{order} - 1$$

Before calling the N-point forward FFT ( ), `ippsFFTInitAlloc_C_16sc` must be called to initialize the FFT specification used for this function.

`pBuffer` must be allocated before calling this function and the buffer size should be obtained by calling `ippsFFTGetBufSize_C_16sc`.

After calling forward FFT functions, `ippsFFTFree_C_16sc` should be called to free all the memory allocated by `ippsFFTInitAlloc_C_16sc`.

`ippsFFTInitAlloc_C_16sc` and `ippsFFTFree_C_16sc` call two system-dependent callback functions, `ippsMalloc` and `ippsFree`. The user is responsible for implementing these two functions.

Input Arguments

- `pSrc` – pointer to the complex array that holds the input signal. The length of this array is 2^`order` in `Ipp16sc` type. Valid data access range is [0,2^order-1]. Must be aligned at 8-byte boundary.
- `pFFTSpec` – pointer to the FFT specification structure
- `scaleFactor` – output scale factor value. Valid data value range is [0,16].
- `pBuffer` – pointer to the work buffer. The buffer must be allocated before the function is called.

Output Arguments

`pDst` –pointer to the complex array that holds the output signal. The length of this array is 2^`order` in `Ipp16sc` type. Valid data access range is [0,2^order-1]. Must be aligned at 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - `order < 0` or `order > 12`
  - `scaleFactor < 0` or `scaleFactor > 16`
  - `pSrc` or `pDst` is not alligned at 8-byte boundary

NOTE. *Before this function is executed, pFFTSpec should be built by calling ippsFFTInitAlloc_C_16sc.*

FFTInv_CToC_16sc_Sfs

Prototype

```
IppStatus ippsFFTInv_CToC_16sc_Sfs (const Ipp16sc *pSrc, Ipp16sc *pDst, const
    IppsFFTSpec_C_16sc *pFFTSpec, int scaleFactor, Ipp8u *pBuffer);
```

Description

This function computes an inverse FFT of a complex signal in the length of 2^order, whose maximum length is 4096.

The DFT formula used in this function for complex inverse FFT is:

$$x[n] = \frac{2^{-scaleFactor}}{2^{order}} \sum_{k=0}^{2^{order}-1} X[k] e^{j\frac{2\pi}{2^{order}}nk}, n = 0,1,...,2^{order} - 1.(N = 2^{order})$$

Before calling N-point inverse FFT ( ), `ippsFFTInitAlloc_C_16sc` must be called to initialize the FFT specification used for this function.

`pBuffer` should also be allocated before calling this function; the buffer size can be retrieved by calling `ippsFFTGetBufSize_C_16sc`.

After calling forward FFT functions, `ippsFFTFree_C_16sc` should be called to free all the memory allocated by `ippsFFTInitAlloc_C_16sc`.

`ippsFFTInitAlloc_C_16sc` and `ippsFFTFree_C_16sc` will call two system-dependent callback functions, `ippsMalloc` and `ippsFree`. The user is responsible for implementing these two functions.

Input Arguments

- `pSrc` – pointer to the real array that holds the input signal. The length of this array is 2^`order` in `Ipp16sc` type. Valid data access range is [0,2^`order` -1]. Align at 8-byte boundary.
- `pFFTSpec` – pointer to the FFT specification structure
- `scaleFactor` – output scale factor value. Valid data vlaue range is [0,16].
- `pBuffer` – pointer to the work buffer which must be initialized before the function is called. Call `ippsFFTGetBufSize_C_16sc` to obtain the size of `pBuffer`.

Output Arguments

- `pDst` – pointer to the complex array that holds the output signal. The length of this array is 2^`order` in `Ipp16sc` type. Valid data access range is [0,2^`order` -1]. Align at 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - `pSrc` or `pDst` are not 8-byte aligned
  - `order < 0` or `order > 12`
  - `scaleFactor < 0` or `scaleFactor > 16`

NOTE. *Before this function is executed, pFFTSpec should be built by calling ippsFFTInitAlloc_C_16s.*

^N Vary-Length Complex to Complex Forward FFT APIs

FFTFwd_CToC_16sc_Sfs
FFTFwd_CToC_32sc_Sfs

Prototype

```
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);
```

Description

These functions compute a forward FFT of a complex signal in the length of 2^{order}, whose maximum length is 4096, and minimum length is 1. The input/output formats and twiddle factors are different for the two APIs. The twiddle factor used in <ippsFFTFwd_CToC_16sc_Sfs> is the Q14 format, while the twiddle factor used in <ippsFFTFwd_CToC_32sc_Sfs> is the Q30 format.

The DFT formula used in this function for complex forward FFT is:

$$X[k] = 2^{-scaleFactor} \sum_{n=0}^{2^{order}-1} x[n] e^{-j\frac{2\pi}{2^{order}}nk}, k = 0,1,...,2^{order} - 1 (N = 2^{order})$$

Before calling N-length forward FFT (N=2^{order}), you should call <ippsFFTInitAlloc_C_16sc> for <ippsFFTFwd_CToC_16sc_sfs> and <ippsFFTInitAlloc_C_32sc> for <ippsFFTFwd_CToC_32sc_Sfs> to prepare and initialize the FFT specification used for this function. Please refer to "Specification Init of 2^N Vary-length Complex to Complex FFT APIs" for detailed information.

Also, allocate pBuffer before calling this function. Obtain the buffer size with <ippsFFTGetBufSize_C_16sc> or <ippsFFTGetBufSize_C_32sc>. Please refer to "Buffer Size of 2^N Vary-length Complex to Complex FFT APIs" for detailed information.

After calling the forward FFT functions, <ippsFFTFree_C_16sc> or <ippsFFTFree_C_32sc> should be called to free all of the memory allocated by <ippsFFTInitAlloc_C_16sc> or <ippsFFTInitAlloc_C_32sc>. Please refer to "Specification Init of 2^N Vary-length Complex to Complex FFT APIs" for detailed information.

<ippsFFTInitAlloc_C_16sc>, <ippsFFTInitAlloc_C_32sc>, <ippsFFTFree_C_16sc>, and <ippsFFTFree_C_32sc> call the two system-dependent callback functions <ippsMalloc> and <ippsFree>. The user is responsible for implementing these two functions. Please refer to section "Callback Function APIs" for detailed information.

Please refer to "Example" for sample code that uses this function.

Input Arguments

- pSrc – pointer to the complex array that holds the input signals. The number of elements in this array is 2^{order}, is Ipp16sc type for <ippsFFTFwd_CToC_16sc_Sfs>, and Ipp32sc type for <ippsFFTFwd_CToC_32sc_Sfs>. Valid data access range is [0,2^{order}–1].
- pFFTSpec – pointer to the FFT specification structure.
- scaleFactor – scale factor of the output.
  Valid data value range for <ippsFFTFwd_CToC_16sc_Sfs> is [0,16], and for <ippsFFTFwd_CToC_32sc_Sfs> is [0,32].

- pBuffer – pointer to the work buffer; must be allocated before this function is called.

Output Arguments

- pDst – pointer to the complex array that holds the output signals. The number of elements in this array is 2^{order}, is Ipp16sc type for <ippsFFTFwd_CToC_16sc_Sfs>, and Ipp32sc type for <ippsFFTFwd_CToC_32sc_Sfs>. Valid data access range is $[0,2^{order}-1]$.

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- If pSrc, pDst, pFFTSpec, or pBuffer is NULL for <ippsFFTFwd_CToC_16sc_Sfs>
- If pSrc, pDst, or pFFTSpec is NULL for <ippsFFTFwd_CToC_32sc_Sfs>
- pSrc or pDst is not 8-byte boundary aligned
- aligned is not 8-byte boundary aligned for <ippsFFTFwd_CToC_16sc_Sfs>
- order <0 or order > 12
- scaleFactor<0 or scaleFactor >16 for <ippsFFTFwd_CToC_16sc_Sfs>
- scaleFactor<0 or scaleFactor >32 for <ippsFFTFwd_CToC_32sc_Sfs>

NOTE. *Alignment requirement:*
pSrc, pDst should be aligned at 8-byte boundary.
pBuffer should be aligned at 8-byte boundary for <ippsFFTFwd_CToC_16sc_Sfs>.

Accuracy Criteria

Assume Out is the output of one of the above IPP FFT APIs and Ref is the output of the corresponding double-precision floating point FFT implementation without any optimized algorithm. The accuracy criterion for <ippsFFTFwd_CToC_16sc_Sfs> is defined as below:

$$ME = \frac{1}{N}\sum_{i=0}^{N-1}\left|Out_i - ref_i\right| \le 2, (N = 2^{order}, 0 \le order \le 12)$$

The accuracy criterion for <ippsFFTFwd_CToC_32sc_Sfs> defined as below:

Define:

$$SNR = 10\log_{10}\frac{\sum_{i=0}^{N-1}\left|ref_i\right|^2}{\sum_{i=0}^{N-1}\left|Out_i - \gamma ef_i\right|^2},\quad d = \underset{i=0}{\overset{N-1}{MAX}}\left|ref_i - out_i\right|$$

The varied length forward FFT <ippsFFTFwd_CToC_32sc_Sfs> satisfies:

SNR >= 96 or d <1 (Note that he latter criterion is set especially for small input signals)

Example

Several steps should be followed to use the <ippsFFTFwd_CToC_16sc_Sfs>.

1. Call <ippsFFTInitAlloc_C_16sc> to prepare and initialize the FFT specification.
2. Call <ippsFFTGetBufSize_C_16sc> to get the buffer size.
3. Allocate pBuffer according to the size retrieved in step 2.
4. Call forward FFT function <ippsFFTFwd_CtoC_16sc_Sfs>.
5. After calling FFT function, call <ippsFFTFree_C_16sc> to free all the memory allocated by <ippsFFTInitAlloc_C_16sc>.
6. <ippsFFTInitAlloc_C_16sc> and <ippsFFTFree_C_16sc> call two system-dependent callback functions <ippsMalloc> and <ippsFree>. The user is responsible for implementing these two functions

The following is an example of forward FFT.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "ippSP.h"


IppStatus fft();

void* _CALLBACK ippsMalloc(int size)
{
    return malloc(size);
}

void _CALLBACK ippsFree(void* pSrcBuf)
{
```

```
    free(pSrcBuf);
}

int main()
{
    fft();
}

IppStatus fft()
{
    Ipp16sc x[8], y[8];
    int n, bufSize;
    IppStatus status;
    IppsFFTSpec_C_16sc* pSpec = NULL;
    Ipp8u *pWorkBuf = NULL;

    srand( (unsigned)time( NULL ) );

    for(n=0; n<8; n++)  {
        x[n].re = (Ipp16s)((rand()%1024)-512);
        x[n].im = (Ipp16s)((rand()%1024)-512);
    }

    status = ippsFFTInitAlloc_C_16sc(&pSpec, 3, IPP_FFT_NODIV_BY_ANY,
   ippAlgHintNone );
    status = ippsFFTGetBufSize_C_16sc(pSpec, &bufSize);
    pWorkBuf= (Ipp8u*)malloc(bufSize);
    status = ippsFFTFwd_CToC_16sc_Sfs ( x, y, pSpec, 0, pWorkBuf );

    for(n=0; n<8; n++)  {
        printf("(%d, ", y[n].re);
        printf("%d)\n", y[n].im);
    }
    if(pSpec) {
        ippsFFTFree_C_16sc ( pSpec );
    }
    free(pWorkBuf);
    return status;
```

```
}
```

^N Vary-Length Complex to Complex Inverse FFT APIs

FFTInv_CToC_16sc_Sfs

Prototype

```
IppStatus ippsFFTInv_CToC_16sc_Sfs (const Ipp16sc *pSrc, Ipp16sc *pDst,
   const IppsFFTSpec_C_16sc *pFFTSpec, int scaleFactor, Ipp8u *pBuffer);
```

Description

This function computes an inverse FFT of a complex signal in the length of 2^{order}, whose maximum length is 4096.

The DFT formula used in this function is:

$$x[n] = \frac{2^{-scaleFactor}}{2^{order}} \sum_{k=0}^{2^{order}-1} X[k] e^{j\frac{2\pi}{2^{order}}nk}, n = 0,1,...,2^{order} - 1. (N = 2^{order})$$

Before calling N-length inverse FFT ($N=2^{order}$), the user should call `<ippsFFTInitAlloc_C_16sc>` to prepare and initialize the FFT specification used for this function. Please refer to "Specification Init of 2^N Vary-length Complex to Complex FFT APIs" for its detailed information.

`pBuffer` must be allocated before calling this function. The buffer size can be retrieved by issuing `<ippsFFTGetBufSize_C_16sc>`. Please refer to"Buffer Size of 2^N Vary-length Complex to Complex FFT APIs" for detailed information.

After calling forward FFT functions, call `<ippsFFTFree_C_16sc>` to free the memory allocated by `<ippsFFTInitAlloc_C_16sc>`. Please refer to "Specification Init of 2^N Vary-length Complex to Complex FFT APIs" for detailed information.

`<ippsFFTInitAlloc_C_16sc>` and `<ippsFFTFree_C_16sc>` call two system-dependent callback functions `<ippsMalloc>` and `<ippsFree>`. The user is responsible for implementing these two functions. Please refer to "Callback Function APIs" for detailed information.

Input Arguments

- `pSrc` – pointer to the real array that holds the input signal. The number of elements in this array is 2^{order} and is Ipp16sc type. Valid data access range is [0, 2^{order}–1].
- `pFFTSpec` – pointer to the FFT specification structure.
- `scaleFactor` – scale factor of the output. Valid data value range is [0,16].
- `pBuffer` – pointer to the work buffer. `pBuffer` must be allocated before this function is called. Its size is obtained by calling `<ippsFFTGetBufSize_C_16sc>`.

Output Arguments

- `pDst` – pointer to the complex array that holds the output signal. The number of elements in this array is 2^{order} and Ipp16sc type. Valid data access range is [0, 2^{order}–1].

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- `pSrc`, `pDst`, `pFFTSpec`, or `pBuffer` is NULL
- `pSrc` or `pDst` is not aligned at 8-byte boundary
- `order` <0 or `order` > 12
- `scaleFactor`<0 or `scaleFactor` >16

NOTE. *Alignment requirement:*
pSrc, pDst should be aligned at 8-byte boundary.

Accuracy Criteria

Assume Out is the output of one of the above IPP FFT APIs and Ref is the output of the corresponding double-precision floating point FFT implementation without any optimized algorithm. The accuracy criteria is defined as below:

$$ME = \frac{1}{N}\sum_{i=0}^{N-1}\left|Out_i - ref_i\right| \le 2, (N = 2^{order}, 0 \le order \le 12)$$

^N Vary-length Real to Complex Forward FFT APIs

FFTFwd_RToCCS_16s32s_Sfs

Prototype

```
IppStatus ippsFFTFwd_RToCCS_16s32s_Sfs (const Ipp16s *pSrc, Ipp32s
   *pDst, const IppsFFTSpec_R_16s32s *pFFTSpec, int scaleFactor, Ipp8u
   *pBuffer);
```

Description

This function computes a forward FFT of a real signal in the length of 2^{order}, whose maximum length is 4096 and uses RCCcs format in the output.

The basic DFT formula of real to complex forward FFT in this function is:

$$x[n] = \frac{2^{-scaleFactor}}{2^{order}} \sum_{k=0}^{2^{order}-1} X[k] e^{j\frac{2\pi}{2^{order}}nk}, n = 0,1,...,2^{order}-1.(N = 2^{order})$$

Before calling N-length inverse FFT (N=2^{order}), call <ippsFFTInitAlloc_R_16s32s> to prepare and initialize the FFT specification used for this function. Please refer to "Specification Init of 2^N Vary-length Real<->Complex FFT" for detailed information.

pBuffer must be allocated before calling this function, the buffer size should be obtained by calling <ippsFFTGetBufSize_R_16s32s>. Please refer to "Buffer Size of 2^N Vary-length Real<->Complex FFT APIs" for detailed information.

After calling forward FFT functions, call <ippsFFTFree_R_16s32s> to free all the memory allocated by <ippsFFTInitAlloc_R_16s32s>. Please refer to "Specification Init of 2^N Vary-length Real<->Complex FFT" for detailed information.

<ippsFFTInitAlloc_R_16s32s> and <ippsFFTFree_R_16s32s> call two system-dependent callback functions <ippsMalloc> and <ippsFree>. The user is responsible for implementing these two functions. Please refer to "Callback Function APIs" for detailed information.

Please refer to "Example" for sample code of how to use this function.

Input Arguments

- `pSrc` – pointer to the real array that holds the input signals. The number of elements in this array is $N=2^{order}$ and is Ipp16s type. Valid data access range is $[0,2^{order}-1]$.
- `pFFTSpec` – pointer to the FFT specification structure.
- `scaleFactor`– scale factor of the output. Valid data value range is [0, 32].
- `pBuffer` – pointer to the work buffer. `pBuffer` must be allocated before this function is called. Its size is retrieved by calling <ippsFFTGetBufSize_R_16s32s>.

Output Arguments

- `pDst` – pointer to the real array that holds the output in RCCcs format. The number of elements of this array in Ipp32s type is $2^{order}+2$. Valid data access range is from $[0,2^{order}+1]$.

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- `pSrc`, `pDst`, `pFFTSpec`, or `pBuffer` is NULL
- `pSrc` or `pDst` is not aligned at 8-byte boundary
- `order` <0 or `order` > 12
- `scaleFactor`<0 or `scaleFactor` >32

NOTE. *Alignment requirement:*
pSrc, pDst should be aligned at 8-byte boundary.

Accuracy Criteria

Assume Out is the output of one of the above IPP FFT APIs and Ref is the output of the corresponding double-precision floating point FFT implementation without any optimized algorithm. The accuracy criteria is defined as below:

Define:

$$SNR = 10\log_{10}\frac{\sum_{i=0}^{N-1}|ref_i|^2}{\sum_{i=0}^{N-1}|Out_i - \gamma ef_i|^2}\ ,\quad d = \underset{i=0}{\overset{N-1}{MAX}}|ref_i - out_i|$$

The real to complex forward FFT <ippsFFTFwd_RToCCS_16s32s_Sfs> satisfies:

SNR >= 115 or d <1 (Note that the latter criterion is set especially for small input signals)

Example

Several steps should be followed to use the <ippsFFTFwd_RToCCS_16s32s_Sfs>.

1. Call <ippsFFTInitAlloc_R_16s32s> to prepare and initialize the FFT specification.
2. Call <ippsFFTGetBufSize_R_16s32s> to get the buffer size.
3. Allocate pBuffer according to the size obtained from step 2.
4. Call forward FFT function <ippsFFTFwd_RToCCS_16s32s_Sfs>.
5. After calling FFT function, call <ippsFFTFree_R_16s32s> to free the memory allocated by <ippsFFTInitAlloc_R_16s32s>.
6. <ippsFFTInitAlloc_R_16s32s> and <ippsFFTFree_R_16s32s> call two system-dependent callback functions <ippsMalloc> and <ippsFree>. The user is responsible for implementing these two functions.

The following is an example for forward Real To Complex FFT.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "ippSP.h"


IppStatus fft();

void* _CALLBACK ippsMalloc(int size)
{
    return malloc(size);
}

void _CALLBACK ippsFree(void* pSrcBuf)
{
```

```
    free(pSrcBuf);
}

int main()
{
    fft();
}

IppStatus fft()
{
Ipp16s x[8];
Ipp32s y[10];
    int n, bufSize;
    IppStatus status;
    IppsFFTSpec_R_16s32s* pSpec = NULL;
    Ipp8u *pWorkBuf = NULL;

    srand( (unsigned)time( NULL ) );

    for(n=0; n<8; n++)  {
        x[n] = (Ipp16s)((rand()%1024)-512);
    }
status = ippsFFTInitAlloc_R_16s32s(&pSpec, 3, IPP_FFT_NODIV_BY_ANY,
         ippAlgHintNone );
    status = ippsFFTGetBufSize_R_16s32s(pSpec, &bufSize);
    pWorkBuf= (Ipp8u*)malloc(bufSize);
    status = ippsFFTFwd_RToCCS_16s32s_Sfs ( x, y, pSpec, 0, pWorkBuf );

    for(n=0; n<=4; n++)  {
        printf("(%d, ", y[2*n]);
        printf("%d)\n", y[2*n+1]);
    }
    if(pSpec) {
        ippsFFTFree_R_16s32s ( pSpec );
    }
    free(pWorkBuf);
    return status;
}
```

^N Vary-length Complex to Real Inverse FFT APIs

FFTFwd_RToCCS_16s32s_Sfs
FFTInv_CCSToR_32s_Sfs

Prototype

```
IppStatus ippsFFTInv_CCSToR_32s16s_Sfs (const Ipp32s *pSrc, Ipp16s
   *pDst,const IppsFFTSpec_R_32s *pFFTSpec, int scaleFactor, Ipp8u
   *pBuffer);
IppStatus ippsFFTInv_CCSToR_32s_Sfs (const Ipp32s *pSrc, Ipp32s *pDst,
   const IppsFFTSpec_R_16s32s *pFFTSpec, int scaleFactor, Ipp8u
   *pBuffer);
```

Descriptions

This function set computes the inverse FFT of a complex signal array using RCCcs to format the output.

Before calling each N-length inverse Complex to Real FFT (N=2^{order}), you must call <ippsFFTInitAlloc_R_16s32s> or <ippsFFTInitAlloc_R_32s> to prepare and initialize the FFT specification used for this function. Please refer to "Specification Init of 2^N Vary-length Real<->Complex FFT" for its detailed information.

pBuffer must be allocated before calling each function. The buffer size can be obtained by calling <ippsFFTGetBufSize_R_16s32s> or <ippsFFTGetBufSize_R_32s>. Please refer to "Buffer Size of 2^N Vary-length Real<->Complex FFT APIs" for detailed information.

After calling forward FFT functions, call <ippsFFTFree_R_16s32s> or <ippsFFTFree_R_32s> to free the memory allocated by <ippsFFTInitAlloc_R_16s32s> or <ippsFFTInitAlloc_R_32s>. Please refer to "Specification Free of 2^N Vary-length Complex to Complex FFT" for detailed information.

<ippsFFTInitAlloc_R_16s32s>, <ippsFFTInitAlloc_R_32s>, <ippsFFTFree_R_16s32s>, and <ippsFFTFree_R_32s> call two system-dependent callback functions <ippsMalloc> and <ippsFree>. The user is responsible for implementing these two functions. Please refer to "Callback Function APIs" for detailed information.

Please refer to "Example" for sample code of how to use this function.

Input Arguments

- `pSrc` – pointer to the complex array that holds the input signals. The number of elements in this array is 2^{order} is and Ipp16sc type for `<ippsFFTInv_CCSToR_32s16s _Sfs>` and Ipp32sc type for `<ippsFFTInv_CCSToR_32s_Sfs>`. Valid data access range is $[0,2^{order}+1]$.
- `pFFTSpec` – pointer to the FFT specification structure.
- `scaleFactor` – scale factor of the output. Valid data value range is [0,32].
- `pBuffer` – pointer to the work buffer. `pBuffer` must be allocated before this function is called. Its size is obtained by calling `pBuffer` or `<ippsFFTGetBufSize_R_32s>`.

Output Arguments

- `pDst` – pointer to the complex array that holds the output signals. The number of elements in this array is 2^{order} and is Ipp16s type for `<ippsFFTInv_CCSToR_32s16s_Sfs>` and Ipp32s type for `<ippsFFTInv_CCSToR_32s_Sfs>`. Valid data access range is $[0,2^{order}-1]$.

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- `pSrc`, `pDst`, `pFFTSpec`, or `pBuffer` is NULL.
- `pSrc` or `pDst` or `pBuffer` is not aligned at 8-byte boundary
- `order` <0 or `order` > 12
- `scaleFactor`<0 or `scaleFactor` >16 for `<ippsFFTInv_CCSToR_32s16s_Sfs>`
- `scaleFactor`<0 or `scaleFactor` >32 for `<ippsFFTInv_CCSToR_32s_Sfs>`

NOTE. *Alignment requirement:*
pSrc, pDst should be aligned at 8-byte boundary.

Format Requirement

- `pSrc` should be in RCCcs format.

Accuracy Criteria

Assume Out is the output of one of the above IPP FFT APIs and Ref is the output of the corresponding double-precision floating point FFT implementation without any optimized algorithm. The accuracy criterion is defined as:

Define:

$$SNR = 10\log_{10}\frac{\sum_{i=0}^{N-1}|ref_i|^2}{\sum_{i=0}^{N-1}|Out_i - \gamma ef_i|^2} \quad , \quad d = \underset{i=0}{\overset{N-1}{MAX}}|ref_i - out_i|, (N = 2^{order}, 0 \le order \le 12)$$

The <ippsFFTInv_CCSToR_32s16s_Sfs> satisfies:

SNR >= 83 or d <1 (Note that the latter criterion is set especially for small input signals)

The <ippsFFTInv_CCSToR_32s_Sfs> satisfies:

SNR >= 116 or d <1 (Note that the latter criterion is set especially for small input signals)

Example

Several steps should be followed to use <ippsFFTInv_CCSToR_32s16s_Sfs>.

1. Call <ippsFFTInitAlloc_R_16s32s> to prepare and initialize the FFT specification.
2. Call <ippsFFTGetBufSize_R_16s32s> to get the buffer size.
3. Allocate pBuffer according to the size retrieved from step 2.
4. Call forward FFT function <ippsFFTInv_CCSToR_32s16s_Sfs>.
5. After calling the FFT function, call <ippsFFTFree_R_16s32s> to free the memory allocated by <ippsFFTInitAlloc_R_16s32s>.
6. <ippsFFTInitAlloc_R_16s32s> and <ippsFFTFree_R_16s32s> call two system-dependent callback functions <ippsMalloc> and <ippsFree>. The user is responsible for implementing these two functions.

The following is an example for forward Real To Complex FFT.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "ippSP.h"


IppStatus fft();

void* _CALLBACK ippsMalloc(int size)
{
    return malloc(size);
}
```

```
void _CALLBACK ippsFree(void* pSrcBuf)
{
    free(pSrcBuf);
}

int main()
{
    fft();
}

IppStatus fft()
{
Ipp32s x[10];
Ipp16s y[8];
    int n, bufSize;
    IppStatus status;
    IppsFFTSpec_R_16s32s* pSpec = NULL;
    Ipp8u *pWorkBuf = NULL;

    srand( (unsigned)time( NULL ) );

         x[0] = (Ipp32s) ((rand()%1024)-512);
         x[4*2] = (Ipp32s) ((rand()%1024)-512);
         x[1] = x[2*4+1] = 0;
    for(n=1; n<4; n++)  {
        x[n*2] = (Ipp32s)((rand()%1024)-512);
        x[n*2+1] = (Ipp32s)((rand()%1024)-512);
    }
    status = ippsFFTInitAlloc_R_16s32s(&pSpec, 3, IPP_FFT_NODIV_BY_ANY,
   ippAlgHintNone );
    status = ippsFFTGetBufSize_R_16s32s(pSpec, &bufSize);
    pWorkBuf= (Ipp8u*)malloc(bufSize);
    status = ippsFFTInv_CCSToR_32s16s_Sfs ( x, y, pSpec, 0, pWorkBuf );

    for(n=0; n<4; n++)  {
        printf("(%d, ", y[2*n]);
        printf("%d)\n", y[2*n+1]);
```

```
        }
        if(pSpec) {
            ippsFFTFree_R_16s32s ( pSpec );
        }
        free(pWorkBuf);
        return status;
    }
```

Specification Init of 2^N Vary-length Complex to Complex FFT APIs

FFTInitAlloc_C_16sc
FFTInitAlloc_C_32sc

Prototype

```
IppStatus ippsFFTInitAlloc_C_16sc (IppsFFTSpec_C_16sc **pFFTSpec, int
   order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32sc (IppsFFTSpec_C_32sc **pFFTSpec, int
   order, int flag, IppHintAlgorithm hint);
```

Description

These two functions initiate the specification for 2^{order} complex FFT, whose maximum length is 4096 and minimum length is 1.

These functions call <ippsMalloc>. The user is responsible for implementing <ippsMalloc>. Please refer to "Callback Function APIs" for detailed information

Input Arguments

- order – the base-2 logarithm of the length. Valid data value range is [0,12].
- flag – reserved argument, only 0 is supported.
- hint – reserved argument, only 0 is supported.

Output Arguments

- `pFFTSpec` – pointer to the FFT specification structure pointer that the specification hass allocated and initiated in this function.

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- The pointer `pFFTSpec` is NULL
- `order` <0 or `order` > 12

Specification Init of 2^N Vary-length Real<->Complex FFT

FFTInitAlloc_R_16s32s

Prototype

```
IppStatus ippsFFTInitAlloc_R_16s32s (IppsFFTSpec_R_32sc **pFFTSpec, int
   order, int flag, IppHintAlgorithm hint);
```

Description

This function initiates the specification for 2^{order} complex FFT, whose maximum length is 4096 and minimum length is 1.

This function calls `<ippsMalloc>` user is responsible for implementing it. Please refer to "Callback Function APIs" for detailed information.

Input Arguments

- `order` – the base-2 logarithm of the length. Valid data value range is [0,12].
- `flag` – reserved argument; only 0 is supported.
- `hint` – reserved argument; only 0 is supported.

Output Arguments

- `pFFTSpec` – pointer to the FFT specification structure pointer that the specification has allocated and initiated in this function.

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- The pointer `pFFTSpec` is NULL
- `order` <0 or `order` > 12

Buffer Size of 2^N Vary-length Complex to Complex FFT APIs

FFTGetBufSize_C_16sc
FFTGetBufSize_C_32sc

Prototype

```
IppStatus ippsFFTGetBufSize_C_16sc (const IppsFFTSpec_C_16sc *pFFTSpec,
   int *pSize);
IppStatus ippsFFTGetBufSize_C_32sc (const IppsFFTSpec_C_32sc *pFFTSpec,
   int *pSize);
```

Description

These functions obtain the size of work buffer used by 2^{order} complex FFT, whose maximum length is 4096 and minimum length is 1.

Input Arguments

- `pFFTSpec` – pointer to the FFT specification structure.

Output Arguments

- `pSize` – pointer to the size of the buffer (in bytes).

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- `pFFTSpec` or `pSize` is NULL

Buffer Size of 2^N Vary-length Real<->Complex FFT APIs

FFTGetBufSize_R_16s32s
FFTGetBufSize_R_32s

Prototype

```
IppStatus ippsFFTGetBufSize_R_16s32s (const IppsFFTSpec_R_16s32s
   *pFFTSpec,int *pSize);
IppStatus ippsFFTGetBufSize_R_32s (const IppsFFTSpec_R_32s *pFFTSpec,
   int *pSize);
```

Description

These functions obtain the size of the work buffer used by 2^{order} real FFT, whose maximum length is 4096 and minimum length is 1.

Input Arguments

- `pFFTSpec` – pointer to the FFT specification structure.

Output Arguments

- `pSize` – pointer to the size of the buffer (in bytes).

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- `pFFTSpec` or `pSize` is NULL

Specification Free of 2^N Vary-length Complex to Complex FFT

FFTFree_C_16sc
FFTFree_C_32sc

Prototype

```
IppStatus ippsFFTFree_C_16sc (const IppsFFTSpec_C_16sc *pFFTSpec);
IppStatus ippsFFTFree_C_32sc (const IppsFFTSpec_C_32sc *pFFTSpec);
```

Description

These functions free the memory used by the FFT specification.

Input Arguments

- `pFFTSpec` – pointer to the FFT specification structure.

Output Arguments

- None

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- `pFFTSpec` is NULL

Specification Free of 2^N Vary-length Real<->Complex FFT

FFTFree_R_16s32s
FFTFree_R_32s

Prototype

```
IppStatus ippsFFTFree_R_16s32s (const IppsFFTSpec_R_16s32s *pFFTSpec);
IppStatus ippsFFTFree_R_32s (const IppsFFTSpec_R_32s *pFFTSpec);
```

Description

These functions free the memory used by the FFT specification.

Input Arguments

- `pFFTSpec` – pointer to the FFT specification structure.

Output Arguments

- None

Return

- IPP_STATUS_OK – No Error.
- IPP_STATUS_BAD_ARG – Bad Arguments.

If following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.

- `pFFTSpec` is NULL

CallBack Functions for Memory Alloc and Free

Malloc

Prototype

```
void* _CALLBACK ippsMalloc(int size);
```

Description

This function allocates a memory block of a specific size.

Input Arguments

- `size` – the size of memory (in bytes) to be allocated.

Output Arguments

- None

Return

- Pointer to a valid allocated memory block if successful, or NULL if error occurs.

Free

Prototype

```
void  _CALLBACK ippsFree(void *pSrcBuf);
```

Description

This function frees a specified memory block.

Input Arguments

- `pSrcBuf` – pointer to the memory block to be freed

Output Arguments

- None

Return

- None

Callback Function APIs

Malloc

Prototype

```
void* _CALLBACK ippsMalloc(int size);
```

Description

This function allocates a memory block of a specified size.

Input Arguments

- `size` – size of memory (in bytes) to be allocated

Output Arguments

- None.

Returns

Pointer to a valid allocated memory block if successful, or NULL if error occurs.

Free

Prototype

```
void  _CALLBACK ippsFree(void *pSrcBuf);
```

Description

This function frees a specified memory block.

Input Arguments

- `pSrcBuf` – pointer to the memory block to be freed

Output Arguments

- None.

Returns

- None.

Usage Example of FFT Functions

Variable-length FFT Usage Example

The procedure of calling variable-length FFT functions is shown below:

1. Call `ippsFFTInitAlloc_C_16sc` to initialize the FFT specification.
2. Call `ippsFFTGetBufSize_C_16sc` to retrieve the buffer size.
3. Allocate pBuffer according to the size returned from `ippsFFTGetBufSize_C_16sc`.
4. Call forward FFT function `ippsFFTFwd_CtoC_16sc_Sfs`.
5. After calling the FFT function, `ippsFFTFree_C_16sc` should be called to free the memory allocated by `ippsFFTInitAlloc_C_16sc`.
6. `ippsFFTInitAlloc_C_16sc` and `ippsFFTFree_C_16sc` call two system-dependent callback functions, `ippsMalloc` and `ippsFree`. The user is responsible for implementing these two functions

Below is an example of forward FFT.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>
#include "ippSP.h"

#define ALIGN_8(addr) (((Ipp32u)(addr)+7)&(~7))

IppStatus fft();

/* Implement callback functions */
void* _CALLBACK ippsMalloc(int size)
{
    return malloc(size);
}

void _CALLBACK ippsFree(void* pSrcBuf)
{
    free(pSrcBuf);
}

int main()
{
    fft();
}

IppStatus fft()
{
     Ipp16sc *pX, *pXA, *pY, *pYA;
     int n, bufSize;
     IppStatus status;
     IppsFFTSpec_C_16sc* pSpec = NULL;
     Ipp8u *pWorkBuf = NULL;

     srand( (unsigned)time( NULL ) );

     /* Allocate aligned buffer for input  */
     pX = (Ipp16sc*)malloc(sizeof(Ipp16sc)*8 + 7);
     pXA= (Ipp16sc*)ALIGN_8(pX);
```

```
        /* Allocate aligned buffer for output */
        pY = (Ipp16sc*)malloc(sizeof(Ipp16sc)*8 + 7);
        pYA= (Ipp16sc*)ALIGN_8(pY);

        /* Fill in time-domain signals to input */
            for(n=0; n<8; n++)  {
               pXA[n].re = (Ipp16s)((rand()%1024)-512);
               pXA[n].im = (Ipp16s)((rand()%1024)-512);
            }

            /* Initialize FFT spec */
            status = ippsFFTInitAlloc_C_16sc(&pSpec, 3,
            IPP_FFT_NODIV_BY_ANY, ippAlgHintNone );

            /* Get working buffer size */
            status = ippsFFTGetBufSize_C_16sc(pSpec, &bufSize);
            /* Allocate working buffer */
            pWorkBuf= (Ipp8u*)malloc(bufSize);
            /* Call Forward FFT */
            status = ippsFFTFwd_CToC_16sc_Sfs ( pXA, pYA, pSpec, 0, pWorkBuf
            );

            /* Print output */
               for(n=0; n<8; n++)  {
                  printf("(%d, ", pYA[n].re);
                  printf("%d)\n", pYA[n].im);
               }

            /* Free allocated buffers */
               if(pSpec) {
                    ippsFFTFree_C_16sc ( pSpec );
                    }
                    free(pWorkBuf);
                    free(pX);
                    free(pY);

                    return status;
    }
```

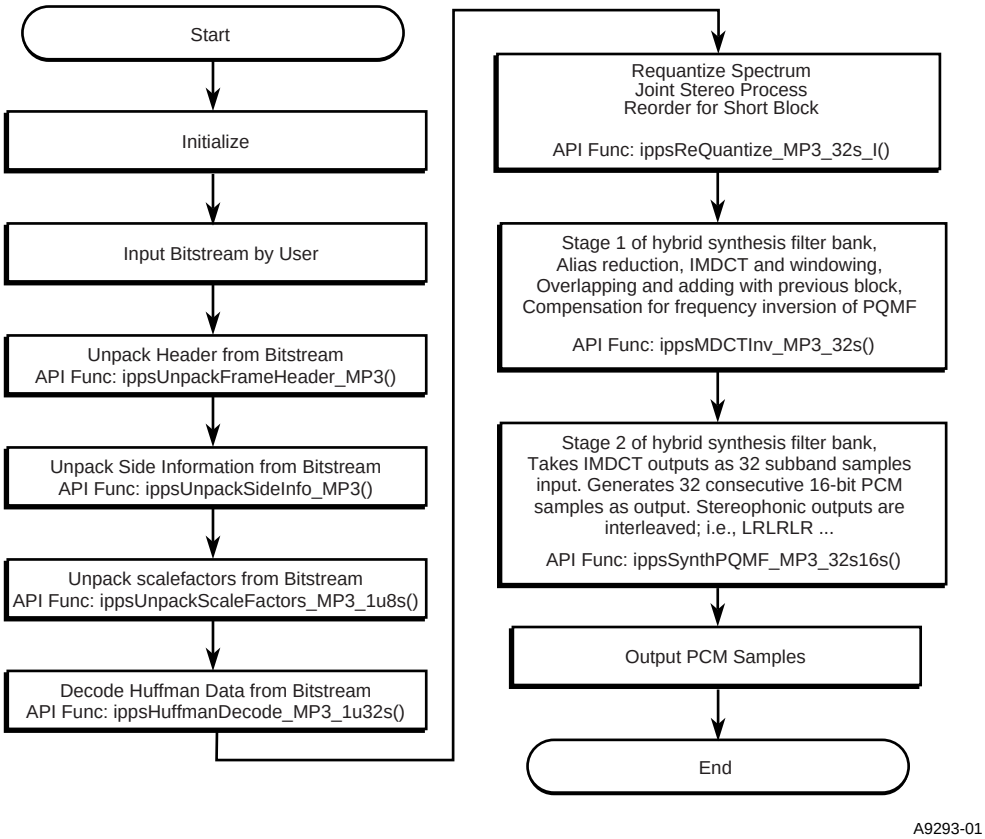
MP3 Audio Decoder

8

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) on Intel® PCA Processors with Intel® Wireless MMX™ (PCA processors with MMX™) that can be used to construct 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. These international standards for perceptual coding (compression) of digital audio are most often denoted by the "MP3" acronym. The MP3 algorithm delivers high-quality audio with bit rates as low as one-tenth of the original, making it a popular choice for cost-sensitive and bandwidth-constrained transmission and/or storage applications. At the same time, the MP3 decoder is characterized by a manageable computational complexity. As a result, MP3 has become the de-facto standard audio compression methodology for emergent portable and handheld digital media, as well as for distribution of high-fidelity compressed audio over networks such as the Internet.

This chapter provides a programmer's reference guide to the Intel® IPP MP3 audio decoder Application Programming Interface (API). As shown in Figure 8-1, the MP3 API consists of nine primitives, two data structures, and several predefined constants and macros.

Figure 8-1 Intel® IPP MP3 Decoder API



To benefit application developers, the design philosophy of the API emphasizes maximum flexibility. On the one hand, developers have the option of building a complete MP3 decoder solution using the compact set of performance optimized MP3 primitives described in this chapter in conjunction with a few administrative and memory management functions customized for the application environment. In this scenario, developers are able to leverage the fact that the MP3 primitives have been tuned carefully for minimum cycle count, minimum memory footprint, and maximum audio quality. On the other hand, developers also have the option of building a custom MP3 decoder while electing to use only a subset of the Intel® IPP MP3 primitives. This development option is facilitated in the API by providing access to the intermediate computational results generated by each of the core MP3 routines such as bit stream unpacking, Huffman

decoding, requantization, stereo processing, the Inverse Modified Discrete Cosine Transform (IMDCT) filter bank, and the Pseudo-Quadrature Mirror Synthesis filter (PQMF) bank. Moreover, the primitive API grants user access to data objects that have a direct and unambiguous mapping to the set of intermediate data objects defined in the ISO/IEC 11172-3 and 13818-3 layer 3 standards. Finally, the API allows the user to exploit fully performance properties of a particular target operating system (OS) by allowing user management of administrative functions such as the high-level bit stream manipulation, memory allocation/deallocation, and control of the various buffers.

The rest of this chapter provides details on the MP3 API and is organized as follows. First, the "Macros and Constants" section defines macros and constants. Next, the "Data Structures" section addresses data structures. Lastly, the "Primitives" section is on the function prototypes.

Macros and Constants

Table 8-1 MP3 Macro and Constant Definitions

Global Macro's Name	Definition	Notes
IPP_MP3_GRANULE_LEN	576	The number of samples in one granule
IPP_MP3_V_BUF_LEN	512	V data buffer length (32-bit words)
IPP_MP3_SF_BUF_LEN	40	Scalefactor buffer length (8-bit words)

Data Structures

The decoder API includes two 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.

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;           /* scalefactor scaling */
int  cntlTabSel;        /* Huffman table index for the <count1> quadruples */
} IppMP3SideInfo;
```

Primitives

There are ten primitives in the decoder API. The first three are concerned with bit stream processing. The primitive <ippsUnpackFrameHeader_MP3> unpacks the audio frame header from the raw input bit stream. The primitive <ippsUnpackSideInfo_MP3> extracts the side information associated with a single frame. The primitive <ippsUnpackScaleFactors_MP3_1u8s> extracts from the bit stream the spectral coefficient scalefactors associated with a one granule of one channel. The next two primitives provide Huffman decoding and requantization of the spectral coefficients. The primitive <ippsHuffmanDecode_MP3_1u32s> decodes the Huffman symbols from the raw bit stream that are associated for one granule of one channel. The primitive <ippsReQuantize_MP3_32s_I> requantizes the spectral samples associated with one granule of one or two channels. This primitive also provides joint stereo and dual_channel decoding. The last two primitives provide the two stages of the hybrid synthesis filter bank. Complete details for each primitive are given next.

UnpackFrameHeader_MP3

Prototype

```
IppStatus ippsUnpackFrameHeader_MP3(Ipp8u **ppBitStream,
   IppMP3FrameHeader *pFrameHeader);
```

Description

Unpacks the audio frame header. If CRC is enabled, this primitive also unpacks the CRC word. Before calling ippsUnpackFrameHeader_MP3, the decoder application should locate the bit stream sync word and ensure that *ppBitStream points to the first byte of the 32-bit frame header. If CRC is enabled, it is assumed 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 primitive updates the pointer *ppBitStream, such 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 primitive does not detect corrupted frame headers.

Input Arguments

ppBitStream – double pointer to the first byte of the MP3 frame header

Output Arguments

- `pFrameHeader` – pointer to the MP3 frame header structure (defined in section "Data Structures")
- `ppBitStream` – double pointer to the byte immediately following the frame header

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – invalid arguments – either `ppBitStream`, `pFrameHeader`, or `*ppBitStream` is Null

UnpackSideInfo_MP3

Prototype

```
IppStatus ippsUnpackSideInfo_MP3(Ipp8u **ppBitStream, IppMP3SideInfo
   *pDstSideInfo, int *pDstMainDataBegin, int *pDstPrivateBits, int
   *pDstScfsi, IppMP3FrameHeader *pFrameHeader);
```

Description

Unpacks the side information from the input bit stream. Before `ippsUnpackSideInfo_MP3` is called, the pointer `*ppBitStream` must point to the first byte of the bit stream that contains the side information associated with the current frame. Before returning to the caller, the primitive updates the pointer `*ppBitStream` such that it references the next byte after the side information.

Input Arguments

- `ppBitStream` – double pointer to the first byte of the side information associated with the current frame in the bit stream buffer
- `pFrameHeader` – pointer to the structure that contains the unpacked MP3 frame header. The header structure provides format information about the input bit stream. Both single- and dual_channel MPEG-1 and MPEG-2 modes are supported.

Output Arguments

- `pDstSideInfo` – pointer to the MP3 side information structure(s). The structure(s) contain(s) 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 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 scalefactor selection information associated with the current frame, 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) }.
- `ppBitStream` – double pointer to the bit stream buffer byte immediately following the side information for the current frame

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad argument; at least one of the following pointers is NULL: `ppBitStream`, `pDstSideInfo`, `pDstMainDataBegin`, `pDstPrivateBits`, `pDstScfsi`, `pFrameHeader`, and/or `*ppBitStream`
- `ippStsErr` – one or more elements of the MP3 frame header structure are invalid, i.e., one or more of the following conditions is true: `pFrameHeader->id` exceeds [0,1]; `pFrameHeader->layer!=1`; `pFrameHeader->mode` exceeds [0,3]; *block_type* is normal and *window_switching_flag* is set.

UnpackScaleFactors_MP3_1u8s

Prototype

```
IppStatus ippsUnpackScaleFactors_MP3_1u8s(Ipp8u **ppBitStream, int
   *pOffset, Ipp8s *pDstScaleFactor, IppMP3SideInfo *pSideInfo, int
   *pScfsi, IppMP3FrameHeader *pFrameHeader, int granule, int channel);
```

Description

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 primitive updates `*ppBitStream` and `*pOffset` such that they point to the next available bit in the input bit stream.

NOTE. *If the intensity position is equal to the maximum value of intensity position (an illegal position), the illegal position is set to negative. Thus, in the requantization module, negative positions indicate illegal positions. Those scalefactors that are not treated as intensity positions must be made positive before using them.*

Input Arguments

- `ppBitStream` – double pointer to the first bit stream buffer byte that is associated with the scalefactors for the current frame, granule, and channel
- `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.
- `pSideInfo` – pointer to the MP3 side information structure associated with the current granule and channel
- `pScfsi` – pointer to scalefactor selection information for the current channel
- `channel` – channel index; can take on the values of either 0 or 1
- `granule` – granule index; can take on the values of either 0 or 1
- `pFrameHeader` – pointer to MP3 frame header structure for the current frame

Output Arguments

- `pDstScaleFactor` – pointer to the scalefactor vector for long and/or short blocks
- `ppBitStrem` – updated double pointer to the next bit stream byte
- `pOffset` – updated pointer to the next bit in the bit stream (indexes the bits of the byte pointed to 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.

Returns

- `ippStsNoErr` – no error detected

- ippStsBadArgErr – bad arguments; one or more of the following pointers is NULL: *ppBitStream, pOffset, pDstScaleFactor, pSideInfo, pScfsi, *ppBitStream, or pFrameHeader. Bad arguments are also flagged when the value of *pOffset exceeds [0,7] or the granule or channel indices have values other than 0 or 1.
- ippStsErr – input data errors detected; one or more of the following are true: pFrameHeader ->id exceeds [0,1], pSideInfo->blockType exceeds [0,3], pSideInfo->mixedBlock exceeds [0,1]. pScfsi [0..3] exceeds [0,1]. If pFrameHeader->id indicates that the bit stream is MPEG-1, pSideInfo->sfCompress exceeds [0,15]. If pFrameHeader->id indicates the bit stream is MPEG-2, pSideInfo->sfCompress exceeds [0,511] or pFrameHeader->modeExt exceeds [0, 3].

HuffmanDecode_MP3_1u32s
HuffmanDecodeSfb_MP3_1u32s
HuffmanDecodeSfbMbp_MP3_1u32s

Prototype

```
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);
```

NOTE. ippsHuffmanDecodeSfbMbp_MP3_1u32s *was added in the IPP 4.1 Beta release. This function should be used for huffman decoding of the MP3 decoder when it is decoding a bit stream in the MPEG-2 LSF format.*

Description

Decodes Huffman symbols for the 576 spectral coefficients associated with one granule of one channel.

Input Arguments

- `ppBitStream` – double pointer to the first bit stream byte that contains the Huffman codewords associated with the current granule and channel
- `pOffset`– pointer to the starting bit position in the bit stream byte pointed 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
- `pSideInfo` – pointer to MP3 structure that contains the side information associated with the current granule and channel
- `pFrameHeader` – pointer to MP3 structure that contains the header associated with the current frame
- `hufSize` – the number of Huffman code bits associated with the current granule and channel
- `pSfbTableLong` – pointer to Scalefactor band table for long block. User can use the default table from MPEG-1 or MPEG-2 standards. User can also use his own table for special purpose. For table format see references [ISO93] Table B.8 and [ISO98] Table B.2.
- `pSfbTableShort` – pointer to Scalefactor band table for short block. User can use the default table from MPEG-1, MPEG-2 standards. User can also use his own table for special purpose. For table format see references [ISO93] Table B.8 and [ISO98] Table B.2.
- `pMbpTable` – pointer to Scalefactor band table for mixed block. User can use the default table from MPEG-1, MPEG-2 standards. User can also use his own table for special purpose.

Output Arguments

- `pDstIs` – pointer to the vector of decoded Huffman symbols used to compute the quantized values of the 576 spectral coefficients that are associated with the current granule and channel
- `pDstNonZeroBound` – pointer to the spectral region above which all coefficients are set equal to zero
- `ppBitStream` – updated double pointer to the particular byte in the bit stream that contains the first new bit following the decoded block of Huffman codes
- `pOffset` – updated pointer to the next bit position in the byte pointed 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

Returns

- `IPP_STATUS_OK` – no error detected

- `IPP_STATUS_BAD_ARG` – bad arguments detected; at least one of the following pointers is NULL: `ppBitStream`, `pOffset`, `pDstIs`, `pDstNonZeroBound`, `pSideInfo`, `pFrameHeader, pSfbTableLong`, or `*ppBitStream`. The flag is also asserted when either of the following is true: `*pOffset < 0`, or `*pOffset > 7`.
- `IPP_STATUS_ERROR` – indicates that the number of remaining Huffman code bits for <count1> partition is less than zero after decoding the <big_values> partition; alternatively, as shown in Table 8-2, the code could also indicate either that one or more elements of the MP3 side information are invalid or that one or more elements of the MP3 frame header are invalid.

Table 8-2 ippStsErr List

Input Data	Invalid Value	Condition
pSideInfo-> bigVals * 2	>IPP_MP3_GRANULE_LEN	None
pSideInfo->bigVals * 2	< 0	None
pSideInfo->winSwitch	Exceeds [0,1]	None
pSideInfo-> blockType	Exceeds [0,3]	None
pSideInfo->blockType	==0	1 == pSideInfo ->winSwitch
pSideInfo-> cnt1TabSel	Exceeds [0,1]	None
pSideInfo-> reg0Cnt	< 0	0 == pSideInfo -> blockType
		continued
pSideInfo-> reg1Cnt	< 0	0 == pSideInfo ->blockType
pSideInfo-> reg0Cnt + pSideInfo -> reg1Cnt + 2	> 22	0 == pSideInfo -> blockType
pSideInfo-> pTableSelect [0]	Exceeds [0,31]	None
pSideInfo-> pTableSelect[1]	Exceeds [0,31]	None
pSideInfo-> pTableSelect [2]	Exceeds[0,31]	0 == pSideInfo ->blockType
pFrameHeader-> id	Exceeds [0,1]	None
pFrameHeader-> layer	!=1	None
pFrameHeader-> samplingFreq	Exceeds [0,2]	None
hufSize	Exceeds [0, pSideInfo-> part23Len]	None

ReQuantize_MP3_32s_I
ReQuantizeSfb_MP3_32s_I

Prototype

```
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);
```

Description

Requantizes 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. Stereophonic mid/side (M/S) and/or intensity decoding is applied if necessary. Requantized spectral samples are returned in the vector pSrcDstIsXr. The reordering operation is applied for short blocks. Users must preallocate a workspace buffer pointed to by `pBuffer` prior to calling the requantization primitive. The value pointed by `pNonZeroBound` will be recalculated according to the output data sequence.

Input Arguments

- `pSrcDstIsXr` – pointer to the vector of decoded Huffman symbols; for stereo and dual_channel modes, right channel data begins at the address `&(pSrcDstIsXr[576])`
- `pNonZeroBound` – (Inout/output argument) 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 a workspace buffer. The buffer length must be 576 samples

- pSfbTableLong – pointer to Scalefactor band table for long block. User can use the default table from MPEG-1 or MPEG-2 standards. User can also use his own table for special purpose. For table format see references [ISO93] Table B.8, and [ISO98] Table B.2.
- pSfbTableShort – pointer to Scalefactor band table for short block. User can use the default table from MPEG-1 or MPEG-2 standards. User can also use his own table for special purpose. For table format see references [ISO93] Table B.8 and [ISO98] Table B.2.

Output Arguments

- pSrcDstIsXr – pointer to the vector of requantized spectral samples for the synthesis filter bank, in Q5.26 format (Qm.n defined in Chapter 1). Only the first (pNonZeroBound[ch]+17)/18 18-point blocks data are effctive. The others are meaningless at all.
- pNonZeroBound – (Input/output argument) 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].

Returns

- ippStsNoErr – no errors detected
- ippStsBadArgErr – bad arguments detected; one or more of the following pointers are NULL: pSrcDstIsXr, pNonZeroBound, pScaleFactor, pSideInfo, pFrameHeader or pBuffer.
- ippStsErr – one or more of the input error conditions listed in Table 8-3 is detected:

Table 8-3 ippStsErr List

Input Data	Invalid Value	Condition
pNonZeroBound [ch]	Exceeds [0,576]	None
pFrameHeader->id	Exceeds [0,1]	None
pFrameHeader -> samplingFreq	Exceeds [0,2]	None
pFrameHeader->mode	Exceeds [0,3]	None
pSideInfo [ch]. blockType	Exceeds [0,3]	None
pFrameHeader-modeExt	Exceeds [0,3]	None
pSideInfo [ch]. mixedBlock	Exceeds [0,1]	None
pSideInfo [ch] .globGain	Exceeds [0,255]	None
pSideInfo [ch]. sfScale	Exceeds [0,1]	None
pSideInfo [ch]. preFlag	Exceeds [0,1]	None
pSideInfo [ch]. pSubBlkGain [w]	Exceeds [0,7]	None

Table 8-3 ippStsErr List

Input Data	Invalid Value	Condition
|pSrcDstIsXr [i]|	>8206	None
pScaleFactor [sfb]	> 7	If pScaleFactor [sfb] is the intensity position for MPEG-1.
pSideInfo [ch]. blockType	pSideInfo [0]. blockType!= pSideInfo [1]. blockType	If the bit stream is joint stereo mode
pSideInfo[ch].mixedBlock	pSideInfo[0].mixedblock ! = pSideInf[1].mixedBlock	If the bit stream is joint stereo mode

NOTE. *In Table 8-3, the range on ch is from 0 to* `chNum-1`*, and the range on w is from 0 to 2, where chNum is the number of channels decoded by the* `pFrameHeader ->mode`*. If* `pFrameHeader ->mode == 3` *then* `chNum = 1`*, otherwise* `chNum = 2`*.*

MDCTInv_MP3_32s

Prototype

```
IppStatus ippsMDCTInv_MP3_32s(Ipp32s *pSrcXr, Ipp32s *pDstY, Ipp32s
   *pSrcDstOverlapAdd, int nonZeroBound, int *pPrevNumOfImdct, int
   blockType, int mixedBlock)
```

Description

Stage 1 of the hybrid synthesis filter bank.This performs the following operations: a) Alias reduction, b) Inverse MDCT according to block size specifiers and mixed block modes, c) Overlap add of IMDCT outputs, and d) Frequency inversion prior to PQMF bank. Because the IMDCT is a lapped transform, the user must 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 equal 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 will contain the overlapped portion of the output generated by operating on the current granule. Upon return from the primitive, the IMDCT sub-band output samples are organized as follows: `pDstY[j*32+subband]`, for j=0 to 17; subband=0 to 31.

NOTE. *The pointers pSrcXr (input argument) and pDstY (output argument) must reference different buffers.*

Input Arguments

`pSrcXr` – pointer to the vector of requantized spectral samples for the current channel and granule, represented in Q5.26 format.

NOTE. *The vector buffer is used as a workspace buffer when the input data has been processed. So the data in the buffer is meaningless when exiting the function*

- `pSrcDstOverlapAdd` – pointer to the overlap-add buffer; contains the overlapped portion of the previous granule's IMDCT output, in Q7.24 format
- `nonZeroBound` – the bound above which all spectral coefficients are zero for the current granule and channel
- `pPrevNumOfImdct` – pointer to the number of IMDCTs computed for the current channel of the previous granule
- `blockType` – block type indicator
- `mixedBlock` – mixed block indicator

Output Arguments

- `pDstY` – pointer to the vector of IMDCT outputs in Q7.24 format, for input to PQMF bank
- `pSrcDstOverlapAdd` – pointer to the updated overlap-add buffer in Q7.24 format; contains overlapped portion of the current granule's IMDCT output, in Q7.24 format
- `pPrevNumOfImdct` – pointer to the number of IMDCTs, for current granule, current channel

Returns

- `ippStsNoErr` – no errors detected
- `ippStsBadArgErr` – bad arguments detected; one or more of the pointers pSrcXr, pDstY, `pSrcDstOverlapAdd`, and/or `pPrevNumOfImdct` is NULL
- `ippStsErr` – one or more of the following input data errors detected: either `blockType` exceeds [0,3], `mixedBlock` exceeds [0,1], `nonZeroBound` exceeds [0,576], or `*pPrevNumOfImdct` exceeds [0,32]

SynthPQMF_MP3_32s16s

Prototype

```
IppStatus ippsSynthPQMF_MP3_32s16s(Ipp32s *pSrcY, Ipp16s *pDstAudioOut,
   Ipp32s *pVBuffer, int *pVPosition, int mode);
```

Description

Stage 2 of the hybrid synthesis filter bank; 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 2, the left and right channel output samples are interleaved (i.e., LRLRLR), such 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. Because the PQMF bank contains memory, the MP3 application must maintain two state variables in between calls to the primitive. First, the application must preallocate for the PQMF computation a workspace buffer of size 512 x Number of Channels. 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.

Input Arguments

- `pSrcY` – pointer to the block of 32 IMDCT sub-band input samples, in Q7.24 format

- pVBuffer – pointer to the input workspace buffer containing Q7.24 data. The elements of this buffer should be initialized to zero during decoder reset. During decoder operation, the values contained in this buffer should be modified only by the PQMF primitive.
- pVPosition – pointer to the internal workspace index; should be initialized to zero during decoder reset. During decoder operation, the value of this index should be preserved between PQMF calls and should be modified only by the primitive.
- mode – flag that indicates whether or not the PCM audio output channels should be interleaved

  1 – not interleaved

  2 – interleaved

Output Arguments

- pDstAudioOut – pointer to a block of 32 reconstructed PCM output samples in 16-bit signed format (little-endian); left and right channels are interleaved according to the mode flag. This should be aligned on a 4-byte boundary
- pVBuffer – pointer to the updated internal workspace buffer containing Q7.24 data; see usage notes under input argument discussion
- pVPosition – pointer to the updated internal workspace index; see usage notes under input argument discussion

Returns

- ippStsNoErr – no errors detected
- ippStsBadArgErr – bad arguments detected; either mode < 1, or mode > 2, or at least one of the following pointers is NULL: pSrcY, pDstAudioOut, pVBuffer, and/or pVPosition.
- ippStsErr – the value of *pVPosition exceeds [0, 15]

MP3 Audio Encoder

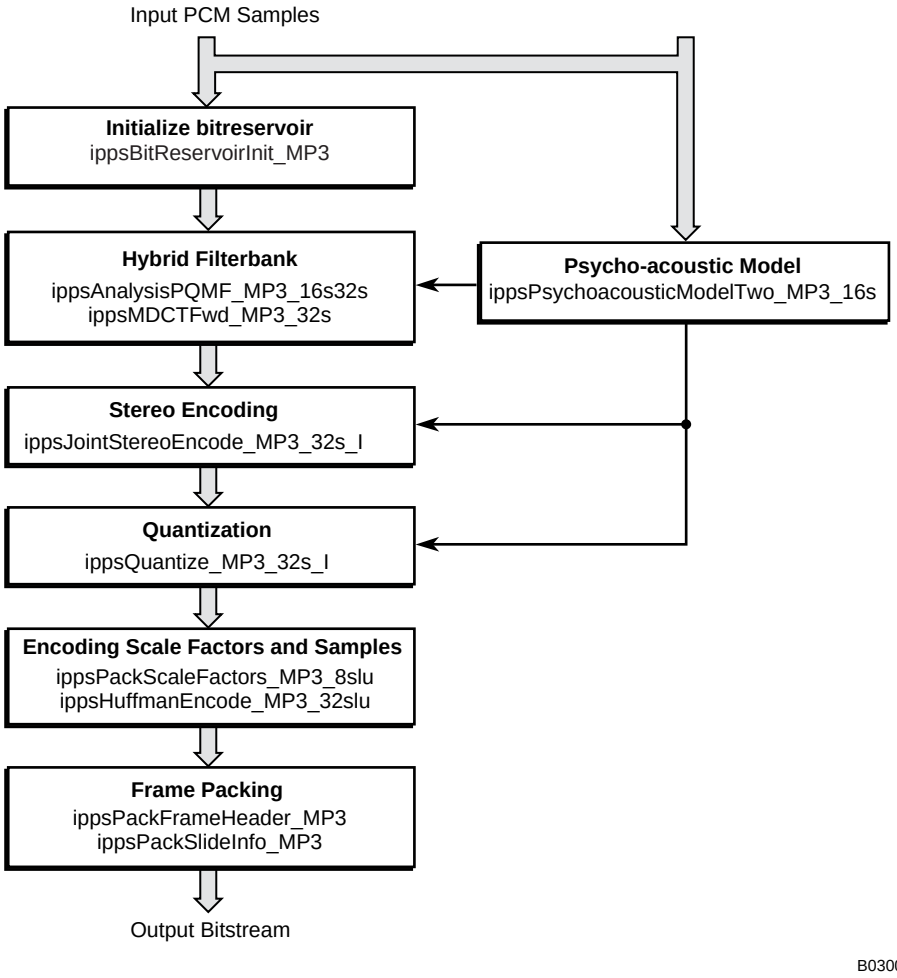
9

The ISO/IEC 11172-3 MPEG-1, Layer III (so-called “MP3”) audio coding algorithms are widely used to compress, respectively, stereophonic and dual-channel music signals. Ideally suited for both transmission and storage applications, the MP3 algorithm delivers high-fidelity audio playback quality with bit rates 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 is widely used in music storage and audio recording.

The MP3 encoder Application Programming Interface (API) provides a variety of capabilities, including bit stream packing functions and MP3 core encoding functions, See [ISO-11172]. The API enables customers to develop a variety of applications quickly using Intel XScale® microarchitecture.

This chapter provides a programmer’s reference guide to the Intel Integrated Performance Primitives (Intel® IPP) MP3 audio encoder API. As shown in Figure 9-1, this API includes several functions as well as predefined macros and constants.

Figure 9-1 Intel® IPP MP3 Encoder API (Flowchart of MP3 Encoder)



Files and Libraries

This section describes the definitions and header files of the Intel® IPP MP3 audio encoder API.

Header Files

User must include <ippdefs.h> and <ippAC.h> at the beginning of the source code before using any IPP functions, as shown in the following example:

```
#include "ippdefs.h"
#include "ippAC.h"
int main()
{
      ..
      /* call MP3 encoder IPP functions */
      ippsAnalysisPQMF_MP3_16s32s (pPcm, pXr, pcmMode);
      ...
}
```

Binary Libraries

The Intel® IPP MP3 audio coding binary library must be referenced by the linker when building an application that references any of the Intel® IPP MP3 encoder primitives.

Two different versions are provided in the installation package. One is the debug version, which provides any debug information. The other is the release version, which is used to link in applications. The binary library for the Pocket PC operating system is named: "ippAC_WMMX40APPC_r.lib".

Macros

A macro, in the language-independent context, is the defined symbolic name that is substituted with particular replacement text at compilation or assembly time. A macro can be a numeric or string constant, as well as functional unit.

Common Macros

The <ippdefs.h> header file contains the following constants in Table 9-1.

Table 9-1 Common Macros

Macros	Defined As	Description
TRUE	1	Logic true
FALSE	0	Logic false
NULL	((void *)0)	NULL pointer
IPP_MAX_16S	32767	The maximum 16-bit signed integer
IPP_MIN_16S	-32768	The minimum 16-bit signed integer
IPP_MAX_32S	2147483647	The maximum 32-bit signed integer
IPP_MIN_32S	-2147483648	The minimum 32-bit signed integer

Flags

Flags are predefined arguments or return values used by the API functions. The file <ippdefs.h> contains the following flags in Table 9-2.

Table 9-2 Flag Macros

Macros	Used In	Description
ippStsNoErr	Status code for all functions	No error
ippStsBadArgErr		Bad argument(s)
ippStsErr		Some errors exists
ippStsNoMemErr		Out of memory

Data Types and Structures

This following sections describe the data types and definitions of the Intel® IPP MP3 audio encoder API.

General Data Types

The Intel XScale® microarchitecture supports only integer data types only, This means that the parameters and return values of the API functions must be of integer type (including 1/2/4/8-byte integer, integer-like structure and pointer). The most frequently used data types used in the API are described in Table 9-3.

Table 9-3 Common Data Types Used

Data Type	Corresponding Data Type in C	Corresponding Data Type in ARM* Assembly
8-bit unsigned integer	Ipp8u	Byte
16-bit integer	Ipp16s	Signed Halfword
32-bit integer	Ipp32s	Signed Word
64-bit integer	Ipp64s	Signed Double Word
32-bit IPP status code	typedef Ipp32s IppStatus;	Signed Word
Pointer	any_type [Note]	Unsigned Word

Note: The term "any_type" refers to any type that is specified in Table 9-3.

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 9-4, the MP3 CODEC API includes several enumerated types that provide semantic interpretations for frequently used constants.

Table 9-4 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

continued

Table 9-4 MP3 Enumerated Data Types (continued)

Enumerated Type Name	Symbolic Values	Constant Value
IppMP3SampleRate	ippMP3SampleRate32000	2
	ippMP3SampleRate44100	0
	ippMP3SampleRate48000	1
IppMP3PcmMode	ippMP3NonInterleavedPCM	1
	ippMP3InterleavedPCM	2
IppMP3Emphasis	IppMP3EmphasisNone	0
	IppMP3Emphasis5015	1
	IppMP3EmphasisReserved	2
	IppMP3CCITTJ17	3

MP3 CODEC Data Structures

The following structures are used in the API definitions of the MP3 encoder Intel® IPP functions.

IppMP3FrameHeader Structure

Structure <IppMP3FrameHeader> contains the complete header information for one frame, listed below:

```
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 switch flag 0: CRC on, 1: CRC off */
   int bitRate;     /* bit rate index. 0,  */
   int samplingFreq; /* sampling frequency index */
   int paddingBit;   /* padding flag 0: no padding, 1 padding  */
   int privateBit;  /* free bit available for private. ISO/IEC guarantees
                    that this bit will not be used in future */
   int mode;        /* mono/stereo indicator. 1:mono, 2:stereo
                    */
    int modeExt;    /* extension to mode. 00: none, 10, mid/state coding
                    enabled. 01: intensity coding enabled. 11:
                    both mid/state and intensity coding enabled*/
   int copyright;   /* copyright protected or not, 0: no, 1: yes  */
   int originalCopy;/* original or copied, 0: copy, 1: original */
   int emphasis;    /* de-emphasis type indicator */
```

```
    int CRCWord;     /* CRC-check word */

} IppMP3FrameHeader;
```

The <emphasis> values are shown in Table 9-5.

Table 9-5 Emphasis Value

Emphasis Value	Emphasis Type
0	none
1	50/15 microseconds
2	reserved
3	CCITT J.17

IppMP3SideInfo Structure

Structure <IppMP3SideInfo> contains all the side information need to decode one granule. MP3 granule side information is comprised of parameters that describe how to decode the scalefactors and Huffman-encoded spectral coefficients.

```
typedef struct {
    int  part23Len;  /* the number of main_data bits used for scalfactor
                        and Huffman-encoded spectral coefficients*/
    int  bigVals;    /* the half number of Huffman data whose maximum amplitudes
                        cam be greater than 1. */
    int  globGain;   /* logarithmically quantized step size information */
    int  sfCompress; /* information to select the number of bits used
                        for the transmission of the scale factors */
    int  winSwitch;  /* switching flag; 0:normal window, 1:check blockType */
    int  blockType;  /* window type indicator: 1:start, 2:short, 3:stop */
    int  mixedBlock; /* flag 0: non mixed block, 1: mixed block */
    int  pTableSelect[3];/* Huffman table index for the 3 regions in
                            <big_values> field */
    int  pSubBlkGain[3]; /* gain offset from the global gain for one
                            subblock */
    int  reg0Cnt;    /* one less than the number of scalefactor bands in
                        the first region of the <big_values> field */
    int  reg1Cnt;    /* one less than the number of scalefactor bands in
                        the second region of the <big_values> field */
    int  preFlag;    /* preemphasis flag: 1=enable high frequency boost */
```

```
   int  sfScale;    /* scalefactor log quantization parameter:
                    0:scalefac_multiplier=0.5, 1:scalefac_multipler=1 */
   int  cnt1TabSel; /* Huffman table index for the <count1> field of
                    quadruples 0:Table B.7 -A, 1:Table B.7-B (ISO/IEC
   11172-3)*/
} IppMP3SideInfo;
```

IppMP3PsychoAcousticModelTwoAnalysis Structure

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). he structure is defined as follows.

```
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;
```

IppMP3PsychoAcousticModelTwoState Structure

The structure <IppMP3PsychoAcousticModelTwoState> contains the state information associated with the Intel® IPP implementation of ISO/IEC 11172-3 psychoacoustic analysis model 2. This facilitates coherent block processing. The structure is defined as follows:

```
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[42]; /* short block masked threshold
        history buffer; Contains masked threshold estimates for the threshold
        calculation partitions associated with the most recent short block */

   Ipp32sc pPrevFFT[2][6];/* FFT history buffer; Contains real and imaginary
        FFT components associated with the two most recent long blocks */
```

```
    Ipp32s  pPrevFFTMag[2][6];/* 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;
```

IppMP3BitReservoir Structure

The structure `<IppMP3BitReservoir>` contains the state information associated with the quantization bit reservoir. The structure is defined as follows:

```
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 Audio Encoder Primitives

This following sections describe the MP3 audio encoder API primitives.

AnalysisPQMF_MP3_16s32s

Prototype

```
IppStatus ippsAnalysisPQMF_MP3_16s32s (const Ipp16s *pSrcPcm, Ipp32s
   *pDstXs, int pcmMode);
```

Description

This function implements stage 1 of the MP3 hybrid analysis filterbank. It 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 (36 times per channel on each frame).

Input Arguments

- `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
  - the most recent 480 (512-32) samples should be contained in the vector `pSrcPcm[pcmMode*i]`, where i=0,1,..,479
  - the samples associated with the current granule should be contained in the vector `pSrcPcm[pcmMode*j]`, where j=480,481,..,1055
- `pcmMode` – The PCM mode flag; informs the PQMF filterbank of which type of input PCM vector organization to expect. These can be:
  - pcmMode=1 denotes non-interleaved PCM input samples
  - pcmMode=2 denotes interleaved PCM input samples

Output Arguments

- `pDstXs` – Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples. These are indexed as follows:
  - pDstXs[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

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. At least one of the following pointer arguments is NULL: `pSrcPcm`, `pDstXs`, `pDelayLine` or `pDelayLineIndex`.

MDCTFwd_MP3_32s

Prototype

```
IppStatus ippsMDCTFwd_MP3_32s (const Ipp32s *pSrcXs, Ipp32s *pDstXr, int
   blockType, int mixedBlock, IppMP3FrameHeader *pFrameHeader, Ipp32s
   *pOverlapBuf);
```

Description

This function implements stage 2 of the MP3 hybrid analysis filterbank. It performs 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 stage I 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).

Input Arguments

- `pSrcXs` – 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, and sb=0,1,…,31 is the subband index. All coefficients are represented using the Q7.24 format.
- `blockType` – Block type indicator. 0: normal block, 1: start block, 2: short block, 3: stop block.

- `mixedBlock` – Mixed block indicator. 0: not mixed, 1: 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.

Output Arguments

pDstXr – Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank. All coefficients are represented using the Q5.26 format.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. At least one of the following pointer arguments is NULL: `pSrcXs`, `pDstXr`, `pFrameHeader` or `pOverlapBuf`

PsychoAcousticModelTwo_MP3_16s

Prototype

```
IppStatus ippsPsychoAcousticModelTwo_MP3_16s(const Ipp16s *pSrcPcm,
   IppMP3PsychoAcoustic ModelTwoAnalysis *pDstPsychoAcousticModelOutput,
   int *pDstIsSfbBound, IppMP3SideInfo *pDstSideInfo, IppMP3FrameHeader
   *pFrameHeader, IppMP3PsychoAcousticModelTwoState
   *pPsychoAcousticModelState, int pcmMode, Ipp32s *pWorkBuffer);
```

Description

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. Model outputs are used during the quantization process 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 (1152 samples per channel = 2 granules * 576 samples per granule) the psychoacoustic model generates the following outputs:

1. Estimated SFB (scalefactor 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, 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. Then, 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 scalefactor 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). Finally, 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 (That is, 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. Short block mode is enabled when pre-echoes are likely; otherwise long blocks are selected.

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 will change 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 joint stereophonic processing mode decision, above), 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. Details are given below on an argument-by-argument basis.

This function is performed on a frame base. The arguments have different sizes for mono and stereo coding.

Input Arguments

- `pSrcPcm` – Pointer to the start of the buffer containing the input PCM audio vector, the samples of which should adhere to the following format specification: 16-bits per sample, signed, little-endian, Q15. The `pSrcPcm` buffer should contain 1152 (= 2 granules x 576 samples/granule) samples if the parameter `pFrameHeader->mode` has the value of 1 (mono), or 2304 (= 2 granules x 2 channels x 576 samples/granule) 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 will result in undefined model outputs.
- `pFrameHeader` – Pointer to the `IppMP3FrameHeader` structure that contains the header associated with the current frame. The `samplingFreq`, id, and mode fields of the structure *pFrameHeader control the behavior of the psychoacoustic model. All three fields must be appropriately initialized prior to calling this function. All other frame header fields are ignored. Only MPEG-1 (id=1) is supported.

- pPsychoAcousticModelState – Pointer to the first element in a set of IppMP3PsychoAcousticModelTwoState structures that contains the psychoacoustic model state information associated with both the previous and the current frames. 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.
- pcmMode – PCM mode flag; informs the psychoacoustic model of 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.

  As an alternative to the constants 1 and 2, appropriately typecast elements ippMP3NonInterleavedPCM and ippMP3InterleavedPCM of the enumerated type IppMP3PcmMode can also be used for pcmMode.
- pWorkBuffer – Pointer to a 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 (6300 elements of type Ipp32s).

Output Arguments

- pDstPsychoAcousticModelOutput – 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 2 ´ 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. Because the intensity coding SFB lower bound is block-specific, the number of valid elements pointed to by pDstIsSfbBound will vary 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, and pIsSfbBound[3*gr + w] for short block granules, where gr is the granule index (0=granule 1, 1=granule 2), and w is the block index (0=block 1, 1=block 2, 2=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 `pDstPsychoAcousticModelOutput`.
- `pPsychoAcousticModelState` – 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 pyschoacoustic model state structure *`pPsychoAcousticModelState` should be initialized to contain the value 0. In the ippSP function domain, this could be accomplished using the function `ippsZero_16s` as follows:

  ```
  ippsZero_16s ((Ipp16s *)
  pPsychoAcousticModelState,sizeof(IppMP3PsychoAcousticModelTwoState)/
  sizeof(Ipp16s)).
  ```
- `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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. At least one of the following pointer arguments is NULL: `pSrcPcm`, `pPsychoAcousticModelState`, `pDstPsychoAcousticModelOutput`, `pFrameHeader`, `pSideInfo`, or `pDstIsSfbBound`.

JointStereoEncode_MP3_32s_I

Prototype

```
IppStatus ippsJointStereoEncode_MP3_32s_I (Ipp32s *pSrcDstXrL, Ipp32s
   *pSrcDstXrR, Ipp8s *pDstScaleFactorR, IppMP3FrameHeader
   *pFrameHeader, IppMP3SideInfo *pSideInfo, int *pIsSfbBound);
```

Description

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 has been 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. Call it for every granule / 2 channels.

If intensity coding has been enabled (`pFrameHeader->modeExt` & 0x01 = 1) the left channel is used to carry the intensity data for SFBs above the SFB intensity lower bound, and the right channel above the SFB lower bound is cleared (all coefficients = 0).

$$L = L + R, \quad R = 0$$

In order to facilitate energy-proportional recovery of the left and right spectral coefficients at the decoder, an intensity energy scalefactor, is_pos, is transmitted in place of the right channel scalefactor since the right channel spectral coefficients above the SFB bound have been eliminated. The energy normalization constant is derived from the L/R SFB energy ratios and then transformed to improve its quantization properties. That is:

$$is\_pos = n\text{int}\left(\frac{12}{\pi}\arctan\left(\sqrt{\frac{L\_energy}{R\_energy}}\right)\right)$$

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 scalefactor is used

to apportion jointly coded signal energy between 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 `ippsJointStereoEncode_MP3_32s_I` must therefore be called once per granule, or twice per frame.

Input Arguments

- `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 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 will be 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; it is represented in `pIsSfbBound[0]`. 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.

Output Arguments

- `pSrcDstXrL` – Pointer to the 576-element joint stereo spectral coefficient output vector associated with the M channel, as well as the intensity coded coefficients above the intensity lower SFB bound. All elements are represented using the Q5.26 format.
- `pSrcDstXrR` – Pointer to the 576-element joint stereo spectral coefficient output vector associated with the S channel. All elements are represented using the Q5.26 format.
- `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`), then `ippsStereoEncode_MP3_32s_I` updates with the appropriate scalefactors those elements of pDstScaleFactorR[] that are associated with intensity coded scalefactor bands. Other SFB entries in the scalefactor vector are not modified. The length of the vector referenced by `pDstScaleFactorR` varies as a function of block size. It contains 21 elements for long block granules, or 36 elements for short block granules.

Returns

- ippStsNoErr – no error
- `ippStsBadArgErr` – `bad arguments.` At least one of the following pointer arguments is NULL: `pSrcDstXrL`, `pSrcDstXrR`, `pFrameHeader`, `pSideInfo`, or `pIsSfbBound`.

Quantize_MP3_32s_I

Prototype

```
IppStatus ippsQuantize_MP3_32s_I (Ipp32s *pSrcDstXrIx, Ipp8s
   *pDstScalefactor, int *pDstScfsi, int *pDstCount1Len, int
   *pDstHufSize, IppMP3FrameHeader *pFrameHeader, IppMP3SideInfo
   *pSideInfo, IppMP3PsychoAcousticModelTwoAnalysis
   *pPsychoAcousticModelOutput, IppMP3PsychoAcousticModelTwoState
   *pPsychoAcousticModelState, IppMP3BitReservoir *pBitResv, int
   meanBits, int *pIsSfbBound, Ipp32s *pWorkBuffer);
```

Description

This function quantizes the spectral coefficients generated by the analysis filterbank such that the resulting distortion (quantization noise) is shaped to match a profile derived from the masked thresholds estimated by the psychoacoustic model. While satisfying these 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 (2 granules, 1 or 2 channels), and it should therefore be called once per frame.

Input Arguments

- `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, and `pSrcDstXrIx`[gr*576 + i] for monaural (single channel) input sources, where i=0,1,…,575 is the spectral coefficient index, gr is the granule index (0=granule 1, 1=granule 2), and ch is the channel index (0=channel 1, 1=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.
- `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 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=granule 1, 1=granule 2), nchan is the number of channels, and ch is the channel index (0=channel 1, 1=channel 2). Upon 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 function entry and updated upon function exit, as described below under the description of output arguments.
- `pPsychoAcousticModelOutput` – 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: `pPsychoAcousticModelOutput[gr*nchan+ch]`, where gr is the granule index (0=granule 1, 1=granule 2), nchan is the number of channels, and ch is the channel index (0=channel 1, 1=channel 2).

- `pPsychoAcousticModelState` – 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 lookahead information (nextBlockType) to manage the bit reservoir. All other structure elements are ignored by the quantizer.
- `pBitResv` – Pointer to the `IppMP3BitReservoir` structure that contains the bit reservoir state information. Upon 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=granule 1, 1=granule 2), and w is the block index (0=block 1, 1=block 2, 2=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 a 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 (720 32-bit words).

Output Arguments

- `pSrcDstXrIx` – Pointer to the output set of quantized spectral coefficient vectors. These are suitable for input to the Huffman encoder. The coefficients are indexed as follows: `pSrcDstXrIx`[gr*1152 + ch*576 + i] for stereophonic and dual-mono input sources, and `pSrcDstXrIx`[gr*576 + i] for monaural (single channel) input sources, where i=0,1,…,575 is the spectral coefficient index, gr is the granule index (0=granule 1, 1=granule 2), and ch is the channel index (0=channel 1, 1=channel 2).
- `pDstScaleFactor` – Pointer to the output set of scalefactors generated during the quantization process. These scalefactors determine the quantizer granularity. Scalefactor vector lengths depend on the block mode associated with each granule. The order of the elements is: (granule 1, channel 1), (granule 1, channel 2), (granule 2, channel 1), (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 scalefactor vector indices and lengths.
- `pDstScfsi` – Pointer to the output vector of scalefactor 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 scalefactor 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=channel 1, 1=channel 2), and scfsi_band is the scalefactor selection group number (group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, and 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 above (of higher frequency than) the bigvals region, the count1 parameter indicates the size of the region in which spectral samples can be combined into 4-tuples 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=granule 1, 1=granule 2), nchan is the number of channels, and ch is the channel index (0=channel 1, 1=channel 2)
- `pDstHufSize` – Pointer to an output vector of Huffman coding bit allocation specifiers. For each granule/channel, these indicate the total number of Huffman bits that are 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 maximum capacity, it is necessary for the quantizer to expend 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]`=number of bits required for Huffman symbols + 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=granule 1, 1=granule 2), nchan is the number of channels, and ch is the channel index (0=channel 1, 1=channel 2).

- pSideInfo – Pointer to the set of updated IppMP3SideInfo side information structures. In all set elements, the quantizer modifies the following structure fields: part23Len, bigVals, globGain, sfCompress, pTableSelect[0]-[2], pSubBlkGain[0]-[2], reg0Cnt, reg1Cnt, sfScale, preFlag, and cnt1TabSel. Detailed functional descriptions of the fields are given in the side information structure discussion. The set contains 2*nchan, elements and is indexed as follows: pSideInfo[gr*nchan+ch], where gr is the granule index (0=granule 1, 1=granule 2), nchan is the number of channels, and ch is the channel index (0=channel 1, 1=channel 2).
- pBitResv – Pointer to the updated IppMP3BitReservoir structure. The quantizer updates the BitsRemaining field to add or remove bits as necessary. All other fields are unmodified by the quantizer.

Returns

- ippStsNoErr – no error
- ippStsBadArgErr– bad arguments. At least one of the pointer arguments is NULL.

PackScalefactors_MP3_8s1u

Prototype

```
IppStatus ippsPackScalefactors_MP3_8s1u (const Ipp8s *pSrcScalefactor,
   Ipp8u **ppBitStream, int *pOffset, IppMP3FrameHeader *pFrameHeader,
   IppMP3SideInfo *pSideInfo, int *pScfsi, int granule, int channel);
```

Description

This function applies noiseless (lossless) coding to the scalefactors and then packs the output into the bit stream buffer. This function operates on one channel of one granule at a time, and it therefore must be called once for each channel of each granule. The resulting bit stream is fully compliant with the syntax specified in ISO/IEC 11172-3.

- pSrcScaleFactor – Pointer to a vector of scalefactors generated during the quantization process for one channel of one granule. Scalefactor vector lengths depend on the block mode; short block granule scalefactor vectors contain 36 elements (12 per subblock), and long block

granule scalefactor vectors contain 21 elements. Thus, short block scalefactor vectors are indexed as follows: `pSrcScaleFactor`[sb*12+sfb], where sb is the subblock index (0=subblock 1, 1=subblock 2, 2=subblock 3) and sfb is the scalefactor band index (0-11), and long block scalefactor vectors are indexed as follows: `pSrcScaleFactor[sfb]`, where sfb is the scalefactor band index (0-20). The associated side information for an individual granule/channel can be used to select the appropriate indexing scheme.

- `ppBitStream` – Pointer to the encoded bit stream buffer. The ppBitStream parameter is a double pointer to the first byte in the bit stream buffer intended to receive the Huffman-encoded scalefactor bits generated by the function `ippsEncodeScaleFactors_MP3_8s1u`. The scalefactor Huffman bits are sequentially written into the stream buffer starting from the bit indexed by the combination of byte pointer *`ppBitStream` and bit pointer `pOffset`.
- `pOffset` – Bit stream bit pointer. Indexes the next available bit in the byte referenced by `*ppBitStream`. The `pOffset` parameter indexes the next available bit in the byte referenced by *`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 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 scalefactor bit allocation. All other *`pSideInfo` fields are ignored by the scalefactor encoder.
- `pScfsi` – Pointer to the scale factor selection information table. This contains the set of binary flags that indicate whether or not scalefactors are being shared across granules of a frame within the predefined scalefactor 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: `pScfsi[ch][scfsi_band]`, where ch is the channel index (0=channel 1, 1=channel 2), and scfsi_band is the scalefactor selection group number (group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, and group 3 includes SFBs 16-20).
- `granule` – Index of the current granule (0=granule 1, 1=granule 2).
- `channel` – Index of the current channel (0=channel 1, 1=channel 2).

Output Arguments

- `ppBitStream` – Updated bit stream byte pointer. This parameter points to the first available bit stream buffer byte immediately following the bits generated by the scalefactor Huffman encoder and sequentially written into the stream buffer. The scalefactor bits are formatted according to the bit stream syntax given in *ISO/IEC 11172-3*.
- `pOffset` – Updated bit stream bit pointer. The `pOffset` parameter indexes the next available bit in the next available byte referenced by the updated bit stream 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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. At least one of the pointer arguments is NULL, or either `granule` or `channel` contains an illegal value.

HuffmanEncode_MP3_32s1u

Prototype

```
IppStatus ippsHuffmanEncode_MP3_32s1u (Ipp32s *pSrcIx, Ipp8u
   **ppDstBitStream, int *pOffset, IppMP3FrameHeader *pFrameHeader,
   IppMP3SideInfo *pSideInfo, int count1Len, int hufSize);
```

Description

This function applies noiseless (lossless) Huffman encoding to the quantized samples and packs the output into the bit stream buffer. This function encodes one granule at a time, and therefore must be called once for each granule of each channel. The resulting bit stream is fully compliant with *ISO/IEC 11172-3*.

Input Arguments

- `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 – Bit stream byte pointer. The ppBitStream parameter is a double pointer to the first byte in the bit stream buffer intended to receive the Huffman-encoded spectral coefficient bits generated by this function. The Huffman-encoded spectral coefficient bits are sequentially written into the stream buffer starting from the bit indexed by the combination of byte pointer *ppBitStream and bit pointer pOffset.
- pOffset – Bit stream bit pointer. The pOffset parameter indexes the next available bit in the byte referenced by *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 bigvals 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 bigvalues 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. Detailed descriptions of all side information elements are given in the structure definition header file.
- count1Len – The count1 region length specifier; indicates the number of spectral samples for the current granule/channel above the bigvals 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 ippsQuantize_MP3_32s_I reflects these padding requirements. That is, HufSize[i]=number of bits required for Huffman symbols + number of padding bits.

Output Arguments

- ppBitStream – Updated bit stream byte pointer. The parameter *ppBitStream points to the first available bit stream 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 bit stream syntax given in *ISO/IEC 11172-3*.

- pOffset – Updated bit stream bit pointer. The pOffset parameter indexes the next available bit in the next available byte referenced by the updated bit stream 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.

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments. At least one of the pointer arguments is NULL, or either granule or channel contains an illegal value.

PackFrameHeader_MP3

Prototype

```
IppStatus ippsPackFrameHeader_MP3 (IppMP3FrameHeader *pSrcFrameHeader,
   Ipp8u **ppBitStream);
```

Description

This function packs the content of the frame header into the bit stream. The resulting bit stream is fully compliant with the syntax specified in ISO 11172-3. This function should be called once per frame.

Input Arguments

- 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 – Pointer to the encoded bit stream buffer. The ppBitStream parameter is a double pointer to the first byte in the bit stream buffer intended to receive the packed frame header bits generated by this function. The frame header bits are sequentially written into the stream buffer starting from the bit indexed by the combination of byte pointer *ppBitStream and bit pointer pOffset.

Output Arguments

`ppBitStream` – Updated bit stream byte pointer. The parameter *`ppBitStream` points to the first available bit stream buffer byte immediately following the packed frame header bits. The frame header bits are formatted according to the bit stream syntax given in *ISO/IEC 11172-3*.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. At least one of the pointer arguments is NULL.

PackSideInfo_MP3

Prototype

```
IppStatus ippsPackSideInfo_MP3 (IppMP3SideInfo *pSrcSideInfo, Ipp8u
   **ppBitStream, int mainDataBegin, int privateBits, int *pSrcScfsi,
   IppMP3FrameHeader *pFrameHeader);
```

Description

This function packs the side information into the bit stream buffer. The resulting bit stream is fully compliant with the syntax specified in ISO 11172-3. This function should be called once per frame.

Input Arguments

- `pSrcSideInfo` – Pointer to the `IppMP3SideInfo` structures. This should contain 2 * channel number of elements. The order is (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 function entry.
- `mainDataBegin` – Negative bit stream offset, in bytes.The value of the parameter `mainDataBegin` is typically the number of bytes remaining in the bit reservoir before the start of quantization for the current frame. An example of how to compute an appropriate value for `mainDataBegin` is given in the MP3 encoder sample program. Header and side information bytes should be excluded from the mainDataBegin computation. The side information formatter packs the 9-bit value of mainDataBegin into the `main_data_begin` field of the output bit stream.

- `privateBits` – Depending on the number of channels, the function `ippsPackSideInfo_MP3` extracts the appropriate number of least significant bits from the parameter `privateBits` and packs them into the `private_bits field` of the output bit stream. The ISO/IEC 11172-3 bit stream syntax reserves a channel-dependent number of application-specific (private) bits in the layer III bit stream 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 or not scalefactors are shared across granules of a frame within predefined scalefactor 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 [2]). The vector is indexed as follows: `pDstScfsi[ch][scfsi_band]`, where ch is the channel index (0=channel 1, 1=channel 2), and scfsi_band is the scalefactor selection group number (group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, and group 3 includes SFBs 16-20).
- `pFrameHeader` – Pointer to the `IppMP3FrameHeader` structure. Only MPEG-1 (id=1) is supported. Upon 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.
- `ppBitStream` – Pointer to the encoded bit stream buffer. The `ppBitStream` parameter is a double pointer to the first byte in the bit stream buffer intended to receive the packed side information bits generated by this function. The side information bits are sequentially written into the stream buffer starting from the byte-aligned location referenced by *`ppBitStream`.

Output Arguments

`ppBitStream` – Updated bit stream byte pointer. The parameter *`ppBitStream` points to the first available bit stream buffer byte immediately following the packed side information bits. The frame header bits are formatted according to the bit stream syntax given in *ISO/IEC 11172-3:1993*.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. At least one of pointer arguments is NULL, or `mainDataBegin` exceeds the range specified in *ISO/IEC 11172-3:1993*.

BitReservoirInit_MP3

Prototype

```
IppStatus ippsBitReservoirInit_MP3(IppMP3BitReservoir *pDstBitResv,
   IppMP3FrameHeader *pFrameHeader);
```

Description

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.

Input Arguments

`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 `ippsBitReservoirInit_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.

Output Arguments

`pDstBitResv` – Pointer to the initialized `IppMP3BitReservoir` state structure. The structure element `BitsRemaining` is initialized to 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`.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. At least one of the pointer arguments is NULL, or the parameter `pFrameHeader.id` is not equal to 1.

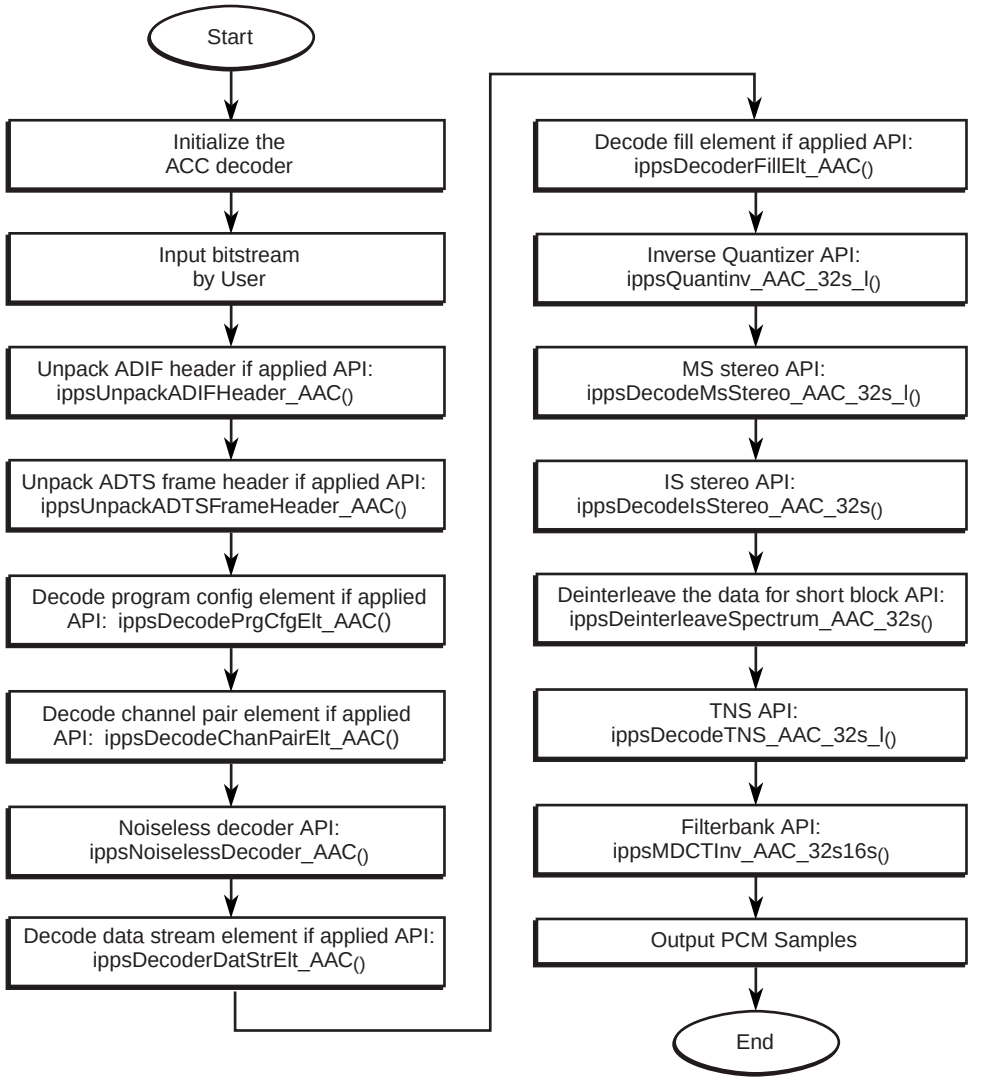
Advanced Audio Coding 10

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 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 will also play a major role for the delivery of high quality music via the Internet. What's more, AAC with some modifications, is the only high-quality audio coding scheme adopted within the MPEG-4 standard, the future "global multimedia language".

This chapter provides information on the AAC decoder for Intel XScale® microarchitecture, including a complete definition of the function calls and data structures that comprise the Application Programming Interface (API). This is part of the Intel® Integrated Performance Primitives (Intel® IPP), which is an efficient and portable library for developing a variety of applications quickly on Intel XScale® microarchitecture.

The AAC decoder API provides a variety of AAC LC decoder functions, including bit stream unpacking and AAC core decoding functions. This provides customers great flexibility in configuring the decoder system. See *ISO/IEC 13818-7:1997* in Appendix B, "Bibliography".

Figure 10-1 AAC Decoder Flowchart



Global Macros

Table 10-1 shows the definitions of global macros.

Table 10-1 Global Macro Definitions

Global Macro Name	Definition	Notes
IPP_AAC_FRAME_LEN	1024	The number of data in one frame
IPP_AAC_SF_LEN	120	scalefactor 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 scalefactor bands number for one frame
IPP_AAC_ELT_NUM	16	maximum number of elements for one program.
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.

Header Files and Libraries

This section describes the definitions and header files of the Intel® IPP MPEG-2 advanced audio coding API.

Header Files

User must include the following files at the beginning of the source code before using any Intel® IPP advanced audio encoder functions:

- ippdefs.h – general header file of Intel® IPP
- ippAC.h – header file of Intel® IPP audio domain

Binary Libraries

The Intel® IPP MP3 advanced audio coding binary library must be referenced when building an application that references any of the Intel® IPP MPEG-2 AAC primitives.

Two different versions are provided in the installation package. One is the debug version, which provides any debug information. The other is the release version, which is used to link in applications. For example, the release version for the Pocket PC* operating system is:

- ippAC_WMMX40APPC_r.lib – Release version for the for the Pocket PC operating system

For specific file names, see the appropriate Release Notes for your operating system.

Data Types and Structures

This section describes the data types and structures of the Intel® IPP advanced audio encoder.

ADIF Header

```
typedef struct {
   Ipp32u    ADIFId;            /* 32-bit, "ADIF" ASCII code */
   int       copyIdPres;        /* copy id flag: 0: off, 1: on */
   int       originalCopy;      /* original bitstream or copy, 0: copy
                                1: original */
   int       home;
   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;               /* original bitstream or copy, 0:
                                copy, 1: original */
    int home;
    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 scalefactor bands */
   int    sfGrouping;              /* scalefactor 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 scalefactor 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; //the 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 two channels
    IppAACLtpInfo pLtpInfo[IPP_AAC_CHAN_NUM];//LTP structure for two 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_PRED_SFB_MAX];
    //FSS information for stereo
    }IppAACExtHeader;
```

TNS Structure for One Layer

```
    typedef struct{
       int tnsDataPresent;
       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;
```

LTP structure

```
typedef struct{
    int ltpDataPresent; //if ltp is used
    int ltpLag;        //the optimal delay from 0 to 2047
    Ipp16s ltpCoef;    //indicate the  LTP coefficient
    int pLtpLongUsed[IPP_AAC_MAX_LTP_SFB]; // 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   pMsUsed[IPP_AAC_SF_LEN]; /* MS stereo flag buffer for each
                                scalefactor band */
} IppAACChanPairElt;
```

Channel Information

```
typedef struct {
    int   tag;
    int   id;                 /* element id */
    int   samplingRateIndex;  /* sampling rate index */
    int   predSfbMax;         /* maximum prediction scalefactor bands */
    int   preWinShape;        /* previous block window shape */

    int   winLen;             /* 128: if short window, 1024: others */
    int   numWin;             /* 1 for long block, 8 for 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]; /* the end of scalefactor offset in each
                                       section */
    int pMaxSect[IPP_AAC_GROUP_NUM_MAX];/* maximum section number for each
                                           group */

    /* TNS data */
    int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX]; /* TNS filter number buffer */
    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;
```

MPEG-2 AAC Primitives

In the following sections, `maxSfb` means number of scalefactor bands transmitted per group. This is unpacked from the bit stream. `numSwb` means number of scalefactor window bands for short block or number of scalefactor window bands for long block. This is calculated according to the sampling rate and the block type.

See clause 8.3.1 of *ISO/IEC 13818-7:1997*.

UnpackADIFHeader_AAC

Prototype

```
IppStatus ippsUnpackADIFHeader_AAC (Ipp8u **ppBitStream,
   IppAACADIFHeader *pADIFHeader, IppAACPrgCfgElt *pPrgCfgElt, int
   prgCfgEltMax)
```

Description

Gets the AAC ADIF format header, including program configuration elements from the input bit stream. See Table 6.2, and 6.21. of *ISO/IEC 13818-7:1997*.

Input Arguments

- `ppBitStream` – double pointer to the current byte before the ADIF header
- `prgCfgEltMax` – the maximum program configure element number

Output Arguments

- `ppBitStream` – double pointer to the current byte after the ADIF header
- `pADIFHeader` – pointer to the `IppACCADIFHeader` structure

- `pPrgCfgElt` – pointer to the `IppAACPrgCfgElt` structure. There must be `prgCfgEltMax` elements in the buffer.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `ppBitStream`, `pADIFHeader`, `pPrgCfgElt` `*ppBitStream` is NULL.
  - `prgCfgEltMax` exceeds [1, 16]
- `ippStsAacPrgNumErr` – the decoded `pADIFHeader->numPrgCfgElt > prgCfgEltMax`.

NOTE. *pADIFHeader->numPrgCfgElt is the number directly unpacked from bit stream plus 1.*

prgCfgEltMax is the number of the program configuration elements that the user wants to support. The valid range is [1, 16]

UnpackADTSFrameHeader_AAC

Prototype

```
IppStatus ippsUnpackADTSFrameHeader_AAC (Ipp8u **ppBitStream,
   IppAACADTSFrameHeader *pADTSFrameHeader)
```

Description

Gets ADTS frame header from the input bit stream. 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.

Input Arguments

`ppBitStream` – double pointer to the current byte

Output Arguments

- `ppBitStream` – double pointer to the current byte after unpacking the ADTS frame header
- `pADTSFrameHeader` – pointer to the `IppAACADTSFrameHeader` structure

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. At least one of the following pointers: `ppBitStream`, `*ppBitStream`, or `pADTSFrameHeader` is NULL.

DecodePrgCfgElt_AAC

Prototype

```
IppStatus ippsDecodePrgCfgElt_AAC (Ipp8u **ppBitStream, int *pOffset,
   IppAACPrgCfgElt *pPrgCfgElt)
```

Description

Gets program configuration element from the input bit stream. See clause 8.5 and Table 6.21 of *ISO/IEC 13818-7*.

Input Arguments

- `ppBitStream` – double pointer to the current byte
- `pOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7. 0: MSB of the byte, 7: LSB of the byte

Output Arguments

- `ppBitStream` – double pointer to the current byte after decoding the program configuration element
- `pOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7. 0: MSB of the byte, 7: LSB of the byte.
- `pPrgCfgElt` – pointer to `IppAACPrgCfgElt` structure

Returns

- `ippStsNoErr` – no error

- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `ppBitStream`, `pOffset`, `pPrgCfgElt`, `*ppBitStream` is NULL.
  - `*pOffset` exceeds [0, 7]

DecodeChanPairElt_AAC

Prototype

```
IppStatus ippsDecodeChanPairElt_AAC (Ipp8u **ppBitStream, int *pOffset,
   IppAACIcsInfo *pIcsInfo, IppAACChanPairElt *pChanPairElt, int
   predSfbMax)
```

Description

Gets `channel_pair_element` from the input bit stream. Individual_channel_stream is not included. If `common_window` flag decoded from the input bit stream is 0, all members of `pIcsInfo` and `pChanPairElt` are not changed except for `pChanPairElt->commonWin`. See clause 8.3 and Table 6.10, 6.11 of *ISO/IEC 13818-7:1997*.

Input Arguments

- `ppBitStream` – double pointer to the current byte
- `pOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7. 0: MSB of the byte, 7: LSB of the byte
- `predSfbMax` – maximum prediction scalefactor bands. For LC profile, set `predSfbMax = 0`

Output Arguments

- `ppBitStream` – double pointer to the current byte after decoding the channel pair element
- `pOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7. 0: MSB of the byte, 7: LSB 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. Some members of the structure must *not* be changed, as shown in Table 10-2.

Table 10-2 Unchanged Members of pIcsInfo

Members	Conditions
`SfGrouping`	`pIcsInfo->winSequence != 2`
`predResetGroupNum`	`pIcsInfo->predDataPres == 0 || pIcsInfo->predReset == 0`
`pPredUsed[sfb]`	`pIcsInfo->predDataPres == 0`

- `pChanPairElt` – pointer to `IppAACChanPairElt` structure. Some members of the structure must *not* be changed, as shown in Table 10-3.

Table 10-3 Unchanged Members of pChanPairElt

Members	Conditions
`pMsUsed[sfb]`	`pChanPairElt->msMaskPres != 1`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `ppBitStream`, `pOffset`, `*ppBitStream`, `pIcsInfo` or `pChanPairElt` is NULL.
  - `*pOffset` exceeds [0, 7]
  - `predSfbMax` < 0
  - `predSfbMax` > 41 (maximum value for all sampling frequency in main profile)
- `ippStsAacMaxsfbErr` – `pIcsInfo->maxSfb` decoded from bit stream greater than 51 (maximum scalefactor band for all sampling frequency)

NoiselessDecoder_LC_AAC

Prototype

```
IppStatus ippsNoiselessDecoder_LC_AAC (Ipp8u **ppBitStream, int
   *pOffset, int commonWin, IppAACChanInfo *pChanInfo, Ipp16s
   *pDstScalefactor, Ipp32s *pDstQuantizedSpectralCoef, Ipp8u
   *pDstSfbCb, Ipp8s *pDstTnsFiltCoef)
```

Description

Decodes all the data for one channel, including scalefactors/intensity positions, spectral coefficients, TNS coefficients and associated side information for LC profile. User must set `pChanInfo->pIcsInfo`, `pChanInfo->samplingRateIndex`, `pChanInfo->predSfbMax` to correct pointer/values before calling this function.

Input Arguments

- `ppBitStream` – double pointer to the current byte
- `pOffset` – pointer to the offset in one byte
- `pChanInfo` – pointer to the channel information `IppAACChanInfo` structure. Members `samplingRateIndex`, `predSfbMax` are treated as input.
- `commonWin` – common window indicator

Output Arguments

- `ppBitStream` – double pointers to bit stream buffer
- `pOffset` – pointer to the offset in one byte
- `pChanInfo` – pointer to the channel information. `IppAACChanInfo` structure. Denote r `pIcsInfo` as `pChanInfo->pIcsInfo` as shown in Table 10-4.

Table 10-4 Input/Output Members List of pChanInfo

Member	Output
Tag	Not used.
id	Not used.
preWinShape	Not used.
pChanPairElt	Not used.

Table 10-4 Input/Output Members List of pChanInfo (continued)

Member	Output
samplingRateIndex	As input. Not changed.
predSfbMax	As input. Must = 0. Not changed.
winLen	As output. Set to 128: if decoded pIcsInfo->winSequence is short block, 1024: others
numWin	As output. Set to 8: if decoded pIcsInfo->winSequence is short block, 1: others
numSwb	As output. Set to the maximum number of scalefactor 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 bit stream.
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 scalefactor 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 which indicates whether the filter is applied in upward or downward direction: 0 for upward and 1 for downward.

Table 10-4 Input/Output Members List of pChanInfo (continued)

Member	Output	
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.	
	Unchanged Members Conditions	
	sfGrouping	pIcsInfo->winSequence != 2
	predResetGroupNum	pIcsInfo->predDataPres == 0 || pIcsInfo->predReset == 0
	pPredUsed[sfb]	pIcsInfo->predDataPres == 0

- `pDstScalefactor` – pointer to the scalefactor or intensity position buffer, buffer length >= 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 scalefactor window bands in each group. Buffer length >= 1024.
- `pDstSfbCb` – pointer to the scalefactor band codebook. Buffer length must >= 120. Store `maxSfb` elements for each group. There are no space between the sequence groups.
- `pDstTnsFiltCoef` – pointer to TNS coefficients. Buffer length must >= 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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `ppBitStream`, `pOffset`, `pChanInfo`, `pDstScalefactor`, `pDstQuantizedSpectralCoef`, `pDstSfbCb`, `pDstTnsFiltCoef` `*ppBitStream` is NULL
  - `pChanInfo->pIcsInfo` is NULL
  - `commonWin` exceeds [0, 1]
  - `*pOffset` exceeds [0, 7]
  - `pChanInfo->samplingRateIndex` exceeds [0, 11]
  - `pChanInfo->predSfbMax` != 0
- `ippStsAacMaxsfbErr` – `pChanInfo->pIcsInfo->maxSfb` decoded from bit stream greater than `pChanInfo->numSwb`

- ippStsAacSfValErr – scalefactor value exceeds normal range error. *pDstScalefactor exceeds [0, 255]
- ippStsAacSectCbErr Codebook pointed by pChanInfo->pSectCb is illegal. *(pChanInfo->pSectCb)==12, 13. If current channel is not the right channel of the channel pair element, *pSectCb = 14, 15 are also illegal, but we do not check these conditions.
- ippStsAacPlsDataErr – The pChanInfo->pIcsInfo->winSequence indicates short sequence and pChanInfo->pulseDataPres indicates pulse data present. Or the start scalefactor band for pulse data >= pChanInfo->numSwb. Or Pulse data position offset >= pChanInfo->winLen.
- ippStsAacGainCtrErr – pChanInfo->gainContrDataPres is decoded as 1 that means gain control data is present. Gain control data is not currently supported.
- ippStsAacCoefValErr – Decoded quantized spectral coefficients value pointed by pDstQuantizedSpectralCoef exceeds [-8191, 8191].
- ippStsAacSectErr – Decoded section number in some group is greater than maxSfb.
- ippStsAacTnsOrderErr – Decoded TNS order is greater than the allowed maximum order of LC profile: 7 for short block, 12 for long block.

DecodeDatStrElt_AAC

Prototype

```
IppStatus ippsDecodeDatStrElt_AAC (Ipp8u **ppBitStream,int *pOffset,
   int *pDataTag, int *pDataCnt, Ipp8u * pDstDataElt)
```

Description

Gets data_stream_element from the input bit stream. See clause 8.6 and Table 6.20 of *ISO/IEC 13818-7:1997*.

Input Arguments

- ppBitStream – double pointer to the current byte
- pOffset – pointer to the bit position in the byte pointed by *ppBitStream. Valid within 0 to 7. 0: MSB of the byte, 7: LSB of the byte.

Output Arguments

- ppBitStream – double pointer to the current byte after the decode data stream element

- pOffset – pointer to the bit position in the byte pointed by *ppBitStream. Valid within 0 to 7. 0: MSB of the byte, 7: LSB 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 length of total data in bytes
- pDstDataElt – pointer to the data stream buffer that contains the data stream extracted from the input bit stream. There are 512 elements in the buffer pointed by pDstDataElt.

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - At least one of the following pointers: ppBitStream, pOffset, *ppBitStream, pDataTag, pDataCnt or pDstDataElt is NULL.
  - *pOffset exceeds [0, 7]

DecodeFillElt_AAC

Prototype

```
IppStatus ippsDecodeFillElt_AAC (Ipp8u **ppBitStream, int *pOffset,
   int *pFillCnt, Ipp8u * pDstFillElt)
```

Description

Gets the fill element from the input bit stream. See clause 8.7 and Table 6.22 of *ISO/IEC 13818-7:1997*.

Input Arguments

- ppBitStream – pointer to the pointer to the current byte
- pOffset – pointer to the bit position in the byte pointed by *ppBitStream. Valid within 0 to 7. 0: MSB of the byte, 7: LSB of the byte.

Output Arguments

- ppBitStream – pointer to the pointer to the current byte after the decode fill element
- pOffset – pointer to the bit position in the byte pointed by *ppBitStream. Valid within 0 to 7. 0: MSB of the byte, 7: LSB 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

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `ppBitStream`, `pOffset`, `*ppBitStream`, `pFillCnt` or `pDstFillElt` is NULL.
  - `*pOffset` exceeds [0, 7]

QuantInv_AAC_32s_I

Prototype

```
IppStatus ippsQuantInv_AAC_32s_I (Ipp32s *pSrcDstSpectralCoef, const
   Ipp16s *pScalefactor, int numWinGrp, const int *pWinGrpLen, int
   maxSfb, const Ipp8u *pSfbCb, int samplingRateIndex, int winLen)
```

Description

Inverse quantize the Huffman symbols for current channel. The formula is shown as below equation. See clause 10 of *ISO/IEC 13818-7:1997*.

$$pSrcDst[i] = sign(pSrcDst[i]) * (pSrcDst[i])^{\frac{4}{3}} * 2^{\left(\frac{1}{4}(pScalefactor[sfb] - 100)\right)}$$

Input Arguments

- `pSrcDstSpectralCoef` – pointer to the input quantized coefficients. For short block the coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024.
- `pScalefactor` – pointer to the scalefactor buffer. Buffer length must >= 120.
- `numWinGrp` – group number
- `pWinGrpLen` – pointer to the number of windows in each group. Buffer length must >= 8.

- maxSfb – max scalefactor bands number for the current block
- pSfbCb – pointer to the scalefactor band codebook, buffer length must >= 120. Only maxSfb elements for each group are meaningful. There are no spaces between the sequence groups.
- samplingRateIndex – sampling rate index. Valid in [0, 11]. See Table 6.5 of *ISO/IEC 13818-7:1997*.
- winLen – the data number in one window

Output Arguments

- pSrcDstSpectralCoef – pointer to the destination inverse quantized coefficient in Q13.18 format . For short block, the coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024. The maximum error of output pSrcDstSpectralCoef [i] is listed in Table 10-5.

Table 10-5 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

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - At least one of the following pointers: pSrcDstSpectralCoef, pScalefactor, pWinGrpLen or pSfbCb is NULL.
  - If short block numWinGrp exceeds [1, 8]
  - If long block, numWinGrp != 1
  - maxSfb exceed [0, 51]
  - samplingRateIndex exceeds [0, 11]
  - winLen is neither 1024 nor 128
- ippStsAacCoefValErr – input coefficients value pointed by pSrcDstSpectralCoef exceeds[-8191, 8191]
- ippStsAacMaxsfbErr – the calculated scalefactor band index exceeds the numSwb in each window

DecodeMsStereo_AAC_32s_I

Prototype

```
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)
```

Description

MS stereo process for pair channels. This also performs the `invert_intensity(group, sfb)` function and stores the values in the `pSfbCb` buffer. If `invert_intensity(group, sfb)` = -1, and if `*pSfbCb` = `INTERITY_HCB`, let `*pSfbCb` = `INTERITY_HCB2`; else if `*pSfbCb` = `INTERITY_HCB2`, let `*pSfbCb` = `INTERITY_HCB`. Only when MS stereo flag is on, perform operation on `pSrcDstL[i]` and `pSrcDstR[i]` as described in the below formula below. See clause 12 of *ISO/IEC 13818-7:1997*.

$$pSrcDstL'[i] = pSrcDstL[i] + pSrcDstR[i],$$
$$pSrcDstR'[i] = pSrcDstL[i] - pSrcDstR[i].$$

Input Arguments

- `pSrcDstL` – pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024. `pSrcDstL` must be 8-byte aligned.
- `pSrcDstR` – pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024. `pSrcDstR` must be 8-byte aligned.
- `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 >= 120.
- `pSfbCbPointer` to the scalefactor band codebook, buffer length must >= 120. Store `maxSfb` elements for each group. There are no space between the sequence groups.
- `numWinGrp` – group number
- `pWinGrpLen` – pointer to the number of windows in each group. Buffer length must >= 8
- `maxSfb` – max 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 – the data number in one window

Output Arguments

- pSrcDstL – pointer to left channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024. pSrcDstL must be 8-byte aligned.
- pSrcDstR – pointer to right channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024. pSrcDstR must be 8-byte aligned.
- pSfbCb– pointer to the scalefactor band codebook. If invert_intensity group, sfb) = -1, and if *pSfbCb = INTERITY_HCB, let *pSfbCb = INTERITY_HCB2; else if *pSfbCb = INTERITY_HCB2, let *pSfbCb = INTERITY_HCB. Buffer length must >= 120. Store maxSfb elements for each group. There is no space between the sequence groups.

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - At least one of the following pointers: pSrcDstL, pSrcDstR, pMsUsed, pWinGrpLen, pSfbCb is NULL.
  - pSrcDstL or pSrcDstR is not 8-byte aligned
  - For short blocks, numWinGrp exceeds [1, 8]
  - For long blocks, numWinGrp != 1
  - maxSfb exceeds [0, 51]
  - msMaskPres exceeds [1, 2]
  - samplingRateIndex exceeds [0, 11]
  - winLen is neither 1024 nor 128
- ippStsAacMaxsfbErr – the calculated scalefactor band index exceeds the numSwb in each window

DecodeIsStereo_AAC_32s

Prototype

```
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)
```

Description

Intensity stereo process for pair channels. Only the `pSfbCb[sfb]` indicates intensity stereo on for that scalefactor band. Perform operation on `pSrcL[i]`, `pDstR[i]` as described in the formula below. `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*.

$$pDstR[i] = pSrcL[i] * is\_intensity(g, sfb) * 2^{\left(-\frac{1}{4} pScalefactor[sfb]\right)}$$

Input Arguments

- `pSrcL` – pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024. `pSrcL` must be 8-byte aligned.
- `pScalefactor` – pointer to the scalefactor buffer. Buffer length must >= 120.
- `pSfbCb` – pointer to the scalefactor band codebook, buffer length must >= 120. Store maxSfb elements for each group.There are no space between the sequence groups.
- `numWinGrp` – group number
- `pWinGrpLen` – pointer to the number of windows in each group. Buffer length must >= 8.
- `maxSfbMax` – 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` – the data number in one window

Output Arguments

- `pDstR` – pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024. `pDstR` must be 8-byte aligned.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `pSrcL`, `pDstR`, `pWinGrpLen`, `pScalefactor`, `pSfbCb` is NULL. If `pSrcL`, `pDstR` is not 8-byte aligned.
  - If short block, `numWinGrp` exceeds [1, 8]
  - If long block, `numWinGrp` != 1
  - `maxSfb` exceeds [0, 51]
  - `samplingRateIndex` exceeds [0, 11]
  - `winLen` is neither 1024 nor 128
- `ippStsAacMaxsfbErr` – the calculated scalefactor band index exceeds the `numSwb` in each window

DeinterleaveSpectrum_AAC_32s

Prototype

```
IppStatus ippsDeinterleaveSpectrum_AAC_32s (const Ipp32s *pSrc, Ipp32s
   *pDst, int numWinGrp, const int *pWinGrpLen, int maxSfb, int
   samplingRateIndex, int winLen)
```

Description

Deinterleaves the coefficients for short block. See clause 8.3.5 of *ISO/IEC 13818-7:1997*.

Input Arguments

- `pSrc` – pointer to source coefficients buffer. The coefficients are interleaved by scalefactor window bands in each group. Buffer length must >= 1024. `pSrc` must be 8-byte aligned.
- `numWinGrp` – group number
- `pWinGrpLen` – pointer to the number of windows in each group. Buffer length must >= 8

- `maxSfbMax` – 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` – the data number in one window

Output Arguments

- `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 scalefactor band index, `sfbWidth` is the scalefactor band width table, `i` is the index within scalefactor band. Buffer length must >= 1024. `pDst` must be 8-byte aligned.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `pSrc`, `pDst`,`pWinGrpLen` is NULL.
  - Either `pSrc` or `pDst` are not 8-byte aligned
  - `numWinGrp` exceeds [1, 8]
  - `maxSfb` exceeds [0, 51]
  - `samplingRateIndex` exceeds [0, 11]
  - `winLen` is not 128
- `ippStsAacMaxsfbErr` – the calculated scalefactor band index exceeds the numswb in each window

DecodeTNS_AAC_32s_I

Prototype

```
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)
```

Description

Decoding process for Temporal Noise Shaping that controls the temporal shape of the quantization noise within each window of the transform. The decoding process for Temporal Noise Shaping is done separately on each window of the current frame by applying the all-pole filtering to selected regions of the spectral coefficients. This function supports LC profile only.

Input Arguments

- `pSrcDstSpectralCoefs` – 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 by `pSrcDstSpectralCoefs`.
- `pTnsNumFilt` – pointer to the number of noise shaping filters that are used for each window of the current frame. There are 8 elements in the buffer pointed 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 scalefactor bands) to which one filter is applied in each window of the current frame. There are 8 elements in the buffer pointed 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 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 (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 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 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 order 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 they exist. 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 they exist, and so on. order0 is the order 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 numWin windows are all processed.
- pTnsDirection – pointer to the token which indicates whether the filter is applied in upward or downward direction: 0 for upward and 1 for downward. There are 8 elements in the buffer pointed 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

  $$w, i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt, w=0 \text{ to } numWin-1, filt=0$$

  to pTnsNumFilt[w]-1.
- maxSfb – the number of scalefactor bands transmitted per window group of the current frame
- profile – the profile index from Table 7.1 in *ISO/IEC 13818-7:1997*
- samplingRateIndex – the index which indicates the sampling rate of the current frame
- winLen – the data number in one window

Output Arguments

pSrcDstSpectralCoefs – pointer to the output spectral coefficients after filtering by the all-pole filters in Q13.18 format. See Table 10-6 for the computation error compared with the double precision data.

Table 10-6 Computation Error List for pSrcDstSpectralCoefs

MAX(error(pSrcDstSpectralCoefs[i]))	Conditions
4095	`8 == numWin`
32767	`1 == numWin`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `pSrcDstSpectralCoefs`, `pTnsNumFilt`, `pTnsRegionLen`, `pTnsFiltOrder`, `pTnsFiltCoefRes`, `pTnsFiltCoef`, or `pTnsDirection` is NULL.
  - `maxSfb` < 0 or `maxSfb` > `numSwb`
  - profile!=1
  - `samplingRateIndex` exceeds [0, 11]
  - `winLen`!=128 and `winLen`!=1024
- `ippStsAacTnsNumFiltErr` – for a short window sequence, `pTnsNumFilt[w]` exceeds [0, 1]; For long window sequence, `pTnsNumFilt[w]` exceeds [0, 3], `w`=0 to `numWin`-1.
- `ippStsAacTnsLenErr` – `*pTnsRegionLen` exceeds [0, `numSwb`]
- `ippStsAacTnsOrderErr` – for short window sequence, `*pTnsFiltOrder` exceeds [0, 7]; For long window sequence, `*pTnsFiltOrder` exceeds [0, 12]
- `ippStsAacTnsCoefResErr` – `pTnsFiltCoefRes[w]` exceeds [3, 4], `w`=0 to `numWin`-1
- `ippStsAacTnsCoefErr` – `*pTnsFiltCoef` exceeds [-8, 7]
- `ippStsAacTnsDirectErr` – `*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 it is 1 for other window sequences. numSwb is the total number of scalefactor window bands for the actual window type (long or short window) of the current frame.*

MDCTInv_AAC_32s16s

Prototype

```
IppStatus ippsMDCTInv_AAC_32s16s (Ipp32s *pSrcSpectralCoefs, Ipp16s
   *pDstPcmAudioOut, Ipp32s *pSrcDstOverlapAddBuf, int winSequence, int
   winShape, int prevWinShape, int pcmMode)
```

This module is used to map the time-frequency domain signal into time domain and generate 1024 reconstructed 16-bit signed little-endian PCM samples as output for each channel. This module consists of an IMDCT transform, and a windowing and an 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 2, 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, output is in the sequence `pDstPcmAudioOut[i], i=0` to 1023. User must also preallocate an input-output buffer pointed by `pSrcDstOverlapAddBuf` for overlap-add operation. Reset this buffer to zero before first call, then use the output of the current call as the input of the next call for the same channel.

Input Arguments

- `pSrcSpectralCoefs` – pointer to the input time-frequency domain samples in Q13.18 format. There are 1024 elements in the buffer pointed by `pSrcSpectralCoefs`.
- `pSrcDstOverlapAddBuf` – pointer to the overlap-add buffer which 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 (LRLRLR…) or not.
  1 = not interleaved;2 = interleaved

Output Arguments

`pDstPcmAudioOut` – Pointer to the output 1024 reconstructed 16-bit signed little-endian PCM samples in Q15, interleaved if needed. See Table 10-7 for `pDstPcmAudioOut` computation errors.

Table 10-7 Computation Error List for pDstPcmAudioOut

MAX(error(pDstPcmAudioOut[i]))	$\sum_{i=0}^{1023} error^2(pDstPcmAudioOut[i])$
1	96

- `pSrcDstOverlapAddBuf` – pointer to the overlap-add buffer which contains the second half of the current block windowed sequence in Q13.18. The computation error compared with double precision is listed in Table 10-8:

Table 10-8 Computation Error List for pSrcDstOverlapAdd

MAX(error(pSrcDstOverlapAddBuf[i]))	$\sum_{i=0}^{1023} error^2(pSrcDstOverlapAddBuf[i])$
4	1536

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `pSrcSpectralCoefs`, `pSrcDstOverlapAddBuf` and `pDstPcmAudioOut` is NULL;
  - `winSequence` < 0, or `winSequence` > 3
  - `winShape` < 0, or `winShape` > 1
  - `prevWinShape` < 0, or `prevWinShape` >
  - `pcmMode` < 1, or `pcmMode` > 2

MPEG-4 AAC Primitives

DecodeMainHeader_AAC

Prototype

```
IppStatus ippsDecodeMainHeader_AAC(Ipp8u **ppBitStream, int *pOffset,
   IppAACMainHeader *pAACMainHeader, int channelNum, int monoStereoFlag)
```

Description

Gets main header information and main layer information from bit stream.

Input Arguments

- `ppBitStream` – double pointers to bit stream buffer
- `pOffset` – pointer to the offset in one byte
- `ChannelNum` – number of channels
- `monoStereoFlag` – current frame has mono and stereo layers

Output Arguments

- `ppBitStream` – double pointers to bit stream buffer after decode main element
- `pOffset` – pointer to the offset in one byte after decode main element
- `pAACMainHeader` – pointer to the main element header, include window sequence, window shape, max ssfb, scalefactor grouping, MS, TNS, ans LTP information

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `ppBitStream`, `ppBitStream`, `pAACMainHeader`, `*ppBitStream`, or `pOffset` is NULL
  - `*pOffset < 0`, or `*pOffset > 7`
  - `channelNum` exceeds [1,2]
  - `monoStereoFlag` exceeds [0,1]

DecodeExtensionHeader_AAC

Prototype

```
IppStatus ippsDecodeExtensionHeader_AAC(Ipp8u **ppBitStream, int
   *pOffset, IppAACExtHeader *pAACExtHeader, int monoStereoFlag, int
   thisLayerStereo, int monoLayerFlag, int preStereoMaxSfb, int
   hightstMonoMaxSfb, int winSequence)
```

Description

Get extension header information and exension layer information from bit stream.

Input Arguments

- `ppBitStream` – double pointers to bit stream buffer
- `pOffset` – pointer to the offset in one byte
- `monoStereoFlag` – current frame has mono and stereo layers
- `thisLayerStereo` – current layer is stereo
- `monoLayerFlag` – current frame has mono layer
- `preStereoMaxSfb` – previous stereo layer's `maxSfb`
- `hightstMonoMaxSfb` – last mono layer's `maxSfb`
- `winSequence` – window type, short or long

Output Arguments

- `ppBitStream` – double pointers to bit stream buffer
- `pOffset` – pointer to the offset in one byte
- `pAACExtHeader` – pointer to the extension element header, include max sfb, ms,tns, FSS control information

Returns

- ippStsNoErr – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `ppBitStream`, `*ppBitStream`, `pAACExtHeader`, or `pOffset` is NULL.
  - `*pOffset` < 0 or `*pOffset` > 7

— monnoStereoFlag exceeds [0,1]
— this LayerStereo exceeds [0,1]
— monoLayerFlag exceeds [0,1]
— preStereoMaxSfb exceeds [0, 51]
— hightstMonoMaxSfb exceeds [0,51]
— winSequence exceeds [0,3]

DecodePNS_AAC_32s

Prototype

```
IppStatus ippsDecodePNS_AAC_32s(Ipp32s *pSrcDstSpec, int
   *pSrcDstLtpFlag, Ipp8u *pSfbCb, Ipp16s *pScaleFactor, int maxSfb, int
   numWinGrp, int *pWinGrpLen, int samplingFreqIndex, int winLen, int
   *pRandomSeed)
```

Description

Implements perceptual noise substitution coding within an ICS. Certain sets of spectral coefficients are derived from random vectors rather than from Huffman-coded symbols and an inverse quantization process.

Input Arguments

- pSrcDstSpec – pointer to spectrum coefficients to be PNS
- pSrcDstLtpFlag – pointer to LTP used flag
- pSfbCb – pointer to scale factor code book
- pScaleFactor – pointer to the scale factor value
- maxSfb – number of scale factor band used in this layer
- numWinGrp – number of window group
- pWinGrpLen – pointer to the length of every window group
- samplingFreqIndex – sampling frequency index
- winLen – window length, 1024 for long, 128 for short
- pRandomSeed – random seed for PNS

Output Arguments

- `pSrcDstSpec` – pointer to the output spectrum substituted by perceptual noise
- `pSrcDstLtpFlag` – pointer to the LTP used flag
- `pRandomSeed` – random seed for PNS

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `pSrcDstSpec`, `pSfbCb`, `pScaleFactor`, `pWinGrpLen` or `pSrcDstLtpFlag` is NULL
  - `maxSfb` exceeds [0,51]
  - `numWinGrp` exceeds [1, 8]
  - `samplingFreqIndex` exceeds [0,12]
  - `winLen` is neither 128 nor 1024

LongTermReconstruct_AAC_32s

Prototype

```
IppStatus ippsLongTermReconstruct_AAC_32s(Ipp32s *pSrcEstSpec, Ipp32s
   *pSrcDstSpec, int *pLtpFlag, int winSequence, int samplingFreqIndex)
```

Description

Use Long Term Reconstruct (LTP) to reduce the redundancy of a signal between successive coding frames. LTP is a forward adaptive predictor, which is inherently less sensitive to round-off numerical errors in the decoder or bi errors in the transmitted spectral coefficients. Add the vector of decoded spectral coefficients and the corresponding frequency domain vector to get the vector of reconstructed spectral coefficients.

Input Arguments

- `pSrcDstSpec` – pointer to spectral coefficients to do long term prediction
- `pSrcEstSpec` – pointer to the frequency domain vector
- `winSequence` – window type (long or short)
- `samplingFreqIndex` – sampling frequency index

- `pLtpFlag` – pointer to the LTP used flag

Output Arguments

`pSrcDstSpec` – pointer to spectral coefficients have been done long term prediction

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `pSrcDstSpec`, `pSrcEstSpec` and `pLtpFlag` is NULL
  - `winSequence` exceeds [0,3]
  - `samplingFreqIndex` exceeds [0,12]

MDCTFwd_AAC_32s

Prototype

```
IppStatus ippsMDCTFwd_AAC_32s(Ipp32s *pSrc, Ipp32s *pDst, Ipp32s
   *pSrcDstOverlapAdd, int winSequence, int winShape, int preWinShape,
   Ipp32s *pWindowedBuf)
```

Description

In the Long Term Reconstruct (LTP) loop, MDCT is needed to generate spectrum coefficient of PCM samples.

Input Arguments

- `pSrc` – pointer to temporal signals to do MDCT
- `pSrcDstOverlapAdd` – pointer to overlap buffer. Not used in MPEG-4 AAC decode
- `winSequence` – window sequence shows that this block is long or short block
- `winShape` – window shape shows current window's shape
- `preWinShape` – window shape shows previous window's shape
- `pWindowedBuf` – work buffer for MDCT, length of pWindowedBuf is at least 2048 words

Output Arguments

- `pSrcDstOverlapAdd` – pointer to overlap buffer. Not used in MPEG-4 AAC decode.

- `pDst` – output of MDCT, the spectral coefficients of PCM samples

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `pSrc`, `pDst`, `pWindowedBuf` or `pSrcDstOverlapAdd` is NULL.
  - `winSequence` exceeds [0,3]
  - `winShape` exceeds [0,1]
  - `preWinShape` exceeds [0,1]

EncodeTNS_AAC_32s_I

Prototype

```
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 samplingFreqIndex, int
   winLen)
```

Description

In the Long Term Reconstruct (LTP) loop, Analysis Temporal Noise Shaping is needed for reversion of TNS.

Input Arguments

- `pSrcDst` – pointer to the spectral coefficients to do encode TNS
- `pTnsNumFilt` – pointer to number of TNS filter
- `pTnsRegionLen` – pointer to length of TNS filter
- `pTnsFiltOrder` – pointer to TNS filter order
- `pTnsFiltCoefRes` – pointer to TNS coef resolution flag
- `pTnsFiltCoef` – pointer to TNS filter coefficients
- `pTnsDirection` – pointer to TNS direction flag
- `maxSfb` – maximum scale factor number

- `profile` – audio profile
- `samplingFreqIndex` – sampling frequency index
- `winLen` – window length

Output Arguments

Pointer to the spectral coefficients have done encode TNS.

Returns

- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `pSrcDst`, `pTnsNumFilt`, `pTnsRegionLen`, `pTnsFiltOrder`, `pTnsFiltCoefRes`, `pTnsFiltCoef` or `pTnsDirection` is NULL
  - `maxSfb` exceeds [0,51]; `winLen` is neither 128 nor 1024; `samplingRateIndex` exceeds [0,12]

LongTermPredict_AAC_32s

Prototype

```
IppStatus ippsLongTermPredict_AAC_32s(Ipp32s *pSrcTimeSignal, Ipp32s
   *pDstEstTimeSignal, IppAACLtpInfo *pAACLtpInfo, int winSequence)
```

Description

In the Long Term Reconstruct (LTP) loop, Analysis LTP is needed to get the predicted time domain signals.

Input Arguments

- `pSrcTimeSignal` – pointer to the temporal signals to be predicted in temporary domain
- `pDstEstTimeSignal` – pointer to the output of samples after LTP
- `pAACLtpInfo` – pointer to the LTP information
- `winSequence` – window type (short or long)

Output Arguments

`pDstEstTimeSignal` – pointer to the output of prediction in time domain

Returns

`ippStsBadArgErr` – bad arguments

- At least one of the pointers: `pSrcDstTime`, or `pAACMainHeader` is NULL
- `winSequence` exceeds [0,3]

NoiseLessDecode_AAC

Prototype

```
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
```

Description

This is a general noiseless decode module for MPEG-2 and MPEG-4 objects. In the AAC scaleable object of MPEG-4, if PNS is used in one scale factor band, the `pDstSfbCb[sfb]` should be NOISE_HCB(13) and the `*pDstScaleFactor` contains the noise energy of this scale factor band. The spectrum in this scale factor band does not have to be Huffman decoded, and the `pDstQuantizedSpectralCoef` of this scale factor band can set to be zero.

In AAC scaleable object, `pDstTnsFiltCoef` and `pAACMainHeader` are not used.

Input Arguments

- `ppBitStream` – double pointer to the bit stream to be parsed
- `pOffset` – pointer to the offset in one byte
- `pAACMainHeader` – pointer to main header information. Not used in scaleable object. When `commonWin==0 && scaleFlag==0`, need to decode LTP information and save in `pAACMainHeader->pLtpInfo[]`.
- `pChanInfo` – pointer to channel information structure
- `windowSequence` – window type, short or long
- `maxSfb` – number of scale factor band
- `commonWin` – if channel pair use the same ics information
- `scaleFlag` – if scaleable is used

- audioObjectType – audio object type indication. 1:main, 2:LC, 6:scaleable

Output Arguments

- ppBitStream – double pointer to the bit stream has been parsed
- pOffset – pointer to the offset in one byte
- pChanInfo – pointer to channel information structure
- pDstScaleFactor – pointer to the scale factor has been parsed
- pDstQuantizedSpectralCoef – pointer to the quantized spectral coefficients after Huffman decoder
- pDstSfbCb – pointer to the scale factor code book index
- pDstTnsFiltCoef – pointer to TNS filter coefficients. Not used in scaleable object.

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - At least one of the pointers: ppBitStream, pOffset, *ppBitStream, pAACMainHeader, pDstScaleFactor, pDstTnsFiltCoef, pDstQuantizedSpectralCoef, pChanInfo or pDstSfbCb is NULL
  - *pOffset exceeds [0,7]
  - winSequence exceeds [0,3]; maxSfb exceeds [0,51]
  - commonWin exceeds [0,1]
  - scaleFlag exceeds [0,1]
  - audioObjectType exceeds [0,16]

LtpUpdate_AAC_32s

Prototype

```
IppStatus ippsLtpUpdate_AAC_32s (Ipp32s *pSpecVal, Ipp32s *pLtpSaveBuf,
   int winSequence, int winShape, int preWinShape, Ipp32s *pWorkBuf)
```

Description

In the Long Term Reconstruct (LTP) loop, buffer update is required. This includes IMDCT and updating the save buffer.

Input Arguments

- `pSpecVal` – pointer to spectral value after TNS decoder in LTP loop
- `pLtpSaveBuf` – pointer to save buffer for LTP. Buffer length should be 3*frameLength
- `winSequence` – window type, 0-long, 1-long start, 2-short, 3-long stop
- `winShape` – window shape, KBD or SIN window
- `preWinShape` – window shape of previous window
- `pWorkBuf` – work buffer for Ltp update, length of pWorkBuf is at least 2048*3 = 6144 Words

Output Arguments

`pLtpSaveBuf` – pointer to save buffer for LTP. Buffer length should be 3*frameLength. The value is saved for next frame.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `pLtpSaveBuf`, `pWorkBuf` or `pSpecVal` is NULL
  - `winSequence` exceeds [0,3]
  - `winShape` exceeds [0,1]
  - `preWinShape` exceeds [0,1]

QuantInv_AAC_32s_I

Prototype

```
IppStatus ippsQuantInv_AAC_32s_I (Ipp32s *pSrcDstSpectralCoef, const
   Ipp16s *pScalefactor, int numWinGrp, const int *pWinGrpLen, int
   maxSfb, const Ipp8u *pSfbCb, int samplingFreqIndex, int winLen)
```

Description

For those scale factor band, *pSfbCb is NOISE_HCB(13), inverse quantization can be omitted.

Input Arguments

- `pSrcDstSpectralCoef` – pointer to the Spectral coefficients before inverse quantization

- pScalefactor – pointer to the scale factor
- numWinGrp – number of window group
- pWinGrpLen – pointer to the length of every window group
- maxSfb – number of scale factor band
- pSfbCb – pointer to the scale factor code book
- samplingFreqIndex – sampling frequency index
- winLen – length of window

Output Arguments

pSrcDstSpectralCoef – pointer to the Spectral coefficients after inverse quantization

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad Arguments. At least one of the pointers: pSrcDstSpectralCoef, pScalefactor, pWinGrpLen and pSfbCb, is NULL

DecodeMsStereo_AAC_32s_I

Prototype

```
IppStatus ippsDecodeMsStereo_AAC_32s_I (Ipp32s *pSrcDstL, Ipp32s
   *pSrcDstR, int msMaskPres, const Ipp8u (*ppMsUsed)[IPP_AAC_SF_MAX],
Ipp8u *pSfbCb, int numWinGrp, const int *pWinGrpLen, int maxSfb, int
   samplingRateIndex, int winLen)
```

Description

This function is comparable to "DecodeMsStereo_AAC_32s_I", but the "MPEG-2 AAC Primitives" format of pMsUsed is different for MPEG-4 AAC scaleable objects. In this case, information of pMsUsed is buffered into a two dimension array – ppMsUsed[IPP_AAC_WIN_MAX][IPP_AAC_SF_MAX].

Input Arguments

- pSrcDstL – pointer to left channel signal
- pSrcDstR – pointer to right channel signal

- `msMaskPres` – MS used indication, 0:non;1:some;2:all
- `ppMsUsed` – pointer to pointer to ms used information
- `pSfbCb` – pointer to scale factor band's code book
- `numWinGrp` – number of window group
- `pWinGrpLen` – pointer to length of window group
- `maxSfb` – maximum scale factor band
- `samplingRateIndex` – sampling frequency index
- `winLen` – window length

Output Arguments

- `pSrcDstL` – pointer to left channel signal after MS decode
- `pSrcDstR` – pointer to right channel signal after MS decode

Returns

- `IppStsNoErr` – no error
- `IppStsBadArgErr` – bad arguments
  - At least one of the following pointers:`PSrcDstL`, `pSrcDstR`, `ppMsUsed`, `pWinGrpLen` or `pSfbCb` is NULL
  - If short block, `numWinGrp` exceeds [1,8]
  - If long block, `numWinGrp` !=1
  - `maxSfb` exceed [0,51]
  - `MsMaskPres` exceeds [1,2]
  - `SamplingRateIndex` exceeds [0,11]
  - `WinLen` is neither 1024 nor 128
- `IppStsAacMaxsfbErr` – the calculated `scaleFactor` band index exceeds the maximum `swb` in each window

Decode Channel Pair Element

DecodeChanPairElt_MPEG4_AAC

Prototype

```
IppStatus ippsDecodeChanPairElt_MPEG4_AAC (Ipp8u **ppBitStream, int
   *pOffset, IppAACIcsInfo *pIcsInfo, IppAACChanPairElt *pChanPairElt,
   IppAACMainHeader *pAACMainHeader, int predSfbMax, int
   audioObjectType)
```

Description

Retrieves the channel_pair_element from the input bit stream. Individual_channel_stream is not included here. If common_window flag that is decoded from the input bit stream is 0, all members of pIcsInfo and pChanPairElt are not changed except for pChanPairElt->commonWin.

Input Arguments

- `ppBitStream` – double pointer to the current byte.
- `pOffset` – pointer to the bit position in the byte pointed by *ppBitStream. Valid within 0 to 7. 0: MSB of the byte, 7: LSB of the byte.
- `pChanPairElt` – pointer to channel pair element.
- `pAACMainHeader` – pointer to Main layer header structure.
- `predSfbMax` – maximum prediction scalefactor bands. For LC profile, set predSfbMax = 0 for there is no predictors.
- `audioObjectType` – index of audio object type. 2: LC, 4: LTP

Output Arguments

- `ppBitStream` – double pointer to the current byte, after decoding the channel pair element.
- `pOffset` – pointer to the bit position in the byte pointed by *ppBitStream. Valid within 0 to 7. 0: MSB of the byte, 7: LSB of the byte.
- `pIcsInfo` – pointer to IppAACIcsInfo structure.

Return

- `ippStsNoErr` – No Error.

- `ippStsBadArgErr` – Bad arguments. Either `ppBitStream`, `pOffset`, `*ppBitStream`, `pIcsInfo`, or `pChanPairElt` is NULL. `*pOffset` exceeds [0, 7]. `predSfbMax` < 0 or `predSfbMax` > 41 (maximum value for all sampling frequency in main profile).
- `ippStsAacMaxsfbErr` – pIcsInfo->maxSfb decoded from bitstream greater than 51(maximum scalefactor band for all sampling frequency).

H.263 Video Decoder and Processing

11

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) that support the general video processing and ITU-T Recommendation H.263 and Annexes - decoder part, which most often is denoted by the “H263+ decoder” acronym.

NOTE. *The general video processing primitives do not have the H.263 as “modifier” in the function name; all H.263-oriented primitives have H.263 as part (modifier) of the function name.*

To benefit application developers, the design philosophy of the API, like all other primitives, emphasizes maximum flexibility and performance. On the one hand, developers have the option of building a complete H.263+ decoder solution using the compact set of performance optimized H.263+ primitives described in this chapter, in conjunction with administrative and memory management functions customized for the application environment. In this scenario, developers are able to leverage the fact that the H.263+ primitives have been tuned carefully for minimum cycle count, minimum memory footprint, and maximum quality. On the other hand, developers also have the option of building a custom H.263+ decoder while electing to use only a subset of Intel® IPP H.263+ primitives. This development option is facilitated in the API, by providing access to the intermediate computational results generated by each of the H.263+ routines. Moreover, the primitive API grants user access to all internal data objects. Finally, the API allows the user to fully exploit performance properties of a particular target operating system (OS), by allowing user management of administrative functions, such as the high-level bit stream manipulation, memory allocation/deallocation, and control of the various buffers. The primitives cover the following aspect of the H.263+ Decoder:

- Motion vector decoding /block-based Motion compensation
- Inverse quantization, inverse zigzag positioning, reconstruction and IDCT; “compact” and “plane” versions.

- Block layer coefficient decoding, includes bit stream parsing, VLC decoding, inverse quantization, inverse zigzag positioning and IDCT, with appropriate clipping on each step.
- Unrestricted Motion Vectors Mode (Annex D in H.263+)
- Advanced Prediction Mode (Annex F in H.263+)
- Deblocking Filter Mode (Annex J in H.263+)

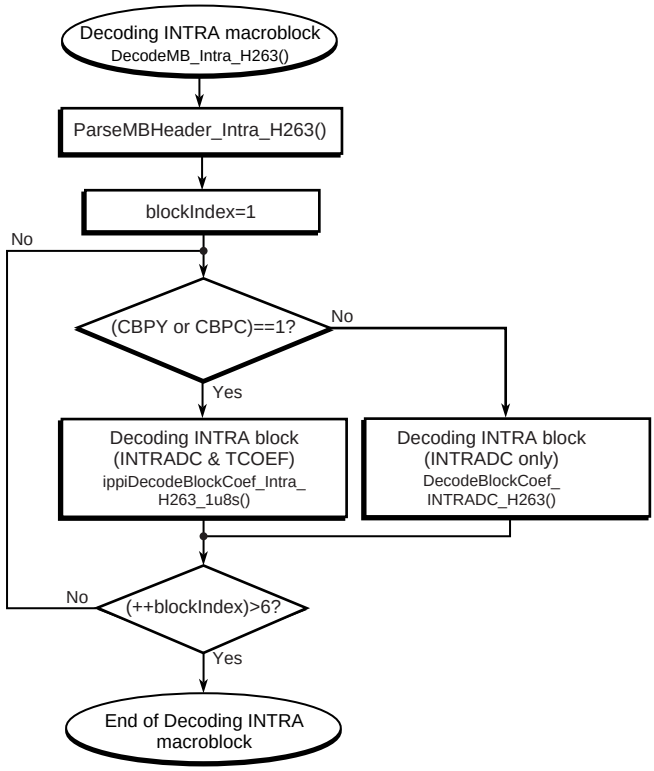
The rest of this chapter provides details on the H.263+ API, and is organized as follows. First, section "High-Level Description" gives a high level description of on the H.263+ primitives. The signal flowcharts are provided to show the data flow. Next, sections "Structure and Macro Definitions" through "H.263+ Middle-Level Primitives" focus on individual primitive and macro/data structures used in the primitives.

High-Level Description

Decoding the INTRA Macroblock

Figure 11-1 shows the process of decoding the INTRA macroblock.

Figure 11-1 Process of Decoding the INTRA Macroblock



A9287-01

NOTE. Cbpc -> mcbpc
`cbpc`: This variable length code represents a pattern of non-transparent luminance blocks with at least one non intra DC transform coefficient in a macroblock.
`mcbpc`: This is a variable length code that is used to derive the macroblock type and the coded block pattern for chrominance.

Figure 11-2 shows how the primary segment can be implemented through the low-level IPP functions. `ippiDecodeBlockCoef_Intra_H263_1u8u()` is a middle-level function.

Figure 11-2 The Process of Decoding INTRA Block Using IPP Low Level Functions

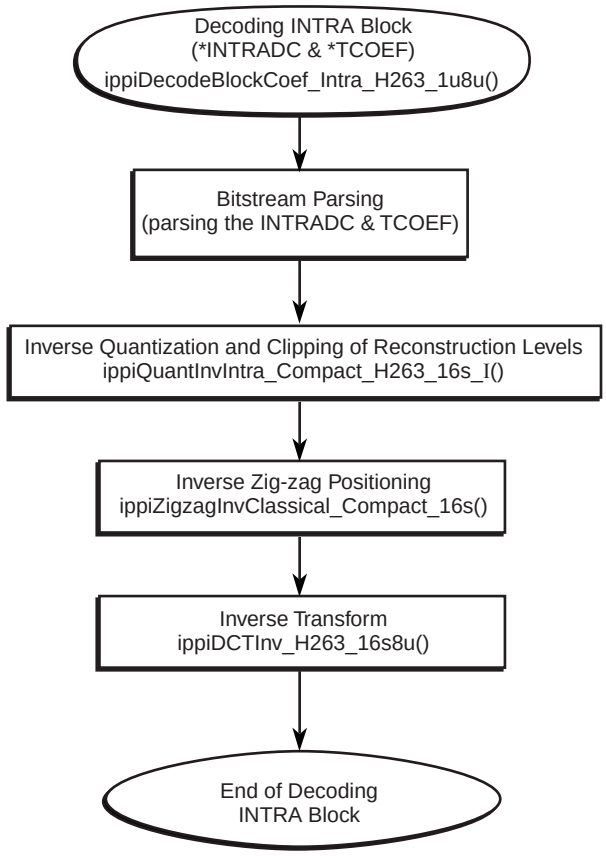
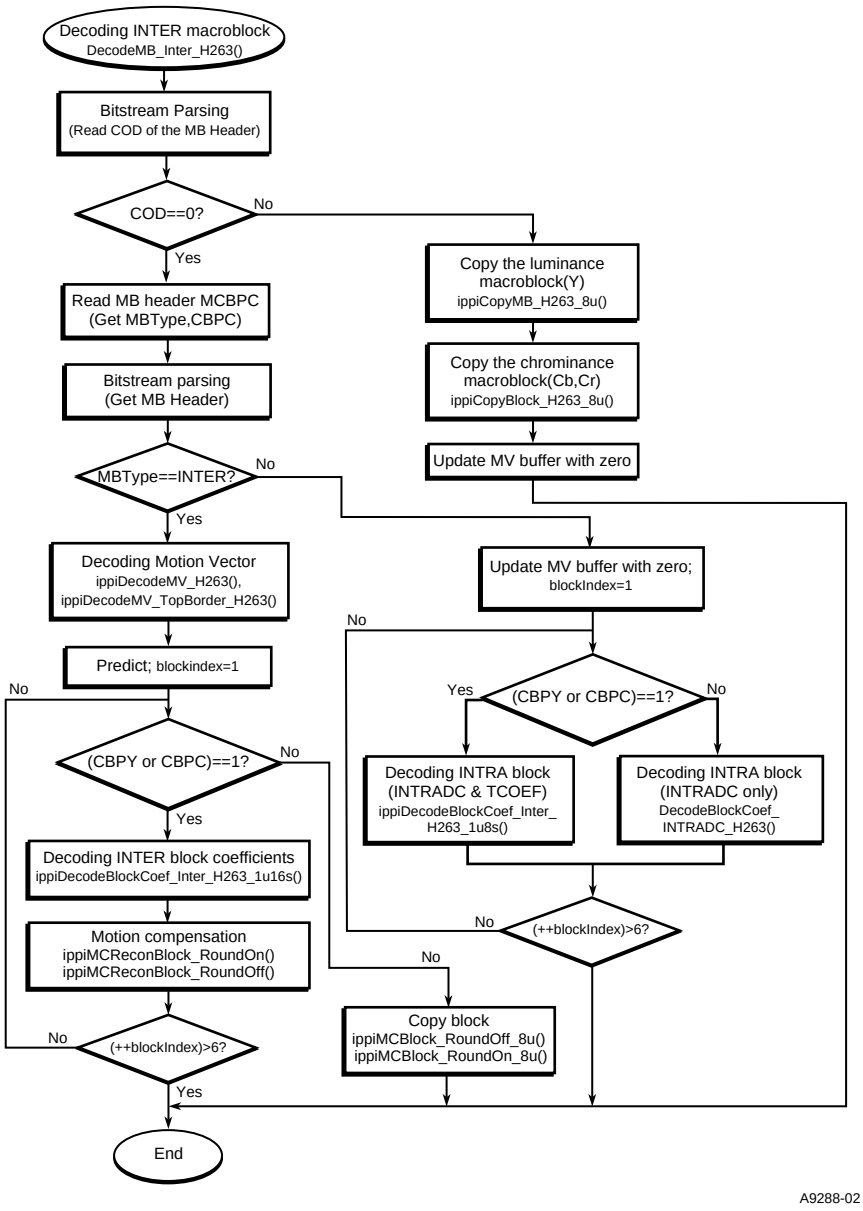


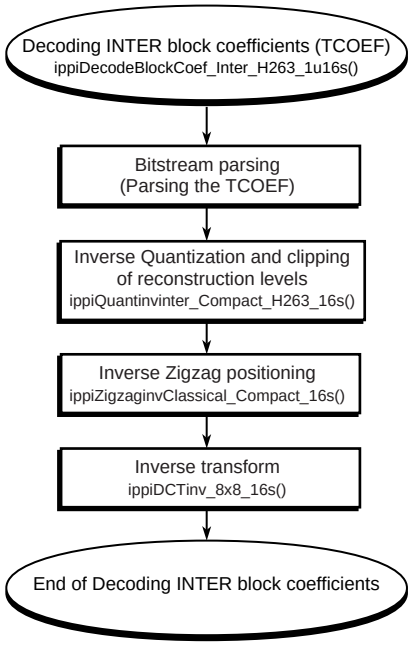
Figure 11-3 shows the process of decoding an inter macroblock.

Figure 11-3 The Process of Decoding INTER Macroblock



`ippiDecodeBlockCoef_Inter_H263_1u8u()` is a middle-level function. Its main part can be implemented through the low-level IPP functions as shown in Figure 11-4.

Figure 11-4 Decoding INTER Block Coefficients Using IPP Low Level Functions



A8856-01

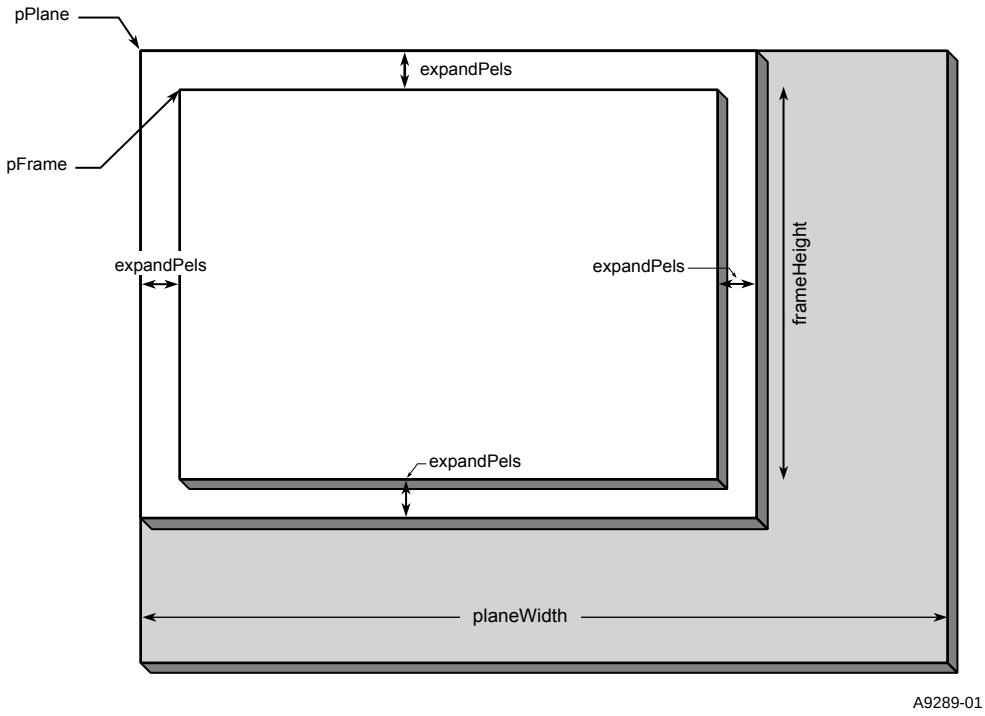
Structure and Macro Definitions

Motion Vector

The structure `IPPMotionVector` is defined as follows and be used in both H.263 and MPEG-4 video processing primitives.

```
typedef struct {
     Ipp16s dx;
     Ipp16s dy;
} IppMotionVector;
```

Figure 11-5 Plane, Frame, and Expanded Pixels



Step

Figure 11-5 describes the relationship between plane, frame and expanded pixels. The Step, frameWidth, frameHeight, and expandPels are defined as following:

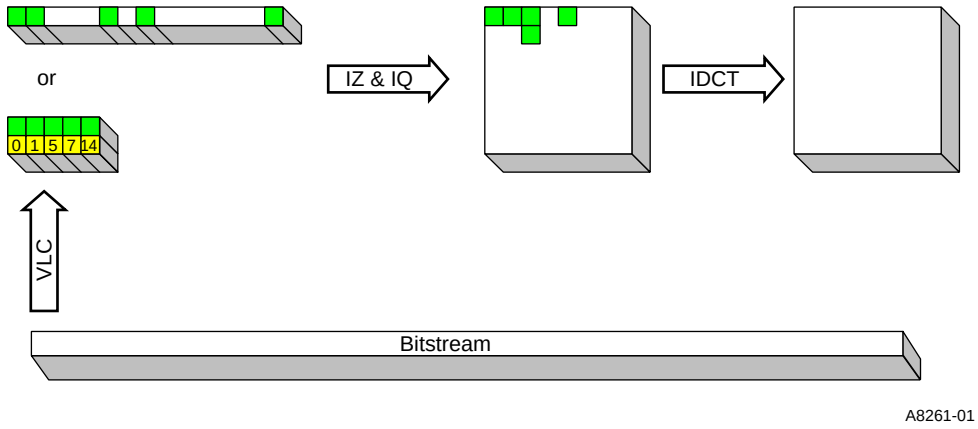
- `Step` – width of the plane, measured in pixels
- `Step` – width of the plane. – planeWidth ≥ frameWidth + 2 * expandPels
- `frameWidth` – width of the frame
- `frameHeight` – height of the frame
- `expandPels` – number of pixels to be expanded in one direction

In the default prediction mode of H.263, motion vectors are restricted such that all pixels referenced by them are within the coded picture area. Thus the value of `expandPels` can be zero, so that: Step = frameWidth.

When motion vectors are allowed to point outside the picture, such as in Annex D. We need expand the frame to plane, so that: Step = frameWidth + 2 * expandPels.

Compact

Figure 11-6 Compact Buffer



H.263 utilizes transform coding to reduce spatial redundancy. Therefore, by exploiting the sparseness of the blocks parsed from the bit stream, we can reduce the calculation in inverse zig-zag positioning and inverse quantization. From variable length decoding, we can know the index of the last non-zero coefficient. Then the buffer length for inverse zig-zag positioning and inverse quantization is much less than 64, so it is named a “compact” buffer. Figure 11-6 shows the concept of compact and full buffers.

Furthermore, in our middle-level implementation, variable length decoding, inverse zig-zag positioning and inverse quantization are incorporated in one step. The zeros between two non-zero coefficients are also squeezed. The buffer is more “compact” then.

Alignment

It is assumed that the frames, macroblocks and blocks are 64-bit aligned. That is:

```
(pPlane & 7) == 0 && (step & 7) == 0
```

General Video Processing and H.263 Decoder Primitives

This section lists all the general video processing primitives and H.263 basic decoder primitives. The general video processing primitives can be easily identified since they do not have the H.263 as "modifier" in the function name, while all H.263-oriented primitives have H.263 as part (modifier) of the function name.

DecodeMV_H263
DecodeMV_TopBorder_H263

Prototype

```
IppStatus ippiDecodeMV_H263 (Ipp8u ** ppBitStream, int * pBitOffset,
   IppMotionVector * pSrcDstMV);
IppStatus ippiDecodeMV_TopBorder_H263 (Ipp8u ** ppBitStream, int *
   pBitOffset, IppMotionVector * pSrcDstMV);
```

Description

Decodes the motion vector by predicting current MV according to three MVs around and adding to the differential motion vector data parsed from the bit stream.

See "Examples" for more detailed information and an illustration of motion vector decoding.

Input Arguments

- `ppBitStream` – indicates the pointer to the current byte in the bit stream buffer. There is no boundary check for the bit stream buffer.
- `pBitOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7.
- `pSrcDstMV` – pointer to the motion vector at the left side of the current macroblock in the motion vector buffer

NOTE. *The motion vectors are saved in a buffer with (number of macroblock per row + 2) in length, measured in the size of IppMotionVector.*

Output Arguments

- `ppBitStream` – `*ppBitStream` is updated after the block is decoded, so that it points to the current byte in the bit stream buffer. *`ppBitStream` should be located at the first bit of MVD in the bit stream prior to the function call.
- `pBitOffset` – `*pBitOffset` is updated so that it points to the current bit position in the byte pointed by `*ppBitStream. *pBitOffset should be located at the first bit of MVD in the bit stream prior to the function call.`
- `pSrcDstMV` – `*(pSrcDstMV + 1)` is updated to the decoded motion vector which will be used to predict the next motion vector and to perform motion compensation

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `ppBitStream`, `*ppBitStream`, `pBitStream`, pBitOffset is NULL.
  - `*pBitOffset` exceeds [0,7]
- `ippStsErr` – status error
  - 9 continuous zero encountered in parsing MVD from the bit stream

NOTE. *Illegal code in bit stream which can not be looked up in the VLC table can cause a status error.*

CopyMB_H263_8u
CopyBlock_H263_8u

Prototype

```
IppStatus ippiCopyMB_H263_8u (const Ipp8u * pSrc, Ipp8u * pDst, int
   step);
IppStatus ippiCopyBlock_H263_8u (const Ipp8u * pSrc, Ipp8u * pDst, int
   step);
```

Description

Copies the reference macroblock/block to the current macroblock/block.

NOTE. *When coded macroblock indication (COD) is set to "1", the macroblock is not coded. The macroblock should be treated as an INTER macroblock with motion vector for the whole block equal to zero and with no coefficient data. In this case, only macroblock copy is performed in baseline H.263 decoder. The macroblock copy is defined as:*

$$pDst[i * step + j] = pSrc[i * step + j]$$

where $i,j = [0,15]$

Input Arguments

- `pSrc` – pointer to the macroblock/block in the reference frame spatially correspondent to the macroblock/block in the current frame
- `step` – width of the source and destination planes

Output Arguments

`pDst` – pointer to the macroblock/block to be copied

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `pDst`, `pSrc` is NULL
  - `pSrc` or pDst is not 64-bit aligned
  - `step` is less than 8 (block version), 16 (MB version), or step is not a multiple of 8

QuantInvIntra_Compact_H263_16s_I
QuantInvInter_Compact_H263_16s_I

Prototype

```
IppStatus ippiQuantInvIntra_Compact_H263_16s_I(Ipp16s * pSrcDst, int
   len, int QP);
IppStatus ippiQuantInvInter_Compact_H263_16s_I(Ipp16s * pSrcDst, int
   len, int QP);
```

Description

Performs inverse quantization on an Intra or Inter coded block stored in a "compact" buffer.

NOTE. *Inverse quantization is defined as:*

$$pDst[i] = \begin{cases} 0, & pSrc[i] = 0 \\ Sign(pSrc[i]) * QP * (2 * |pSrc[i]| + 1) & pSrc[i] \neq 0, \text{QP is odd} \\ Sign(pSrc[i]) * (QP * (2 * |pSrc[i]| + 1) - 1) & pSrc[i] \neq 0, \text{QP is even} \end{cases}$$

$$\text{where } i = \begin{cases} 0, 1, ..., \text{len} - 1 & \text{INTER mode} \\ 1, 2, ..., \text{len} - 1 & \text{INTRA mode} \end{cases}, \text{and}$$

$$pDst[0] = \begin{cases} 1024, & pSrc[0] = 255, \text{and in INTRA mode} \\ 8 * pSrc[0], & \text{otherwise, and in INTRA mode} \end{cases}$$

Input Arguments

- `pSrcDst` – pointer to input (quantized) block
- `len` – length of the input and output compact buffer

- QP – quantization parameter

Output Arguments

- pSrcDst – pointer to output (reconstructed) block

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - pSrcDst is NULL
  - QP exceeds [1,31] or len exceeds [1,64]

ZigzagInvClassical_Compact_16s
ZigzagInvHorizontal_Compact_16s
ZigzagInvVertical_Compact_16s

Prototype

```
IppStatus ippiZigzagInvClassical_Compact_16s (const Ipp16s * pSrc, int
   len, Ipp16s * pDst);
IppStatus ippiZigzagInvHorizontal_Compact_16s (const Ipp16s * pSrc, int
   len, Ipp16s * pDst);
IppStatus ippiZigzagInvVertical_Compact_16s (const Ipp16s * pSrc, int
   len, Ipp16s * pDst);
```

Description

Performs classical, horizontal or vertical inverse zigzag scan on a block stored in a “compact” buffer.

NOTE. *An inverse scan is a mapping from the normal scan pattern to one zigzag scan pattern. For example, after performing a classical zigzag scan on a block, pDst[3] = pSrc[6].*

Input Arguments

- `pSrc` – pointer to input (zigzagged) block
- `len` – length of the input and output compact buffer

Output Arguments

`pDst` – pointer to output (normally scanned) block. The output is in a “full” buffer which contains 64 elements. The output buffer should be zeroed out prior to the function call.

Returns

IPP status code.

Three scan patterns are defined and compared with the normal scan as shown in the following figure (the numbers indicate the scanning sequence):

Figure 11-7 Zigzag Scan Patterns

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Classical Zigzag Scan

0	1	2	3	10	11	12	13
4	5	8	9	17	16	15	14
6	7	19	18	26	27	28	29
20	21	24	25	30	31	32	33
22	23	34	35	42	43	44	45
36	37	40	41	46	47	48	49
38	39	50	51	56	57	58	59
52	53	54	55	60	61	62	63

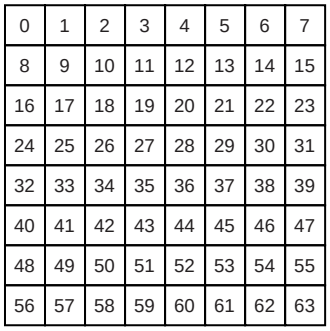
Alternate-Horizontal Scan

0	4	6	20	22	36	38	52
1	5	7	21	23	37	39	53
2	8	19	24	34	40	50	54
3	9	18	25	35	41	51	55
10	17	26	30	42	46	56	60
11	16	27	31	43	47	57	61
12	15	28	32	44	48	58	62
13	14	29	33	45	49	59	63

Alternate-Vertical Scan

A9120-01

Figure 11-8 Normal Scan Pattern



DCT8x8Inv_Video_16s8u_C1R
DCT8x8Inv_Video_16s_C1
DCT8x8Inv_Video_16s_C1I

Prototype

```
IppStatus ippiDCT8x8Inv_Video_16s8u_C1R (const Ipp16s * pSrc, Ipp8u *
   pDst, int dstStep);
IppStatus ippiDCT8x8Inv_Video_16s_C1 (const Ipp16s * pSrc, Ipp16s *
   pDst);
IppStatus ippiDCT8x8Inv_Video_16s_C1I (Ipp16s * pSrcDst);
```

Description

Performs 8x8 2D inverse discrete cosine transform and places the results in the destination plane.

NOTE. *The 2D (8 by 8) inverse DCT (normalized) is defined by:*

$$x_{n,m} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} c_u c_v X_{u,v} \cos\left[\frac{(2n+1)u\pi}{16}\right]\cos\left[\frac{(2m+1)v\pi}{16}\right]$$

with $u,v = 0, 1, \ldots, 7$

where n, m = spatial coordinates in the pixel domain,

u, v = coordinates in the transform domain,

$$c_l = \begin{cases} 1/\sqrt{2}, & l = 0 \\ 1, & l \neq 0 \end{cases}$$

The output is clipped to [0, 255].

Input Arguments

- `pSrc` – pointer to the input DCT coefficient. `pSrc` should be 64-bit aligned.
- `dstStep` – width of the destination plane.

Output Arguments

- `pDst` – pointer to the block in the destination plane

Returns

IPP status code.

NOTE. *pSrc and pSrcDst should be 64-bit aligned. (pSrcDst does not seem to be listed in Input or Output Args.)*

ReconMB_H263
ReconMB_H263_I
ReconBlock_H263
ReconBlock_H263_I

Prototype

```
IppStatus ippiReconMB_H263 (const Ipp8u * pSrc, const Ipp16s *
   pSrcResidual, Ipp8u * pDst, int step);
IppStatus ippiReconMB_H263_I (Ipp8u * pSrcDst, const Ipp16s *
   pSrcResidual, int step);
IppStatus ippiReconBlock_H263 (const Ipp8u * pSrc, const Ipp16s *
   pSrcResidual, Ipp8u * pDst, int step);
IppStatus ippiReconBlock_H263_I (Ipp8u * pSrcDst, const Ipp16s *
   pSrcResidual, int step);
```

Description

Reconstructs INTER macroblock/block by summing the prediction and the results of the inverse transformation (residuals).

NOTE. *To prevent quantization distortion of transform coefficient amplitudes causing arithmetic overflow in the encoder and decoder loops, clipping functions are inserted. Clipping range is [0, 255].*

Input Arguments

- `pSrc/pSrcDst` – pointer to input prediction in the reference plane
- `pSrcResidual` – pointer to the result of the inverse transformation in a residual buffer (64 elements)
- `step` – width of the source and destination planes

Output Arguments

`pDst/pSrcDst` – pointer to the reconstructed macroblock/block in the destination plane

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `pDst`, `pSrcDst`, `pSrcResidual` is NULL.
  - Any one of `pSrcDst`, `pDst`, or `pSrcResidual` is not 64-bit aligned.
  - `step` is less than 8 (block version), 16 (MB version) or step is not a multiple of 8.

MCReconBlock_RoundOff

Prototype

```
IppStatus ippiMCReconBlock_RoundOff(const Ipp8u *pSrc, int srcStep,
   Ipp16s * pSrcResidue, Ipp8u * pDst, int dstStep, int predictType);
```

Description

Reconstructs INTER block by summing the motion compensation results and the results of the inverse transformation (residuals). RCONTRL = 0.

NOTE. *RoundOff in the function name means rounding control parsed from the bit stream equals zero. The high level function has the responsibility to switch to the appropriate function according to the value of rounding control.*

Input Arguments

- `pSrc` – pointer to block in the reference plane
- `SrcStep` – width of the source plane. This should be greater than 8 and be a multiple of 8.
- `pSrcResidue` – pointer to the result of the inverse transformation in a residual buffer (64 elements). This should be aligned on an 8-byte boundary.
- `DstStep` – width of the destination plane
- `predictType` – bilinear interpolation type. Refer to "Bilinear Interpolation Type"

Output Arguments

`pDst` – pointer to the reconstructed block in the destination plane. This should be aligned on an 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `pSrc`, `pSrcResidue`, `pDst` is NULL
  - `predictType` is out of [0, 3]
  - `pDst` or `pSrcResidue` is not 8-byte aligned
  - `srcStep` is <= 8 or or not a multiple of 8
  - `DstStep` is <8 or not a multiple of 8

MCReconBlock_RoundOn

Prototype

```
IppStatus ippiMCReconBlock_RoundOn(const Ipp8u *pSrc, int srcStep,
   Ipp16s * pSrcResidue, Ipp8u * pDst, int dstStep, int predictType);
```

Description

Reconstructs INTER block by summing the motion compensation results and the results of the inverse transformation (residuals). RCONTRL = 1.

NOTE. *RoundOn in the function name means rounding control parsed from the bit stream equals one. The high level function has the responsibility to switch to the appropriate function according to the value of rounding control.*

Input Arguments

- pSrc – pointer to block in the reference plane
- srcStep – width of the source plane. This should be greater than 8 and be a multiple of 8.
- pSrcResidue – pointer to the result of the inverse transformation in a residual buffer (64 elements). This should be aligned on an 8-byte boundary.
- `DstStep` – width of the destination plane
- `predictType` – bilinear interpolation type. Refer to "Bilinear Interpolation Type".

Output Arguments

`pDst` – Pointer to the reconstructed block in the destination plane. This should be aligned on an 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: `pSrc`, `pDst` is NULL
  - `predictType` is out of [0, 3]
  - If `pDst` or `pSrcResidue` is not 8-byte aligned
  - `srcStep` is <= 8 or or not a multiple of 8
  - `DstStep` is <8 or not a multiple of 8

MCBlock_RoundOff_8u

Prototype

```
IppStatus ippiMCBlock_RoundOff_8u(const Ipp8u *pSrc, int srcStep, Ipp8u *
   pDst, int dstStep, int predictType);
```

Description

Do motion compensation and copy the result to the current block. RCONTRL = 0.

NOTE. *RoundOff in the function name means rounding control parsed from the bit stream equals zero. The high level function has the responsibility to switch to the appropriate function according to the value of rounding control.*

Input Arguments

- `pSrc` – pointer to block in the reference plane
- `srcStep` – width of the source plane. This should be greater than 8 and be a multiple of 8.
- `dstStep` – width of the destination plane
- `predictType` – bilinear interpolation type. Refer to "Bilinear Interpolation Type".

Output Arguments

`pDst` – Pointer to the collocated block in the destination plane. This should be aligned on an 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - If at least one of the pointers: `pSrc`, `pSrcResidue`, `pDst` is NULL
  - If `predictType` is out of [0, 3]
  - If `pDst` is not 8-byte aligned
  - `srcStep` is <= 8 or or not a multiple of 8
  - `DstStep` is <8 or not a multiple of 8

MCBlock_RoundOn_8u

Prototype

```
IppStatus ippiMCBlock_RoundOn_8u(const Ipp8u *pSrc, int srcStep, Ipp8u *
   pDst, int dstStep, int predictType);
```

Description

Do motion compensation and copy the result to the current block. RCONTRL = 1.

NOTE. *RoundOn in the function name means rounding control parsed from the bit stream equals one. The high level function has the responsibility to switch to the appropriate function according to the value of rounding control.*

Input Arguments

- `pSrc`– pointer to block in the reference plane
- `srcStep` – width of the source plane. This should greater than 8 and be a multiple of 8.
- `dstStep` – width of the destination plane
- `predictType` – bilinear interpolation type. Refer to "Bilinear Interpolation Type".

Output Arguments

`pDst` – Pointer to the collocated block in the destination plane. This should be aligned on an 8-byte boundary.

Returns

- `ippStsNoErr` – no error
  - `ippStsBadArgErr` – bad arguments
  - If at least one of the pointers: `pSrc`, `pSrcResidue`, `pDst` is NULL
  - If `predictType` is out of [0, 3]
  - `pDst` is not 8-byte aligned
  - `srcStep` is <= 8 or or not a multiple of 8
  - `DstStep` is <8 or not a multiple of 8

DCT8x8Fwd_Video_16s_C1I

Prototype

```
IppStatus ippiDCT8x8Fwd_Video_16s_C1I (Ipp16s * pSrcDst);
```

Description

Performs 8x8 2D forward discrete cosine transform in-place.

Input Arguments

`pSrcDst` – Pointer to the input block (in spatial domain). This should be aligned on an 8-byte boundary. Input should be limited to [-256, 255] (9-bit signed). Otherwise, the output will be incorrect.

Output Arguments

`pSrcDst` – Pointer to the transformed block (DCT coefficients). This should be aligned on an 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrcDst`
  - `pSrcDst` is not aligned on an 8-byte boundary

NOTE. *Limit the input to [-256, 255] (9-bit signed). Otherwise, the output will be incorrect.*

DCT8x8Fwd_Video_16s_C1

```
IppStatus ippiDCT8x8Fwd_Video_16s_C1 (const Ipp16s * pSrc, Ipp16s *
   pDst);
```

Description

Performs 8x8 2D forward discrete cosine transform for inter blocks.

Input Arguments

`pSrc` – Pointer to the input block (in spatial domain). This should be aligned on an 8-byte boundary. Limit the input to [-256, 255] (9-bit signed). Otherwise, the output will be incorrect.

Output Arguments

`pDst` – Pointer to the transformed block (DCT coefficients). This should be aligned on an 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrc`, `pDst`
  - pSrc is not 8-byte aligned
  - pDst is not 8-byte aligned

NOTE. *Limit the input to [-256, 255] (9-bit signed). Otherwise, the output will be incorrect.*

DCT8x8Fwd_Video_8u16s_C1R

Prototype

```
IppStatus ippiDCT8x8Fwd_Video_8u16s_C1R (const Ipp8u * pSrc, int srcStep,
   Ipp16s * pDst);
```

Description

Performs 8x8 2D forward discrete cosine transform for intra blocks.

Input Arguments

- `pSrc` – pointer to the input block (in spatial domain). This should be aligned on an 8-byte boundary.
- `srcStep` – width of the source plane. It should be no less than 8 and be a multiple of 8.

Output Arguments

`pDst` – pointer to the transformed block (DCT coefficients). This should be aligned on an 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrc`, `pDst`
  - `srcStep` <8 or is not a multiple of 8
  - `pSrc` is not 8-byte aligned
  - `pDst` is not 8-byte aligned

H.263+ Primitives

The primitives in this section provide Optional Mode Support for H263+ decoder.

ExpandFrame_H263_8u

Prototype

```
IppStatus ippiExpandFrame_H263_8u (Ipp8u * pSrcDstPlane, int frameWidth,
   int frameHeight, int expandPels, int step);
```

Description

Expands the frame to the plane in order to enable the motion vectors over picture boundaries feature.

Motion vectors over picture boundaries feature should be enabled if the annexes D, F, J of H.263+ are supported. When a pixel referenced by a motion vector is outside the coded picture area, an edge pixel is used instead.

It is assumed that the picture frame has been reconstructed prior to the function call.

Input Arguments

- `pSrcDstPlane` – pointer to plane
- `frameWidth` – width of the frame
- `frameHeight` – height of the frame
- `expandPels` – number of pixels to be expanded in one direction
- `step` – width of plane step. >= planeWidth = frameWidth + 2 * expandPels

Output Arguments

`pSrcDstPlane` – pointer to plane

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - `pSrcDstPlane` is NULL
  - `pSrcDstPlane` is not 64-bit aligned
  - Any one of `step`, `frameWidth`, or `expandPels` is less than 8
  - Any one of `step`, `frameWidth`, or `expandPels` is not a multiple of 8
  - `frameHeight` is less than or equal 0
  - `step` < (frameWidth + 2 * expandPels)

PredictBlock_OBMC_8u

Prototype

```
IppStatus ippiPredictBlock_OBMC_8u (const Ipp8u * pSrcRef, Ipp8u * pDst,
   int step, IppMotionVector * pMVCur, IppMotionVector * pMVLeft,
   IppMotionVector * pMVRight, IppMotionVector * pMVAbove,
   IppMotionVector * pMVBelow);
```

Description

Predicts current block from the reference frame using overlapped block motion compensation (OBMC).

Input Arguments

- `pSrcRef` – pointer to the block in the reference (source) plane spatially correspond to the block being predicted in the current plane. `pSrcRef` should be 64-bit aligned.
- `pMVCur`, `pMVLeft`, `pMVRight`, `pMVAbove`, `pMVBelow` – pointer to the current block and of the blocks left to, right to, above and below the current block.

NOTE. *Any one of the five motion vectors should be adjusted prior to the function call if the sample referenced by the motion vector stays outside the decode picture area.*

- `step` – width of the source and destination planes

Output Arguments

`pDst` – pointer to the compensated block in the destination plane. `pDst` should be 64-bit aligned.

Returns

IPP status code.

NOTE. *If one of the surrounding blocks was not coded, the pointer to the corresponding remote motion vector should be set to zero motion vector.*

If one of the surrounding blocks was coded in INTRA mode, or was outside the picture border, the pointer to the corresponding remote motion vector should be set to the pointer to the current motion vector the same as that passed to pMVCur).

If the current block is at the bottom of the macroblock, the remote motion vector corresponding with an 8 by 8 luminance block in the macroblock below the current macroblock should be replaced by the motion vector for the current block (as described in note 3, above).

FilterDeblocking_HorEdge_H263_8u_I
FilterDeblocking_VerEdge_H263_8u_I

Prototype

```
IppStatus ippiFilterDeblocking_HorEdge_H263_8u_I (Ipp8u * pSrcDst, int
   step, int QP);
IppStatus ippiFilterDeblocking_VerEdge_H263_8u_I (Ipp8u * pSrcDst, int
   step, int QP);
```

Description

Performs deblock filtering of one block edge (horizontal or vertical) on the reconstructed frames.

Input Arguments

- `pSrcDst` – pointer to the first pixel of the second block (block 2) of the two applied blocks. `pSrcDst` points to the first pixel of the block 2 as shown in Figure 11-9.
- `step` – width of the source and destination plane. `step` is a multiple of 8.
- `QP` – Quantization parameter. QP's value is found as described in Section J.3 of Annex J in H.263+

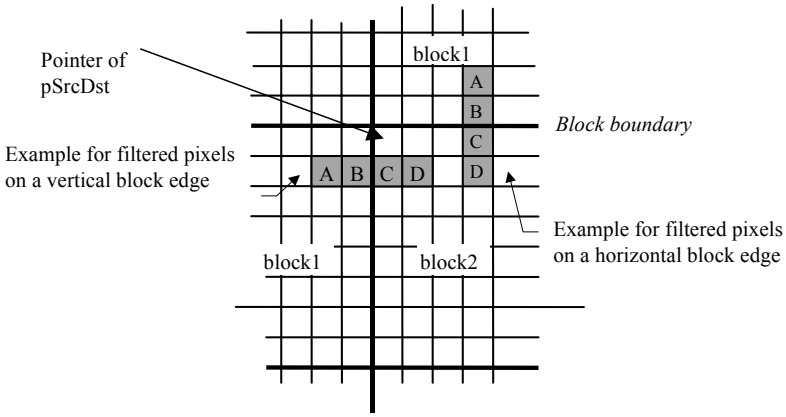
Output Arguments

`pSrcDst` – pointer to the first pixel of the second block (block 2) of the two applied blocks

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - `pSrcDst` is NULL
  - `pSrcDst` is not 64-bit aligned
  - `QP` exceeds [1,31]
  - `step` is not a multiple of 8 or step is less than 8

Figure 11-9 SrcDst Pointer



H.263+ Middle-Level Primitives

The middle-level primitives are described in this section. These primitives are built to performance the tasks that usually require several low-level primitives. A middle-level primitive call usually consumes fewer CPU cycles than the equivalent calls to several low-level primitives.

DecodeBlockCoef_Intra_H263_1u8u

Prototype

```
IppStatus ippiDecodeBlockCoef_Intra_H263_1u8u (Ipp8u ** ppBitStream, int
   * pBitOffset, Ipp8u * pDst, int step, int QP);
```

Description

Decodes the INTRA block coefficients. Inverse quantization, inverse zigzag positioning (classical) and IDCT, with appropriate clipping on each step, are performed on the coefficients. The results are then placed in the output frame/plane on a pixel basis.

For INTRA block, the output values are clipped to [0, 255] and written to current frame within the destination plane.

NOTE. *This function is used only when at least one non-zero AC coefficient of current block exists in the bit stream.*

The maximum absolute error is no more than 1.

Input Arguments

- `ppBitStream` – indicates the pointer to the current byte in the bit stream buffer. There is no boundary check for the bit stream buffer.
- `pBitOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7.
- `step` – width of the destination plane
- `QP`– quantization parameter (for non-INTRADC coefficients)

Output Arguments

- `ppBitStream` – `*ppBitStream` is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- `pBitOffset` – `*pBitOffset` is updated so that it points to the current bit position in the byte pointed by `*ppBitStream`
- `pDst` – pointer to the block in the destination plane

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `ppBitStream`,`*ppBitStream`, `pBitOffset`, `pDst` is NULL
  - `pDst` is not 64-bit aligned
  - `*pBitOffset` exceeds [0,7] or QP exceeds [1,31]
  - `step` is less than 8 or step is not a multiple of 8
- `ippStsErr` – status error
  - 0x00 or 0x80 is encountered in parsing DC coefficient
  - Invalid VLC code is encountered in parsing AC coefficient, if ESCAPE code encountered, 0x00 for LEVEL is deemed as error, but 0x80 is not

— Illegal code in bit stream which can not be looked up in VLC table could result in status error
— The count of the parsed code exceeds 64

DecodeBlockCoef_Inter_H263_1u16s

Prototype

```
IppStatus ippiDecodeBlockCoef_Inter_H263_1u16s (Ipp8u ** ppBitStream,
   int * pBitOffset, Ipp16s * pDst, int QP);
```

Description

Decodes the INTER block coefficients. Inverse quantization, inverse zigzag positioning (classical) and IDCT, with appropriate clipping on each step, are performed on the coefficients. The results (residuals) are placed in a contiguous array of 64 short int.

NOTE. *For INTER block, the output buffer holds the residuals for further reconstruction.*

The maximum absolute error is no more than 1.

Input Arguments

- `ppBitStream` – indicates the pointer to the current byte in the bit stream buffer. There is no boundary check for the bit stream buffer.
- `pBitOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7.
- `QP` – quantization parameter

Output Arguments

- `ppBitStream` – `*ppBitStream` is updated after the block is decoded, so that it points to the current byte in the bit stream buffer

- `pBitOffset` – *`pBitOffset` is updated so that it points to the up dated current bit position in the byte pointed by `*ppBitStream`
- `pDst` – pointer to the decoded residual buffer (a contiguous array of 64 short int)

Returns

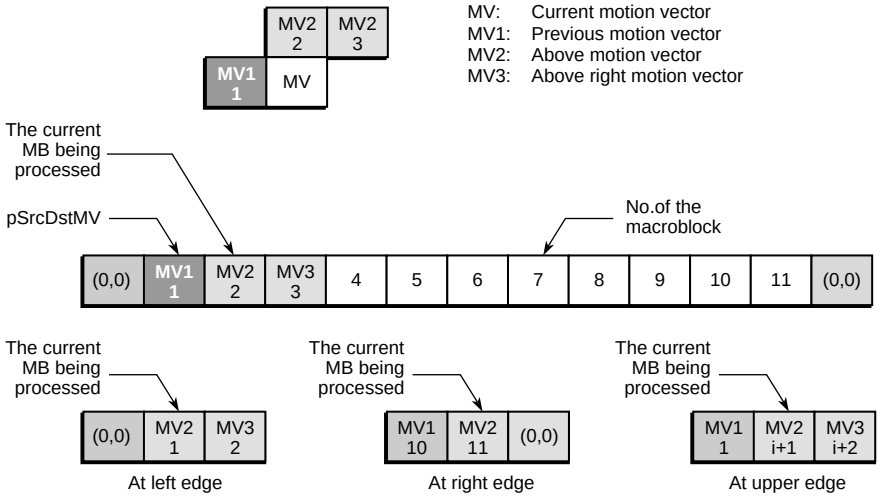
- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers: `ppBitStream`, *`ppBitStream`, `pBitOffset`, `pDst` is NULL
  - `pDst` is not 64-bit aligned
  - *`pBitOffset` exceeds [0,7] or QP exceeds [1,31]
- `ippStsErr` – status error
  - Invalid VLC code is encountered in parsing AC coefficient, if ESCAPE code encountered, 0x00 for LEVEL is deemed as error, but 0x80 is not
  - Illegal code in bit stream which can't be looked up in VLC table could result in status error
  - The count of the parsed code exceeds 64

Examples

Motion Vector Decoding

- The motion vectors are saved in a buffer with [number-of-macroblock-per-row +2] * size of (IppMotionVector) in length (byte). The first and the last motion vectors in this buffer are always (0,0). The n-th unit in this buffer, excluding the first and last one, is the motion vector of the (n-1)th macroblock in a line.
- When the corresponding macroblock is at the top of the picture or at the top of the GOB if the GOB header of the current GOB is non-empty, function `ippiDecodeMV_TopBorder_H263()` is employed. The candidate predictor is stored on the left of the current one in the buffer.
- If not any of the above cases, in H263+ baseline, `ippiDecodeMV_H263()` is employed. The three candidate predictors (MV1, MV2, MV3) are stored as Figure 11-10 depicts.
- The buffer is updated after function has been called. The decoded motion vector replaces the candidate predictor MV2 for future use.

Figure 11-10 Motion Vector Decoding




```
IppStatus ippsDecodeMV_H263_1u8s
     (Ipp8u  *  pBitStream,  int  *  pBitOffset, IppMotionVector * pScrDstMV);
```

A8857-01

MPEG-4 Video Decoder 12

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) that are built to support the ISO/IEC 14496-2 MPEG-4 video decoder. MPEG-4 is a widely used coding method for video signals in various applications such as digital storage media, internet, various forms of wired or wireless communication, etc.

To benefit application developers, the design philosophy of the primitives, like all other primitives, emphasizes maximum flexibility and performance. On the one hand, developers have the option of building a complete MPEG-4 decoder solution using the compact set of performance optimized MPEG-4 primitives described in this chapter in conjunction with the user's administrative and memory management functions customized for the application environment. In this scenario, developers are able to leverage the fact that the MPEG-4 primitives have been tuned carefully for minimum cycle count, minimum memory footprint, and maximum quality. On the other hand, developers also have the option of building a custom MPEG-4 decoder while electing to use only a subset of the Intel® IPP MPEG-4 primitives. This development option is facilitated in the API by providing access to the intermediate computational results generated by each of the MPEG-4 routines. Finally, the API allows the user to fully exploit performance properties of a particular target operating system (OS) by allowing user management of administrative functions such as the high-level bit stream manipulation, memory allocation/deallocation, and control of the various buffers, etc.

The primitives cover the following aspects of MPEG-4 decoder:

- Progressive, non-scalable texture decoding and shape decoding
- Macroblock-based repetitive and extended padding
- Block-based VLC decoding and inverse zigzag scan
- Motion vector (include Motion Vector for shape) decoding and padding
- Intra DC/AC prediction
- Block layer coefficient decoding, including bit stream parsing, VLC decoding, intra DC/AC prediction (for intra blocks), inverse quantization, inverse zigzag and IDCT, with appropriate clipping on each step.

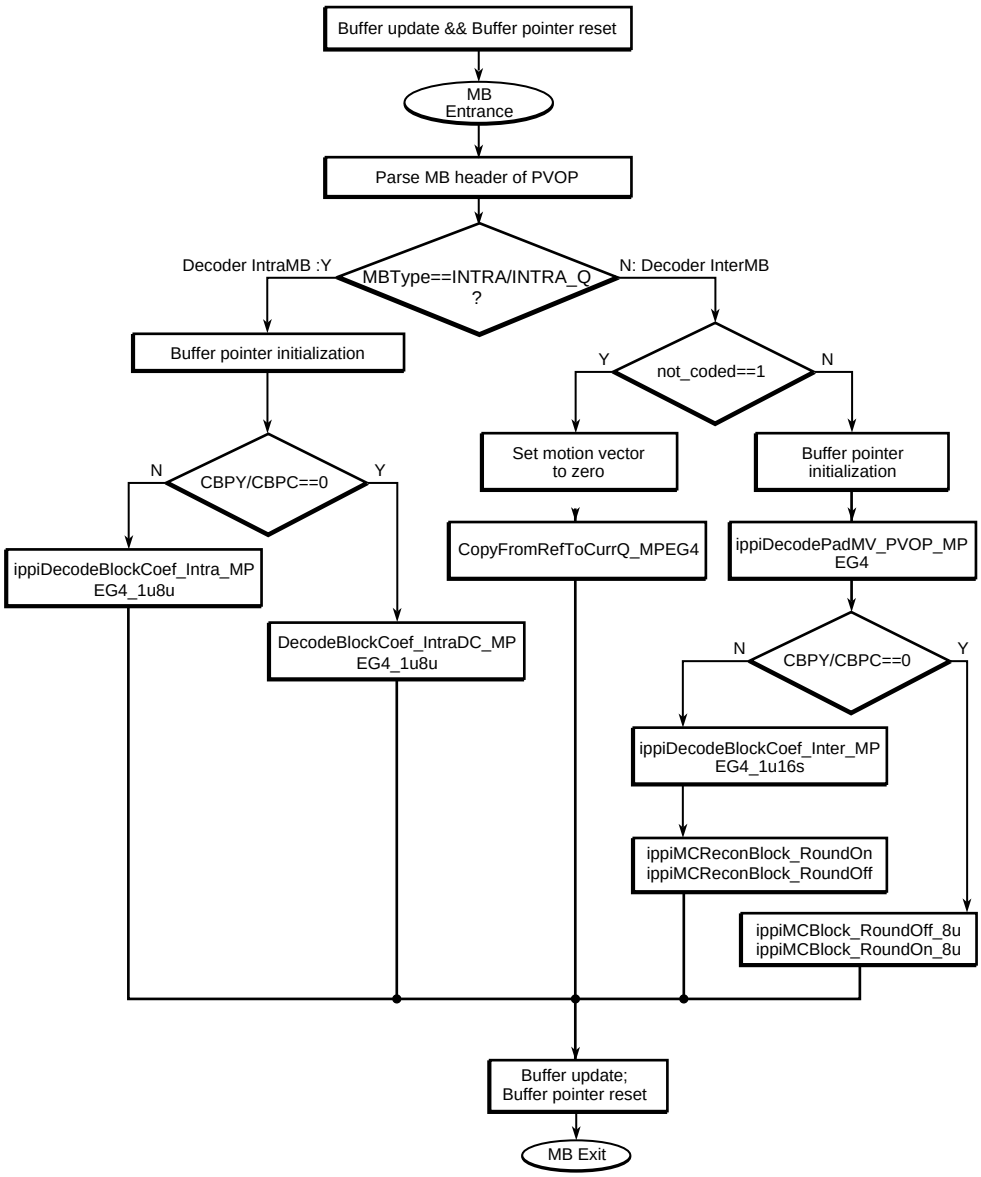
- Motion compensation and reconstruction
- BAB decoding.

The rest of this chapter provides details on the MPEG-4 API, and is organized as follows. First, section "High-Level Description" gives a high level description of on the usage of MPEG-4 primitives. The signal flowcharts are given to show the data flow. Next, section "Data Types and Structures" focuses on individual macro/data structures used in the primitives. Section "MPEG-4 Decoder Primitives" focuses on each API definition and function.

High-Level Description

Figure 12-1 shows the general steps to take to decode a MacroBlock (MB) in Predictive coded Video Object Plane (PVOP) and the associated primitives.

Figure 12-1 Signal Flowchart in Decode a MB in PVOP and the Associated Primitives



A9290-02

Data Types and Structures

The following sections describe the data types and structures for the Intel® IPP MPEG-4 video decoder.

Video Components

Video components are defined as follows:

```
typedef enum {
    IPP_VIDEO_LUMINANCE,       /* Luminance component   */
    IPP_VIDEO_CHROMINANCE,     /* Chrominance component */
    IPP_VIDEO_ALPHA            /* Alpha component       */
} IppVideoComponent;
```

Pixel Planes and Alpha Plane

The decoder's output is stored in five planes (if shape mode selected). They are three texture planes denoted by Y plane (luminance component), Cb plane and Cr plane (chrominance components), one grayscale alpha plane denoted by A plane (alpha component) and one Binary plane denoted by B plane (binary component).

The size of the Y plane relates to, but usually is not equal to, that of the VOL as a result of the VOP expansion. Since luminance VOP is generally expanded (and padded) with 16 pixels to each of the four directions, the width and height of the Y plane are 32 pixels larger than those of the VOL respectively. Users are recommended to allocate memory for the VOP planes for their needs. The primitives in this sections do not operate directly on the VOP planes, but on the block level.

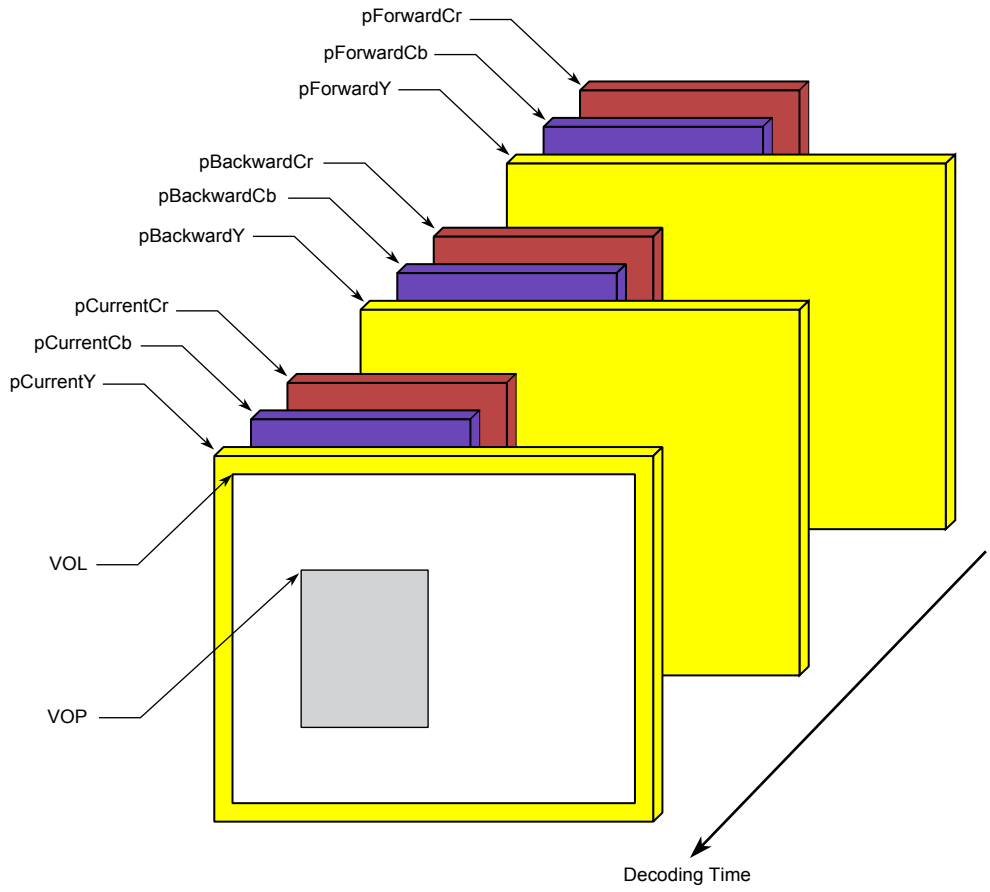
The size (W, H) of Cb or Cr plane is half the size of Y plane, because chrominance VOPs are expanded with 8 pixels to each direction.

The A and B plane has the same pixel size as the Y plane. All pixels in the Y/A/Cb/Cr plane occupy 8 bits, while a pixel in the B plane occupies 1 bit.

Figure 12-2 shows the relationship among pixel plane, VOL and VOP.

Three sets of pixel planes, each consists of a Y plane, a Cb plane and a Cr plane, should be 64-bit aligned in user's allocation. They are referred to as current, forward and backward. Backward pixel plane set is not required in Simple profile, because it does not support B-VOP.

Figure 12-2 Pixel Plane, VOL and VOP



Macroblock Types

Macroblock types in Intra coded (I-), Predictive coded (P-) and Bidirectional Predictive coded (B-) VOP are defined as following:

```
typedef enum {
      IPP_VIDEO_INTER             = 0,   /* P picture or P-VOP */
      IPP_VIDEO_INTER_Q           = 1,   /* P picture or P-VOP */
      IPP_VIDEO_INTER4V           = 2,   /* P picture or P-VOP */
      IPP_VIDEO_INTRA             = 3,   /* I and P picture, or I- and P-VOP */
      IPP_VIDEO_INTRA_Q           = 4,   /* I and P picture, or I- and P-VOP */
      IPP_VIDEO_INTER4V_Q         = 5,   /* P picture or P-VOP (H.263 only)*/
      IPP_VIDEO_DIRECT            = 6,   /* B picture or B-VOP (MPEG-4 only) */
      IPP_VIDEO_INTERPOLATE       = 7,   /* B picture or B-VOP */
      IPP_VIDEO_BACKWARD          = 8,   /* B picture or B-VOP */
      IPP_VIDEO_FORWARD           = 9    /* B picture or B-VOP */
      IPP_VIDEO_NOTCODED          = 10   /* B picture or B-VOP */
} IppMacroblockType;
```

NOTE. *About the Suffix:*
Q - Quantization Parameter should be updated in indicated macroblock.
4V - 4 blocks in indicated macroblock have respective motion vector, otherwise, 4 blocks in indicated macroblock have same motion vector.

Motion Vector

Motion vector is defined as in "Motion Vector"

```
typedef struct {
    Ipp16s dx;
    Ipp16s dy;
} IppMotionVector;
```

Depending on the MB type, there are up to eight valid motion vector(s) in a MB. Below is the vector manipulation scheme adopted by IPP MPEG-4 CODEC, where `pMVForward` and `pMVBackward` denote the two vector buffers allocated for each MB:

- Two buffers per MB in P or B-VOP, including `pMVForward[4]` and `pMVBackward[4]`.
- Each element contains a block based motion vector, contiguously stored per each buffer.

- In P-VOP, only `pMVForward[ ]` is used and valid; `pMVBackward[ ]` is not used.
- If MB type is "`IPP_VIDEO_INTER`" or "`IPP_VIDEO_INTER_Q`", and if not transparent, `pMVForward[0]~[3]` must be filled with the same decoded MV.
- If MB is INTRA coded or skipped, `pMVForward[0]~[3]` should be padded with zero MVs.
- In B-VOP, `pMVForward[]` and `pMVBackward[]` may or may not be used, which depends on the MB type.
- If B_VOP, if MB type is not "`IPP_VIDEO_DIRECT`", `pMVForward[1]~[3]` and `pMVBackward[1]~[3]` are NOT used.

NOTE. *Coordinates are related to the absolute coordinate system shown in Figure 7-19 of* ISO/IEC 14496-2.

Transparent Status

Transparent status is a three-state value in one byte, or `Ipp8u`. The three possible states are defined as:

```
enum {
    IPP_VIDEO_TRANSPARENT      = 0,    /* Wholly transparent */
    IPP_VIDEO_PARTIAL          = 1,    /* Partially transparent */
    IPP_VIDEO_OPAQUE           = 2     /* Opaque */
};
```

Transparent status is block based in MPEG-4. Thus,

- One buffer per MB
- Four elements per buffer
- Each element occupies one byte (Ipp8u) for one block
- The first element (for block 0) must be *32-bit aligned* (which should be ensured by user)
- MB transparent status could be known by evaluating the value of the whole word directly.

Quantization Parameter

Quantization parameters of intra-coded macroblocks should be stored in order to perform DC and AC prediction for the intra-coded macroblocks spatially to the right and/or below, if they exist.

- One row buffer for current dealt VOP.
- The buffer is used for coefficient prediction.

- Before decoding an "intra" or "intra+q" MB, the buffer saves the QPs of the upper MB and left MB if they exist.
- After an "intra" or "intra+q" MB is decoded, the corresponding QP buffer (who stored the MB spatially above before) should be updated by the current QP.
- Each element is one byte (Ipp8u) for one MB.

Direction

Direction is concerned when performing DC/AC prediction and zigzag scan.

```
enum {
    IPP_VIDEO_NONE          = 0,
    IPP_VIDEO_HORIZONTAL    = 1,
    IPP_VIDEO_VERTICAL      = 2
};
```

Rectangle Plane

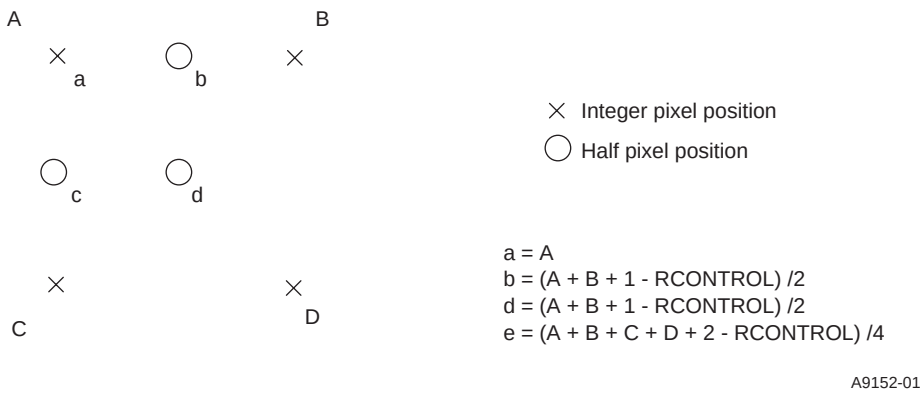
```
typedef structure _IppiRect {
    int  x;
    int  y;
    int  width;
    int  height;
}
```

Bilinear Interpolation Type

Bilinear interpolation type is used for motion compensation and reconstruction.

```
enum {
    IPP_VIDEO_INTEGER_PIXEL   = 0,   /* case "a" in Figure 12-3 */
    IPP_VIDEO_HALF_PIXEL_X    = 1,   /* case "b" in Figure 12-3 */
    IPP_VIDEO_HALF_PIXEL_Y    = 2,   /* case "c" in Figure 12-3 */
    IPP_VIDEO_HALF_PIXEL_XY   = 3    /* case "d" in Figure 12-3 */
```

Figure 12-3 Halfpixel Prediction by Bilinear Interpolation



Buffers

This section describes buffers and their layout required by Intel® IPP. The user must allocate and/or initialize the buffers according to the following specifications:

- Video plane buffers

  User should allocate buffers to store the decoded picture (H.263) or video object (MPEG-4) consisting of texture components (Y/Cb/Cr) and alpha components (binary/grayscale) if shape coding is supported. The presence of each component depends on the VOL shape type according to `video_object_layer_shape`. Table 12-1 details the dependency, where “x” means required and “-” means not required.

Table 12-1 Video Buffer Allocation Requirements

VOL Shape Type	Texture Y/Cb/Cr Planes	Binary Alpha Plane	Grayscale Alpha Plane
rectangular	x	-	-
binary	x	x	-
binary only	-	x	-
grayscale	x	x	x

Two sets of these buffers should be available – one for the current picture/VO, the other for the previous (forward-reference) picture/VO; an additional set of buffers should also be allocated if bi-directional prediction (B-VOP) is supported. A more detailed description of the video planes could be found in section "Pixel Planes and Alpha Plane".

- Motion vector buffers

  A motion vector buffer contains four elements of IppMotionVector data. Each element contains a block based motion vector, contiguously stored per each buffer. A user should allocate one MV buffer per MB for P-VOP; if B-VOP is supported, two MV buffers should be available for bi-directional prediction.

  If macroblock type is `IPP_VIDEO_INTER or IPP_VIDEO_INTER_Q`, but not `IPP_VIDEO_INTER4V`, and if it is not transparent, the four elements must be filled with the same MV.

  If MB is INTRA coded or skipped, the four elements should be padded with zero MVs.

  In B-VOP, the forward or backward MV buffer may or may not be used, which depends on the MB type.

  If MB type is not `IPP_VIDEO_DIRECT` in B-VOP, elements [1]~[3] of either buffers are NOT used; only elements [0] may be used.

- Transparent status buffer

  User should allocate one transparent status buffer for the MB. Elements, of type `Ipp8u`, are block based, contiguously stored. There are 4 elements for a MB, corresponding to the 16x16 area.

NOTE. *The address of the first element (for block 0) must be 32-bit aligned, which should be ensured by the user. MB transparent status is then detected by evaluating the value of the whole word.*

- Quantization parameter buffer

  User should allocate one “row buffer” storing the quantization parameters (QPs) with one byte (`Ipp8u`) for each element. This buffer is used for coefficient prediction. Before decoding an INTRA coded MB, the corresponding element in the QP buffer saves the QP of the MB of the upper MB row. After an MB is decoded, the corresponding element (the element storing QP of the MB above) should be updated by the QP of the current MB.

- Coefficient buffers

  User should allocate two coefficient buffers for Intra DC/AC prediction – a row buffer that contains ((mb_num_per_row * 2 + 1) * 8) elements of Ipp16s, and a column buffer that contains 16 elements of Ipp16s.

  Every 8 elements of both row and column buffers, plus one element 8 units ahead in row buffer, are used to perform DC/AC prediction for an INTRA coded block in a MB. Each group stores the coefficient predictors of the neighbor block spatially upper or left to the block currently to be decoded. Within every 8 elements, the first one stores the DC coefficient and the others store quantized AC coefficients. A negative-valued DC coefficient signals that this neighbor block is not INTRA coded, thus neither the DC nor the AC coefficients are valid.

  All DC elements in row buffer should be initialized to -1 prior to decoding each VOP. In addition, the two DC elements in column buffer should also be initialized to -1 prior decoding each MB row.

  If the current MB/block is transparent, or the MB_Type is IPP_VIDEO_INTER/ IPP_VIDEO_INTER_Q/ IPP_VIDEO_INTER_4V, the corresponding DC elements in the row buffer and column buffer should also be initialized to –1 to indicate that no predictor for later AC/DC prediction.

  The detailed coefficient buffer layout is illustrated in Figure 12-4.

- BAB-mode buffer

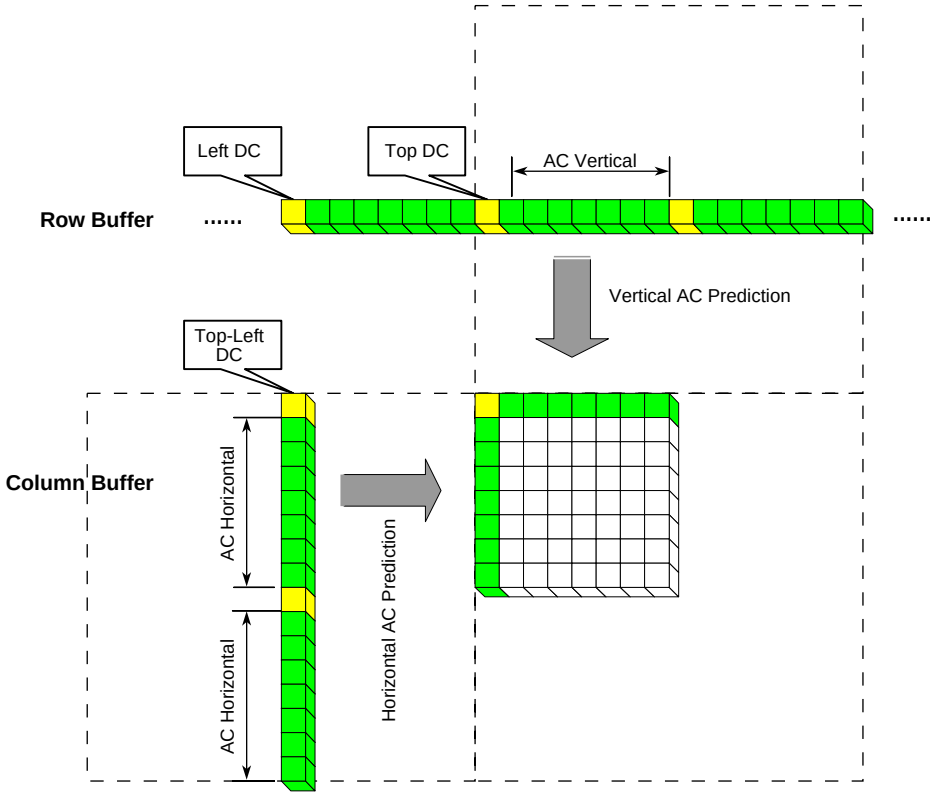
  User should allocate one BAB mode buffer storing the mode of each BAB. There are seven types of BAB, and before decoding further shape information, the current BAB mode should be decoded first.

  The detail information about BAB mode can refer to subclause 7.6.1.6 of ISO/IEC 14496-2.

- Motion vector for shape buffer

  User should allocate one buffer storing the MVs. There is one MVs for a BAB, so the buffer size decided by the BAB number of current plane.

Figure 12-4 Coefficient Buffer Layout Chart




```
DC Coefficient Prediction:
     QFx[0][0] = PQFx[0][0] + Fp[0][0]//dc_scaler

AC Coefficient Prediction:
     QFx[v][0] = PQFx[v][0] + (QFp[v][0] * QPp)//QPx   v = 1 to 7
  or:
     QFx[0][u] = PQFx[0][u] + (QFp[0][u] * QPp)//QPx   u = 1 to 7

API:
IppStatus ippiPredictReconCoefIntra_MPEG4_16s (
     Ipp16s * pSrcDst, Ipp16s * pPredBufRow, Ipp16s * pPredBufCol,
     int curQP, int predQP, int predDir, int ACPredFlag,
     IppVideoComponent videoComp
);
```

A8418-01

MPEG-4 Decoder Primitives

This section describes all of the MPEG4 Video Decoding Domain primitives. See:

- Chapter 12 for a short listing of the all the primitives in this chapter.
- The Index section for an alphabetic list of all IPPs.
- The Contents which lists the IPPs as they appear in the chapter.

DecodePadMV_PVOP_MPEG4

Prototype

```
IppStatus ippiDecodePadMV_PVOP_MPEG4 (Ipp8u ** ppBitStream,
   int * pBitOffset, IppMotionVector * pSrcMVLeftMB, IppMotionVector *
   pSrcMVUpperMB, IppMotionVector * pSrcMVUpperRightMB, IppMotionVector
   * pDstMVCurMB, Ipp8u * pTranspLeftMB, Ipp8u * pTranspUpperMB, Ipp8u *
   pTranspUpperRightMB, Ipp8u * pTranspCurMB, int fcodeForward,
   IppMacroblockType MBType);
```

Description

Decodes and pads four motion vectors of the non-intra macroblock in P-VOP.

The motion vector padding process is specified in subclause 7.6.1.6 of ISO/IEC 14496-2.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream buffer
- `pBitOffset` – pointer to the bit position in the byte pointed to by `*ppBitStream`. `*pBitOffset` is valid within [0-7].
- `pSrcMVLeftMB`
  `pSrcMVUpperMB`
  `pSrcMVUpperRightMB` – pointers to the motion vector buffers of the macroblocks specially at the left, upper and upper-right side of the current macroblock respectively.
- `pTranspLeftMB`
  `pTranspUpperMB`
  `pTranspUpperRightMB`

pTranspCurMB – pointers to the transparent status buffers of the macroblocks specially at the left, upper and upper-right side of, and the current macroblock respectively. Set to IPP_VIDEO_TRANSPARENT if outside boundary (see Note below).

- fcodeForward – a code equal to vop_fcode_forward in MPEG-4 bit stream syntax
- MBType – the type of the current macroblock. If MBType is not equal to IPP_VIDEO_INTER4V, the destination motion vector buffer is still filled with the same decoded vector.

Output Arguments

- ppBitStream – *ppBitStream is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- pBitOffset – *pBitOffset is updated so that it points to the current bit position in the byte pointed by *ppBitStream
- pDstMVCurMB – pointer to the motion vector buffer of the current macroblock which contains four decoded motion vectors

Returns

- IppStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - — At least one of the following pointers is NULL: ppBitStream, *ppBitStream, pBitOffset, pTranspLeftMB, pTranspUpperMB, pTranspUpperRight, pTranspCurMB, pDstMVCurMB
    or
  - — At least one of following cases is true: *pBitOffset exceeds [0,7], fcodeForward exceeds (0,7], MBType less than zero, transparent status or the motion vector buffer is not 32-bit aligned.
- ippStsErr – status error
  - — Meet illegal code in bit stream which can not be looked up in VLC table

NOTE. *Any neighborhood macroblock outside the current VOP or video packet or outside the current GOB (when short_video_header is "1") for which gob_header_empty is "0" is treated as transparent, according to subclause 7.6.5 in* ISO/IEC 14496-2.

DecodeMV_BVOP_Backward_MPEG4

Prototype

```
IppStatus ippiDecodeMV_BVOP_Backward_MPEG4 (Ipp8u ** ppBitStream, int *
   pBitOffset, IppMotionVector * pSrcDstMVB, int fcodeBackward);
```

Description

Decodes motion vectors of the macroblock in B-VOP backward mode. After decoding a macroblock of backward mode only the backward predictor is set to the decoded backward vector.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream buffer
- `pBitOffset` – pointer to the bit position in the byte pointed to by *ppBitStream. *pBitOffset is valid within [0-7].
- `pSrcDstMVB` – pointer to the backward motion vector predictor
- fcodeBackward – a code equal to vop_fcode_backward in MPEG-4 bit stream syntax

Output Arguments

- `ppBitStream` – *ppBitStream is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- `pBitOffset` – *pBitOffset is updated so that it points to the current bit position in the byte pointed by *ppBitStream
- `pSrcDstMVB` – pointer to the backward motion vector of the current macroblock. The backward motion vector predictor should be reset to zero at the beginning of each macroblock row.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `ppBitStream`, *`ppBitStream`, `pBitOffset`, `pSrcDstMVB`.
  - At least one of following cases: *pBitOffset exceeds [0,7], fcodeBackward exceeds (0,7].
- ippStsErr – status error
  - Meet illegal code in bit stream which can not be looked up in VLC table

DecodeMV_BVOP_Forward_MPEG4

Prototype

```
IppStatus ippiDecodeMV_BVOP_Forward_MPEG4 (Ipp8u ** ppBitStream, int *
   pBitOffset, IppMotionVector * pSrcDstMVF, int fcodeForward);
```

Description

Decodes motion vectors of the macroblock in B-VOP forward mode. After decoding a macroblock of forward mode only the forward predictor is set to the decoded forward vector.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream buffer
- `pBitOffset` – pointer to the bit position in the byte pointed to by *ppBitStream. *pBitOffset is valid within [0-7]
- `pSrcDstMVF` – pointer to the forward motion vector predictor
- `fcodeForward` – a code equal to vop_fcode_forward in MPEG-4 bit stream syntax

Output Arguments

- `ppBitStream` – `*ppBitStream` is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- `pBitOffset` – `*pBitOffset` is updated so that it points to the current bit position in the byte pointed by *ppBitStream
- `pSrcDstMVF` – pointer to the forward motion vector of the current macroblock. The forward motion vector predictor should be reset to zero at the beginning of each macroblock row.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the following pointers is NULL: `ppBitStream`, `*ppBitStream`, `pBitOffset`, `pSrcDstMVF`.
    or

— At least one of following cases is true:
– *pBitOffset exceeds [0,7]
– fcodeForward exceeds [0,7]
– The motion vector buffer is not 32-bit aligned

- ippStsErr – status error
  — Meet illegal code in bit stream which can not be looked up in VLC table

DecodeMV_BVOP_Interpolate_MPEG4

Prototype

```
IppStatus ippiDecodeMV_BVOP_Interpolate_MPEG4 (Ipp8u ** ppBitStream, int
   * pBitOffset, IppMotionVector * pSrcDstMVF, IppMotionVector *
   pSrcDstMVB, int fcodeForward, int fcodeBackward);
```

Description

Decodes motion vectors of the macroblock in B-VOP interpolate mode. After decoding a macroblock of interpolate mode both the forward and backward predictor are updated separately with the decoded vectors of the same type (forward/backward).

Input Arguments

- ppBitStream – pointer to the pointer to the current byte in the bit stream buffer
- pBitOffset – pointer to the bit position in the byte pointed to by *ppBitStream. *pBitOffset is valid within [0-7].
- pSrcDstMVF – pointer to the forward motion vector predictor. The forward motion vector predictor should be reset to zero at the beginning of each macroblock row.
- pSrcDstMVB – pointer to the backward motion vector predictor. The backward motion vector predictor should be reset to zero at the beginning of each macroblock row.
- fcodeForward – a code equal to vop_fcode_forward in MPEG-4 bit stream syntax
- fcodeBackward – a code equal to vop_fcode_backward in MPEG-4 bit stream syntax

Output Arguments

- ppBitStream – *ppBitStream is updated after the block is decoded, so that it points to the current byte in the bit stream buffer

- pBitOffset – *pBitOffset is updated so that it points to the current bit position in the byte pointed by *ppBitStream
- pSrcDstMVF – pointer to the forward motion vector of the current macroblock
- pSrcDstMVB – pointer to the backward motion vector of the current macroblock

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - — At least one of the following pointers is NULL: ppBitStream *ppBitStream pBitOffset, pSrcDstMVF, pSrcDstMVB
    or
  - — At least one of the following cases is true:
    – *pBitOffset exceeds [0,7]
    – fcodeForward or fcodeBackward exceeds (0,7]
- ippStsErr – status error
  - — Meet illegal code in bit stream which can not be looked up in VLC table

DecodeMV_BVOP_Direct_MPEG4

Prototype

```
IppStatus ippiDecodeMV_BVOP_Direct_MPEG4 (Ipp8u ** ppBitStream, int *
   pBitOffset, const IppMotionVector * pSrcMV, IppMotionVector *
   pDstMVF, IppMotionVector * pDstMVB, Ipp8u * pTranspSrcMB, int TRB, int
   TRD);
```

Description

Decodes motion vector(s) of the macroblock in B-VOP using direct mode.

Input Arguments

- ppBitStream – pointer to the pointer to the current byte in the bit stream buffer
- pBitOffset – pointer to the bit position in the byte pointed to by *ppBitStream. *pBitOffset is valid within [0-7].

- `pSrcMV` – pointer to the motion vector buffer of the co-located macroblock in the most recently decoded I- or P-VOP
- `pTranspSrcMB` – pointer to the transparent status buffer of the co-located macroblock
- TRB – the difference in temporal reference of the B-VOP and the previous reference VOP
- TRD – the difference in temporal reference of the temporally next reference VOP with temporally previous reference VOP

Output Arguments

- `ppBitStream` – *ppBitStream is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- `pBitOffset` – *pBitOffset is updated so that it points to the current bit position in the byte pointed by *ppBitStream
- `pDstMVF` – pointer to the forward motion vector buffer of the current macroblock which contains decoded forward motion vector
- `pDstMVB` – pointer to the backward motion vector buffer of the current macroblock which contains decoded backward motion vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `ppBitStream`, `*ppBitStream`, `pBitOffset`, `pSrcMV` , `pTranspSrcMB`, `pDstMVF`, `pDstMVB`
    or
  - At least one of following cases is true: *pBitOffset exceeds [0,7], TRB <= 0, TRD <= 0, or transparent status buffer is not 32-bit aligned.
- `ippStsErr` – status error
  - Meet illegal code in bit stream which can not be looked up in VLC table

DecodeMV_BVOP_DirectSkip_MPEG4

Prototype

```
IppStatus ippiDecodeMV_BVOP_DirectSkip_MPEG4 (const IppMotionVector *
   pSrcMV, IppMotionVector * pDstMVF, IppMotionVector * pDstMVB, Ipp8u *
   pTranspSrcMB, int TRB, int TRD);
```

Description

Decodes motion vector(s) of the macroblock in B-VOP using direct mode when the current macroblock is skipped. A macroblock in B-VOP is skipped if modb == '1'.

Description

- `pSrcMV` – pointer to the motion vector buffer of the co-located macroblock in the most recently decoded I- or P-VOP
- `pTranspSrcMB` – pointer to the transparent status buffer of the co-located macroblock
- TRB – the difference in temporal reference of the B-VOP and the previous reference VOP
- TRD – the difference in temporal reference of the temporally next reference VOP with temporally previous reference VOP

Output Arguments

- `pDstMVF` – pointer to the forward motion vector buffer of the current macroblock which contains decoded forward motion vector
- `pDstMVB` – pointer to the backward motion vector buffer of the current macroblock which contains decoded backward motion vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrcMV`, `pTranspSrcMB` , `pDstMVF`, `pDstMVB`
    or
  - At least one of below cases is true: TRB <= 0, TRD <= 0, or transparent status buffer is not 32-bit aligned.

LimitMVToRect_MPEG4

Prototype

```
IppStatus ippiLimitMVToRect_MPEG4 (const IppMotionVector * pSrcMV,
   IppMotionVector *pDstMV, IppiRect * pRectVOPRef, int Xcoord, int
   Ycoord, int size);
```

Description

Limit the motion vector of current block/macroblock into the expanded bounding rectangle.

Input Arguments

- `pSrcMV` – pointer to the motion vector of current block or macroblock
- `pRectVOPRef` – pointer to the bounding rectangle
- `Xcoord`, `Ycoord` – the coordinates of the current block or macroblock
- `size` – the size of block or macroblock

Output Arguments

`pDstMV` – pointer to the limited motion vector

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the following pointers is NULL: `pSrcMV, pDstMV, or pRectVOPRef.` or
  - — At least one of following case is true: size is neither BLOCK_SIZE nor MB_SIZE; the width (or height) of rectangle is less than twice of size.

PredictReconCoefIntra_MPEG4_16s

Prototype

```
IppStatus ippiPredictReconCoefIntra_MPEG4_16s (Ipp16s * pSrcDst, Ipp16s
   * pPredBufRow, Ipp16s * pPredBufCol, int curQP, int predQP, int
   predDir, int ACPredFlag, IppVideoComponent videoComp);
```

Description

Performs adaptive DC/AC coefficient prediction for an intra block. Prior to the function call, prediction direction (`predDir`) should be selected as specified in subclause 7.4.3.1 of ISO/IEC 14496-2.

Description

- `pSrcDst` – pointer to the coefficient buffer which contains the quantized coefficient residuals (PQF) of the current block
- `pPredBufRow` – pointer to the coefficient row buffer
- `pPredBufCol` – pointer to the coefficient column buffer
- `curQP` – quantization parameter of the current block. `curQP` may equal to `predQP` especially when the current block and the predictor block are in the same macroblock.
- `predQP` – quantization parameter of the predictor block
- `predDir` – indicates the prediction direction which takes one of the following values:
  - IPP_VIDEO_HORIZONTAL – predict horizontally
  - IPP_VIDEO_VERTICAL – predict vertically
- `ACPredFlag` – a flag indicating if AC prediction should be performed. It is equal to ac_pred_flag in the bit stream syntax of MPEG-4
- `videoComp` – video component type (luminance, chrominance or alpha) of the current block

Output Arguments

- `pSrcDst` – pointer to the coefficient buffer which contains the quantized coefficients (QF) of the current block
- `pPredBufRow` – pointer to the updated coefficient row buffer
- `pPredBufCol` – pointer to the updated coefficient column buffer

NOTE. *Buffer update: Update the AC prediction buffer (both row and column buffer).*

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers is NULL: `pSrcDst`, `pPredBufRow`, or `pPredBufCol`.
    or
    At least one the following cases: curQP <= 0, predQP <= 0, preDir exceeds [1,2].
    or
  - At least one of the pointers `pSrcDst`, `pPredBufRow`, or `pPredBufCol` is not 32-bit aligned.

PadCurrent_16x16_MPEG4_8u_I
PadCurrent_8x8_MPEG4_8u_I

Prototype

```
IppStatus ippiPadCurrent_16x16_MPEG4_8u_I (const Ipp8u * pSrcBAB, int
   stepBinary, Ipp8u * pSrcDst, int stepTexture);
IppStatus ippiPadCurrent_8x8_MPEG4_8u_I (const Ipp8u * pSrcBAB, int
   stepTexture, Ipp8u * pSrcDst);
```

Description

Performs horizontal and vertical repetitive padding process on luminance/alpha macroblock or chrominance block. The horizontal and vertical repetitive padding processes are specified in subclause 7.6.1.1 and 7.6.1.2 of ISO/IEC 14496-2 respectively.

Input Arguments

- `pSrcDst` – pointer to the block to be padded

- `stepTexture` – width of the source texture (Luminance, Chrominance or Grayscale alpha) plane (numbered with pixel)
- `stepBinary` – width of the source binary alpha plane (for 16X16 version) or source binary alpha buffer (for 8X8 version) (numbered with byte)
- `pSrcBAB` – pointer to the binary alpha plane (for 16X16 version) or binary alpha block buffer (for 8X8 version). In 8X8 version, the buffer contains 32 bytes (256 bits) for 16 by 16 luminance or alpha block, or 8 bytes (64 bits) for 8 by 8 chrominance block.

NOTE. *For chrominance components, the BAB is generated by subsampling the shape block of the corresponding luminance component.*

Output Arguments

`pSrcDst` – pointer to the padded block

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the pointers is NULL: `pSrcDst` or `pSrcBAB`.
    or
  - — In 16 by 16 case, at least one of below case: stepTexture < 16, stepBinary < 2, stepTexture is not 4 multiple.
    or
  - — In 8 by 8 case, at least one of below case: stepTexture < 8, stepTexture is not 4 multiple.
    or
  - — `pSrcDst` is not 32-bit aligned.
    or
  - — All the elements of current BAB are zero.

PadMBHorizontal_MPEG4_8u

Prototype

```
IppStatus ippiPadMBHorizontal_MPEG4_8u (const Ipp8u * pSrcY, const Ipp8u
   * pSrcCb, const Ipp8u * pSrcCr, const Ipp8u * pSrcA, Ipp8u * pDstY,
   Ipp8u * pDstCb, Ipp8u * pDstCr, Ipp8u * pDstA, int stepYA, int
   stepCbCr);
```

Description

Performs horizontal extended padding process on exterior macroblock, which includes luminance, chrominance and alpha (if available) blocks, immediately next to boundary macroblock.

NOTE. *The MB version pads all blocks of luminance, chrominance and alpha (if exists) in one MB, while 16x16 version could be used to pad only 4 luminance or alpha blocks, and 8x8 version to pad one chrominance (Cb or Cr) block.*

Input Arguments

- `stepYA` – width of the luminance or alpha planes. (numbered with pixel)
- `stepCbCr` – width of the Chrominance planes. (numbered with pixel)
- `pSrcY` – pointer to one of the vertical border of the boundary luminance blocks that are chosen to pad the exterior macroblock
- `pSrcCb` –pointer to one of the vertical border of the boundary Cb block that is chosen to pad the exterior macroblock
- `pSrcCr` – pointer to one of the vertical border of the boundary Cr block that is chosen to pad the exterior macroblock
- `pSrcA` – pointer to one of the horizontal border of the boundary alpha blocks that are chosen to pad the exterior macroblock. If `pSrcA` equals to NULL, then no alpha plane is available. Otherwise, the alpha plane should be padded.

Output Arguments

- `pDstY` – pointer to the padded exterior luminance blocks

- pDstCb – pointer to the padded exterior Cb block
- pDstCr – pointer to the padded exterior Cr block
- pDstA – pointer to the padded exterior alpha blocks

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - — At least one of the following pointers is NULL: pSrcY, pSrcCb, pSrcCr, pDstY, pDstCb, pDstCr.
    or
  - — If pSrcA != NULL, pDstA = NULL or pDstA is not 32-bit aligned.
    or
  - — At least one of pDstY, pDstCb, or pDstCr is not 32-bit aligned.
    or
  - — At least one of the following conditions is true:
    – stepYA < 16
    – stepCbCr < 8
    – stepYA or stepCbCr is not a multiple of 4

PadMBVertical_MPEG4_8u

Prototype

```
IppStatus ippiPadMBVertical_MPEG4_8u (const Ipp8u * pSrcY, const Ipp8u *
   pSrcCb, const Ipp8u * pSrcCr, const Ipp8u * pSrcA, Ipp8u * pDstY,
   Ipp8u * pDstCb, Ipp8u * pDstCr, Ipp8u * pDstA, int stepYA, int
   stepCbCr);
```

Description

Performs vertical extended padding process on exterior macroblock, which includes luminance, chrominance and alpha (if available) blocks, immediately next to boundary macroblock.

NOTE. *The MB version pads all blocks of luminance, chrominance and alpha (if exist) in one MB, while 16X16 version could be used to pad only 4 luminance or alpha blocks, and 8X8 version to pad one chrominance (Cb or Cr) block.*

Input Arguments

- `stepYA` – width of the Luminance and/or alpha planes
- `stepCbCr` – width of the Chrominance planes
- `pSrcY` – pointer to one of the horizontal border of the boundary luminance blocks that are chosen to pad the exterior macroblock
- `pSrcCb` – pointer to one of the horizontal border of the boundary Cb block that is chosen to pad the exterior macroblock
- `pSrcCr` – pointer to one of the horizontal border of the boundary Cr block that is chosen to pad the exterior macroblock
- `pSrcA` – pointer to one of the horizontal border of the boundary alpha blocks that are chosen to pad the exterior macroblock. If `pSrcA` equals to NULL, then no alpha plane is available. Otherwise, the alpha plane should be padded in MB version.

Output Arguments

- `pDstY` – pointer to the padded exterior luminance blocks
- `pDstCb` – pointer to the padded exterior Cb block
- `pDstCr` – pointer to the padded exterior Cr block
- `pDstA` – pointer to the padded exterior alpha blocks

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL:
    `pSrcY`
    `pSrcCb`
    `pSrcCr`
    `pDstY`
    `pDstCb`
    `pDstCr`

— At least one of below case: stepYA < 16, stepCbCr < 8, stepYA or stepCbCr is not 4 multiple.
— At least one of the following is not 32-bit aligned:
`pSrcY`
`pSrcCb`
`pSrcCr`
`pDstY`
`pDstCb`
`pDstCr`
— If `pSrcA` is not NULL, `pSrcA` is not 32-bit aligned, `pDstA` is NULL or `pDstA` is not 32-bit aligned.

PadMBGray_MPEG4_8u

Prototype

```
IppStatus ippiPadMBGray_MPEG4_8u (Ipp8u grayVal, Ipp8u * pDstY, Ipp8u *
   pDstCb, Ipp8u * pDstCr, Ipp8u * pDstA, int stepYA, int stepCbCr);
```

Description

Fills gray value in exterior macroblock (includes luminance, chrominance and alpha (if available) blocks) that is not located next to any boundary macroblock.

NOTE. *The MB version pads all blocks of luminance, chrominance and alpha (if exist) in one MB, while 16X16 version could be used to pad only 4 luminance or alpha blocks, and 8X8 version to pad one chrominance (Cb or Cr) block.*

Input Arguments

- gravely – the gray value to fill the exterior macroblock/block. It should be set to $2^{bits\_per\_pixel - 1}$, where bits_per_pixel = 8 here.
- `stepYA` – width of the Luminance and/or alpha planes.(numbered with pixel)

- `stepCbCr` – width of the Chrominance planes.

Output Arguments

- `pDstY` – pointer to the padded exterior luminance blocks. `pDstY` should be 32-bit aligned.
- `pDstCb` – pointer to the padded exterior Kb block. `DstCb` should be 32-bit aligned.
- `pDstCr` – pointer to the padded exterior Cr block. `pDstCr` should be 32-bit aligned.
- `pDstA` – pointer to the padded exterior alpha blocks. If `pDstA` equals to NULL, then no alpha plane is available. Otherwise, the alpha plane should be padded in MB version. `pDstA` should be 32-bit aligned.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the following pointers is NULL: `pDstY`, `pDstCb`, `pDstCr`.
    or
  - — At least one of below case: stepYA < 16, stepCbCr < 8, grayvalue <= 0, stepYA or stepCbCr is not a multiple of 4.
    or
  - — At least one of pDstY, pDstCb, pDstCr not 32-bit aligned, If pDstA != NULL, pDstA not 32-bit aligned.

PadMV_MPEG4

Prototype

```
IppStatus ippiPadMV_MPEG4 (IppMotionVector * pSrcDstMV, Ipp8u * pTransp);
```

Description

Performs motion vector padding for a macroblock.

Input Arguments

- `pSrcDstMV` – pointer to motion vector buffer of the current macroblock
- `pTransp` – pointer to transparent status buffer of the current macroblock

Output Arguments

`pSrcDstMV` – pointer to motion vector buffer in which the motion vectors have been padded

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – Bad arguments
  - — At least one of the following pointers NULL: `pSrcDstMV`, `pTransp`
    or
  - — Transparent status or motion vector buffer is not 32-bit aligned.

DecodeVLCZigzag_IntraDCVLC_MPEG4_1u16s DecodeVLCZigzag_IntraACVLC_MPEG4_1u16s

Prototype

```
IppStatus ippiDecodeVLCZigzag_IntraDCVLC_MPEG4_1u16s(Ipp8u **
   ppBitStream, int * pBitOffset, Ipp16s * pDst, int predDir,
   IppVideoComponent videoComp);
IppStatus ippiDecodeVLCZigzag_IntraACVLC_MPEG4_1u16s(Ipp8u **
   ppBitStream, int * pBitOffset, Ipp16s * pDst, int predDir);
```

Description

Performs VLC decoding and inverse zigzag scan for one intra coded block.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bitstream buffer
- `pBitOffset` – pointer to the bit position in the byte pointed to by *ppBitStream. *pBitOffset is valid within [0-7].

  Bit Position in one byte: |Most Least|
  *pBitOffset |0 1 2 3 4 5 6 7|

- `predDir` – AC prediction direction which is used to decide the zigzag scan pattern. It takes one of the following values:

IPP_VIDEO_NONE	AC prediction not used; perform classical zigzag scan;
IPP_VIDEO_HORIZONTAL	Horizontal prediction; perform alternate-vertical zigzag scan;
IPP_VIDEO_VERTICAL	Vertical prediction; thus perform alternate-horizontal zigzag scan.

- `videoComp` – video component type (luminance, chrominance or alpha) of the current block

Output Arguments

- `ppBitStream` – *ppBitStream is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- `pBitOffset` – *pBitOffset is updated so that it points to the current bit position in the byte pointed by *ppBitStream
- `pDst` – pointer to the coefficient buffer of current block. Should be 32-bit aligned

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers is NULL: ppBitStream, *ppBitStream, pBitOffset, pDst.
    or
  - At least one of below case: *pBitOffset exceeds [0,7], preDir exceeds [0,2].
    or
  - pDst is not 32-bit aligned
- `ippStsErr`
  - In ippiDecodeVLCZigzag_IntraDCVLC_MPEG4_1u16s, dc_size > 12
  - At least one of mark bit equals to zero
  - Meet illegal stream which can not be looked up in VLC table
  - Meet forbidden code in VLC FLC table
  - If the number of coefficients beyond 64

NOTE. *The IntraDCVLC version uses Intra DC VLC solution to decode Intra DC coefficients, while the IntraACVLC version uses Intra AC VLC solution to decode Intra DC coefficient.*

DecodeVLCZigzag_Inter_MPEG4_1u16s

Prototype

```
IppStatus ippiDecodeVLCZigzag_Inter_MPEG4_1u16s(Ipp8u ** ppBitStream,
   int * pBitOffset, Ipp16s * pDst);
```

Description

Performs VLC decoding and inverse zigzag scan for one inter coded block.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream buffer
- `pBitOffset` – pointer to the bit position in the byte pointed to by *ppBitStream. *pBitOffset is valid within [0-7].

Output Arguments

- `ppBitStream` – *ppBitStream is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- `pBitOffset` – *pBitOffset is updated so that it points to the current bit position in the byte pointed by *ppBitStream
- `pDst` – pointer to the coefficient buffer of current block. Should be 32-bit aligned.

Returns

- `ippStsBadArgErr` – bad arguments
  - — At least one of the following pointers is NULL: ppBitStream, *ppBitStream, pBitOffset, pDst
    or
  - — pDst is not 32-bit aligned
    or
  - — *pBitOffset exceeds [0,7].
- `ippStsErr` – status error
  - — At least one of mark bit equals to zero
  - — Meet illegal stream which can not be looked up in VLC table
  - — Meet forbidden code in VLC FLC table

— If the number of coefficients beyond 64

QuantInvIntra_MPEG4_16s_I
QuantInvInter_MPEG4_16s_I

Prototype

```
IppStatus ippiQuantInvIntra_MPEG4_16s_I(Ipp16s * pSrcDst, int QP, const
   Ipp8u * pQMatrix, IppVideoComponent videoComp);
IppStatus ippiQuantInvInter_MPEG4_16s_I(Ipp16s * pSrcDst, int QP, const
   Ipp8u * pQMatrix);
```

Description

Performs inverse quantization on intra/inter coded block.

This function supports bits_per_pixel = 8. Mismatch control is performed for the first MPEG-4 mode inverse quantization method.

- The output coefficients are clipped to the range: [-2048, 2047].
- Mismatch control is performed for the first inverse quantization method.

Input Arguments

- `pSrcDst` – pointer to the input (quantized) intra/inter block
- `QP` – quantization parameter (quantiser_scale)
- `pQMatrix` – pointer to quantization weighting matrix. If pQMatrix is NULL, this function will use the H.263 mode inverse quantization method; otherwise, it will use the MPEG-4 mode method.
- `videoComp` – (Intra version only.) Video component type of the current block. Takes one of the following flags: IPP_VIDEO_LUMINANCE, IPP_VIDEO_CHROMINANCE, IPP_VIDEO_ALPHA.

Output Arguments

- `pSrcDst` – pointer to the output (dequantized) intra/inter block

Returns

- `ippStsNoErr` – no error

- `ippStsBadArgErr` – bad arguments
  - — If pSrcDst is NULL.
    or
  - — If QP <= 0.
    or
  - — videoComp is none of IPP_VIDEO_LUMINANCE, IPP_VIDEO_CHROMINANCE and IPP_VIDEO_ALPHA.

DecodeBlockCoef_Intra_MPEG4_1u8u

Prototype

```
IppStatus ippiDecodeBlockCoef_Intra_MPEG4_1u8u (Ipp8u ** ppBitStream,
   int *pBitOffset, Ipp8u *pDst, int step, Ipp16s *pCoefBufRow, Ipp16s
   *pCoefBufCol, Ipp8u curQP, Ipp8u *pQPBuf, const Ipp8u *pQMatrix, int
   blockIndex, int intraDCVLC, int ACPredFlag);
```

Description

Decodes the INTRA block coefficients. Inverse quantization, inversely zigzag positioning, and IDCT, with appropriate clipping on each step, are performed on the coefficients. The results are then placed in the output frame/plane on a pixel basis.

For INTRA block, the output values are clipped to [0, 255] and written to corresponding block buffer within the destination plane.

NOTE. *This function will be used only when at least one non-zero AC coefficient of current block exists in the bit stream. DC only condition will be handled in another function.*

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream buffer. There is no boundary check for the bit stream buffer.

- `pBitOffset` – pointer to the bit position in the byte pointed to by `*ppBitStream`. `*pBitOffset` is valid within [0-7].
- `step` – width of the destination plane
- `pCoefBufRow` – pointer to the coefficient row buffer
- `pCoefBufCol` – pointer to the coefficient column buffer
- `curQP` – quantization parameter of the macroblock which the current block belongs to
- `pQPBuf` – pointer to the quantization parameter buffer
- `pQMatrix` – pointer to quantization weighting matrix. If pQMatrix is NULL, this function will use the H.263 mode inverse quantization method; otherwise, it will use the MPEG-4 mode method.
- `blockIndex` – block index indicating the component type and position as defined in subclause 6.1.3.8, Figure 6-5 of ISO/IEC 14496-2. Furthermore, index 6 to 9 indicate the alpha blocks spatially corresponding to luminance block 0 to 3 in the same macroblock.
- `intraDCVLC` – a code determined by intra_dc_vlc_thr and QP. This allows a mechanism to switch between two VLC for coding of Intra DC coefficients as per Table 6-21 of ISO/IEC 14496-2. If the current block is a alpha block, the parameter "intraDCVLC" should not be zero.
- `ACPredFlag` – a flag equal to ac_pred_flag (of luminance) or ac_pred_flag_alpha (of alpha block) indicating if the ac coefficients of the first row or first column are differentially coded for intra coded macroblock.

Output Arguments

- `ppBitStream` – `*ppBitStream` is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- `pBitOffset` – `*pBitOffset` is updated so that it points to the current bit position in the byte pointed by `*ppBitStream`
- `pDst` – pointer to the block in the destination plane. `pDst` should be 64-bit aligned.
- `pCoefBufRow` – pointer to the updated coefficient row buffer.

NOTE. *The coefficient buffers should be updated according to the predefined structure. See* "Buffers".

- `pCoefBufCol` – pointer to the updated coefficient column buffer

NOTE. *The accuracy of the output values is "1" relative to the reference code.*

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the following pointers is NULL: `ppBitStream`, `*ppBitStream`, `pBitOffset`, `pCoefBufRow`, `pCoefBufCol`, `pQPBuf`, `pDst`.
    or
  - — At least one of the below case: `*pBitOffset` exceeds [0,7], curQP exceeds (1, 31), `blockIndex` exceeds [0,9], step is not the multiple of 8, `intraDCVLC` is zero while `blockIndex` greater than 5.
    or
  - — `pDst` is not 64-bit aligned
- `ippStsErr` – status error

Refer to "DecodeVLCZigzag_Inter_MPEG4_1u16s".

DecodeBlockCoef_Inter_MPEG4_1u16s

Prototype

```
IppStatus ippiDecodeBlockCoef_Inter_MPEG4_1u16s (Ipp8u ** ppBitStream,
   int * pBitOffset, Ipp16s * pDst, int QP, const Ipp8u * pQMatrix);
```

Description

Decodes the INTER block coefficients. Inverse quantization, inversely zigzag positioning and IDCT, with appropriate clipping on each step, are performed on the coefficients. The results (residuals) are placed in a contiguous array of 64 elements.

For INTER block, the output buffer holds the residuals for further reconstruction.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream buffer. There is no boundary check for the bit stream buffer.
- `pBitOffset` – pointer to the bit position in the byte pointed to by `*ppBitStream`. `*pBitOffset` is valid within [0-7]
- `QP` – quantization parameter
- `pQMatrix` – pointer to quantization weighting matrix. If pQMatrix is NULL, this function will use the H.263 mode inverse quantization method; otherwise, it will use the MPEG-4 mode method.

Output Arguments

- `ppBitStream` – `*ppBitStream` is updated after the block is decoded, so that it points to the current byte in the bit stream buffer
- `pBitOffset` – `*pBitOffset` is updated so that it points to the current bit position in the byte pointed by `*ppBitStream`
- `pDst` – pointer to the decoded residual buffer (a contiguous array of 64 elements of Ipp16s data type)

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the following pointers is Null:

    `ppBitStream`
    `*ppBitStream`
    `pBitOffset`
    `pDst`
  - — At least one of the below case:

    `*pBitOffset` exceeds [0,7], QP <= 0;
  - — `pDst` not 64-bit aligned
- `ippStsErr` – status error. Refer to `ippStsErr` of "DecodeVLCZigzag_Inter_MPEG4_1u16s".

NOTE. *The accuracy of the output values is "1" relative to the reference code.*

DecodeCAEIntraH_MPEG4_1u8u
DecodeCAEIntraV_MPEG4_1u8u

Prototype

```
IppStatus ippiDecodeCAEIntraH_MPEG4_1u8u(Ipp8u ** ppBitStream, int
   *pBitOffset, Ipp8u * pBinarySrcDst, int step, int blocksize);
IppStatus ippiDecodeCAEIntraV_MPEG4_1u8u (Ipp8u ** ppBitStream, int
   *pBitOffset, Ipp8u * pBinarySrcDst, int step, int blocksize) ;
```

Description

Performs Context Arithmetic Code decoding in intra macroblock. H indicates scan type is horizontal. V indicates scan type is vertical. Convert ratio is supported in these functions.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte from which the intra block starts
- `pBitOffset` – pointer to the bit position in the byte pointed to by *ppBitStream. *pBitOffset is valid within [0-7]
- `pBinarySrcDst` – pointer to the Source-Dest Binary macroblock the left and top border should be loaded before
- `step` – width of source-dest binary plane, in bytes
- `blocksize` – macroblock size, if convert ratio take effects, it means subsampled macro block size

Output Arguments

- `ppBitStream` – pointer to the pointer to the current byte from which the intra block starts
- `pBitOffset` – pointer to the bit position in the byte pointed to by *ppBitStream. *pBitOffset is valid within [0-7]
- `pBinarySrcDst` – pointer to the Source-Dest Binary macroblock the left and top border should be loaded before

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

— At least one of the following pointers is NULL: `ppBitStream`, `*ppBitStream`, `pBitOffset`, `pBinaryDst`
or
Blocksize is not 16, 8 or 4.
or
— *pBitOffset exceeds [0,7]

DecodeCAEInterH_MPEG4_1u8u
DecodeCAEInterV_MPEG4_1u8u

Prototype

```
IppStatus ippiDecodeCAEInterH_MPEG4_1u8u (Ipp8u ** ppBitStream, int *
   pBitOffset, const Ipp8u * pBinarySrcPred, int offsetPred, Ipp8u *
   pBinarySrcDst, int step, int blocksize);
IppStatus ippiDecodeCAEInterV_MPEG4_1u8u (Ipp8u ** ppBitStream, int *
   pBitOffset, const Ipp8u * pBinarySrcPred, int offsetPred, Ipp8u *
   pBinarySrcDst, int step, int blocksize);
```

Description

Performs Context Arithmetic Code decoding in inter macroblock. H indicates scan type is horizontal. V indicates scan type is vertical. Convert ratio is supported in these functions.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte from which the intra block starts
- `pBitOffset` – pointer to the bit position in the byte pointed to by `*ppBitStream`. `*pBitOffset` is valid within [0-7]
- `pBinarySrcPred` – pointer to the related macroblock in the reference binary plane, the left and top border should be loaded before. But pointer points to top-left corner of this macro block, not the extended zone.
- `pBinarySrcDst` – pointer to the Source-Dest Binary macroblock the left and top border should be loaded before
- `offsetPred` – the bit position of first pixel in reference macroblock, valid within Bits 0 to 7
  Where:
  — MSB=zero (0)

— LSB=seven (7)

Bit Position in one byte: |Most Least|
*pBitOffset |0 1 2 3 4 5 6 7|

- `step` – width of source-dest binary plane and reference plane, in byte. If blocksize not equals to 16, it indicates binary buffer step.
- `blocksize` – macroblock size, if convert ratio take effects, it means subsampled macro block size

NOTE. *Reference Binary plane and current Binary plane have same step. If blocksize does not equal 16 (convert ratio take effects), then step indicates both reference binary buffer and current binary buffer.*

Output Arguments

- `ppBitStream` – pointer to the pointer to the current byte from which the intra block starts
- `pBitOffset` – pointer to the bit position in the byte pointed to by `*ppBitStream`. `*pBitOffset` is valid within [0-7]
- `pBinarySrcDst` – pointer to the Source-Dest Binary macroblock the left and top border should be loaded before

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the following pointers is NULL: `ppBitStream`, `*ppBitStream`, `pBitOffset`, `pBinaryDst`.
    or
  - — *`pBitOffset` exceeds [0,7] or offsetPred exceeds [0,7]
    or
  - — `step` < 2
    or
  - — `Blocksize` is not 16, 8 or 4

DecodeMVS_MPEG4

Prototype

```
IppStatus,ippiDecodeMVS_MPEG4 (Ipp8u **ppBitStream, int *pBitOffset,
   IppMotionVector * pSrcDstMVS const Ipp8u * pSrcBABMode, int
   stepBABMode, const IppMotionVector * pSrcMVLeftMB, const
   IppMotionVector * pSrcMVUpperMB,const IppMotionVector *
   pSrcMVUpperRightMB, const Ipp8u * pTranspLeftMB,const Ipp8u *
   pTranspUpperMB, const Ipp8u * pTranspUpperRightMB, int predFlag)
```

Description

Decode MVs (Motion Vector of shape) according to the spec.

Input Arguments

- `ppBitStream` – Pointer to the pointer to the current byte in the bit stream buffer.
- `PBitOffset` – Pointer to the bit position in the byte pointed by *ppBitStream. Valid within 0 to 7.
- `pSrcDstMVs` – Pointer to the shape motion vector buffer of the current BAB.
- `pSrcBABMode` – Pointer to the BAB mode buffer of current BAB, which stored in the BAB mode plane.
- `stepBMBMode` – The width of the BAB mode plane.
- `pSrcMVLeftMB, pSrcMVUpperMB, pSrcMVUpperRightMB` – Pointers to the motion vector buffers of the macroblocks spacially at the left, upper and upper-right side of the current macroblock respectively.
- `pTranspLeftMB, pTranspUpperMB, pTranspUpperRightMB, pTranspCurMB` – Pointers to the transparent status buffers of the macroblocks, spacially at the left, upper, and upper-right side of, and the current macroblock respectively.
- `PredFlag` – The flag will be set zero, while the current VOP is BVOP or the current VOL is shape only mode; else, the flag is nonzero.

Output Arguments

- PSrcDstMVS – Pointer to the decoded motion vector of shape.

Returns

- IPP status code
- `ippStsBadArgErr` – bad arguments
  - — At least one of the pointers: *ppBitStream, ppBitStream, pBitOffset, pSrcDstMVS, SrcBABMod, pSrcMVLeftMB, pSrcMVUpperMB, pSrcMVUpperRightMB, pTranspLeftMB, pTranspUpperMB, pTranspUpperRightMB is NULL.
  - — BAD_ARGUMENT_DEFINITION:
    While stepBMBMode <= 0, or *pBitOffset exceeds [0,7].

NOTE. *Only when the BAB_TYPE is not equal to Transparent, Opaque and IntraCAE, the MVs need to be decoded.*

PadMBOpaque_MPEG4_8u_P4R

Prototype

```
IppStatus ippiPadMBOpaque_MPEG4_8u_P4R  (const Ipp32u *
pSrcTrasptMBLeft, Ipp8u * pSrcDstCurrY, Ipp8u * pSrcDstCurrCb, Ipp8u *
pSrcDstCurrCr, Ipp8u * pSrcDstCurrA, Ipp8u * pSrcDstPadded, int iMBX, int
iMBY, int stepYA, int stepCbCr);
```

Description

Performs general padding as current macroblock is opaque. What this function do is padding it's necessary neighboring macroblocks.

Input Arguments

- `pSrcDstCurrY` – pointer to the top-left of current luminance (Y) block in current VOP.
- `pSrcDstCurrCb` – pointer to the top-left of current chrominance (Cb) block in current VOP.
- `pSrcDstCurrCr` – pointer to the top-left of current chrominance (Cr) block in current VOP.
- `pSrcDstCurrA` – pointer to the top-left of current alpha (A) block in current VOP.
- `pSrcTrasptMBLeft` – pointer to left macroblock's transparent buffer.

- `pSrcDstPadded` – pointer to padded buffer, which indicates the related macroblock is padded or not. It make sense when related macroblock is transparent one.
- `iMBX` – current macroblock's X direction index in VOP, the start one is 0.
- `iMBY` – current macroblock's Y direction index in VOP, the start one is 0.
- `stepYA` – width of luminance/alpha plane in byte.
- `stepCbCr` – width of chronminance plane in byte.

Output Arguments

`pSrcDstCurrY` – pointer to the top-left of current luminance (Y) blocks.

`pSrcDstCurrCb` – pointer to the top-left of current chrominance (Cb) block.

`pSrcDstCurrCr` – pointer to the top-left of current chrominance (Cr) block.

`pSrcDstCurrA` – pointer to the top-left of current alpha (A) block in current VOP.

`pSrcDstPadded` – pointer to padded buffer, which indicates the related macroblock is padded or not.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the pointers: pSrcDstCurrY, pSrcDstCurrCb, pSrcDstCurrCr, pTrasptMBLeft, pTrasptMBTop, pPadded is NULL.
  - — While stepYA < 16 or stepCbCr < 8, or stepYA or stepCbCr is not 4 multiple.
  - — "iMBX < 0 or iMBY < 0;
  - — While anyone of pSrcDstCurrY, pSrcDstCurrCb, pSrcDstCurrCr, pSrcDstCurrA (if make sense) is not 32-bit align.

NOTE. *One line of continuous transparent status buffer needed, pTrasptMBLeft points to the left neighboring macroblock in that buffer. *(pTrasptMBLeft +1) stores the top macroblock's transparent status.*

NOTE. *One line of continuous padded status buffer needed, pSrcDstPadded points to the current macroblock's padded status, *(pSrcDstPadded -1) stores left macroblock's padded status.*

PadMBTransparent_MPEG4_8u_P4R

Prototype

```
IppStatus ippiPadMBTransparent_MPEG4_8u_P4R (const Ipp32u *
pSrcTrasptMBLeft,  Ipp8u * pSrcDstCurrY, Ipp8u * pSrcDstCurrCb, Ipp8u *
pSrcDstCurrCr, Ipp8u * pSrcDstCurrA, Ipp8u * pSrcDstPadded, Ipp8u
grayVal, int iMBX, int iMBY, int iMBXLimit, int iMBYLimit,  int stepYA,
int stepCbCr);
```

Description

Performs general padding as current macroblock is transparent. What this function do is not only padding itself, but also padding its neighboring macroblocks if necessary.

Input Arguments

- `pSrcTrasptMBLeft` – pointer to left macroblock's transparent buffer.
- `pSrcDstCurrY` – pointer to the top-left of current luminance (Y) blocks.
- `pSrcDstCurrCb` – pointer to the top-left of current chrominance (Cb) block.
- `pSrcDstCurrCr` – pointer to the top-left of current chrominance (Cr) block.
- `pSrcDstCurrA` – pointer to the top-left of current alpha (A) block in current VOP.
- `pSrcTrasptMBLeft` – pointer to transparent buffer of the left neighboring macroblock.
- `pSrcDstPadded` – pointer to padded buffer, which indicates the related macroblock is padded or not.
- `grayVal` – he gray value to fill the exterior macroblock/block. It should be set to 2bits_per_pixel - 1, where bits_per_pixel = 8 here.
- `iMBX` – current macroblock's X direction index in VOP
- `iMBY` – current macroblock's Y direction index in VOP
- `iMBXLimit` – the number of macroblock in current VOP's X direction.
- `iMBYLimit` – the number of macroblock in current VOP's Y direction.
- `stepYA` – width of luminance/alpha plane in byte.
- `stepCbCr` – width of chronminance plane in byte.
- `stepBinary` – width of binary plane in byte.

Output Arguments

- `pSrcDstCurrY` – pointer to the top-left of current luminance (Y) blocks.
- `PSrcDstCurrCb` – pointer to the top-left of current chrominance (Cb) block.
- `PSrcDstCurrCr` – pointer to the top-left of current chrominance (Cr) block.
- `pSrcDstCurrA` – pointer to the top-left of current alpha (A) block in current VOP.
- `PSrcDstPadded` – pointer to padded buffer, which indicates the related macroblock is padded or not.

Returns

IPP status code

BAD_ARGUMENT_DEFINITION:

— At least one of the pointers: pSrcDstCurrY, pSrcDstCurrCb, pSrcDstCurrCr, pSrcTrasptMBLeft, pSrcDstPadded is NULL.

— While stepYA < 16, stepCbCr < 8 or stepBinary < 2. stepYA or stepCbCr is not 4 multiple.

— iMBX < 0, iMBY < 0, iMBX >= iMBXLimit or iMBY >= iMBYLimit;

NOTE. *At least one of the pointers: pSrcDstCurrY, pSrcDstCurrCb, pSrcDstCurrCr, pSrcTrasptMBLeft, pSrcDstPadded is NULL.One line of continuous padded status buffer needed, pSrcDstPadded points to the current macroblock's padded status, *(pSrcDstPadded -1) stores left macroblock's padded status.*

NOTE. *While stepYA < 16, stepCbCr < 8 or stepBinary < 2. stepYA or stepCbCr is not 4 multiple.*

NOTE. *iMBX < 0, iMBY < 0, iMBX >= iMBXLimit or iMBY >= iMBYLimit;*

PadMBPartial_MPEG4_8u_P4R

Input Arguments

```
IppStatus ippiPadMBPartial_MPEG4_8u_P4R (const Ipp8u * pSrcBAB, const
   Ipp32u * pSrcTrasptMBLeft, Ipp8u * pSrcDstCurrY, Ipp8u *
   pSrcDstCurrCb, Ipp8u * pSrcDstCurrCr, Ipp8u * pSrcDstCurrA, Ipp8u *
   pSrcDstPadded, int iMBX, int iMBY, int stepYA, int stepCbCr, int
   stepBinary);
```

Description

Performs general padding as current macroblock is partial. What this function do is not only padding itself, but also padding it's necessary neighboring macroblocks.

Input Arguments

- `pSrcBAB` – pointer to the binary alpha block in current VOP.
- `pSrcDstCurrY` – pointer to the top-left of current luminance (Y) block in current VOP.
- `pSrcDstCurrCb` – pointer to the top-left of current chrominance (Cb) block in current VOP.
- `pSrcDstCurrCr` – pointer to the top-left of current chrominance (Cr) block in current VOP.
- `pSrcDstCurrA` – pointer to the top-left of current alpha (A) block in current VOP.
- `pSrcTrasptMBLeft` – pointer to left macroblock's transparent buffer.
- `pSrcDstPadded` – pointer to padded buffer, which indicates the related macroblock is padded or not. It make sense when related macroblock is transparent one
- `iMBX` – current macroblock's X direction index in VOP, the start one is 0.
- `iMBY` – current macroblock's Y direction index in VOP, the start one is 0.
- `stepYA` – width of luminance/alpha plane in byte.
- `stepCbCr` – width of chronminance plane in byte.
- `stepBinary` – width of binary plane in byte.

Output Arguments

- `pSrcDstCurrY` – pointer to the top-left of current luminance (Y) blocks.
- `pSrcDstCurrCb` – pointer to the top-left of current chrominance (Cb) block.
- `pSrcDstCurrCr` – pointer to the top-left of current chrominance (Cr) block.
- `pSrcDstCurrA` – pointer to the top-left of current alpha (A) block in current VOP.

- `pSrcDstPadded` – pointer to padded buffer, which indicates the related macroblock is padded or not.

Returns

- IPP status code.
- BAD_ARGUMENT_DEFINITION:
  - At least one of the pointers: pSrcBAB, pSrcDstCurrY, pSrcDstCurrCb, pSrcDstCurrCr, pTrasptMBLeft, pTrasptMBTop, pPadded is NULL.
  - While stepYA < 16, stepCbCr < 8 or stepBinary < 2, or stepYA or stepCbCr is not 4 multiple.
  - iMBX < 0 or iMBY < 0;
  - While anyone of pSrcDstCurrY, pSrcDstCurrCb, pSrcDstCurrCr, pSrcDstCurrA (if make sense) is not 32-bit align.

NOTE. *One line of continuous transparent status buffer needed, pTrasptMBLeft points to the left neighboring macroblock in that buffer. *(pTrasptMBLeft +1) stores the top macroblock's transparent status.*

NOTE. *One line of continuous padded status buffer needed, pSrcDstPadded points to the current macroblock's padded status, *(pSrcDstPadded -1) stores left macroblock's padded status.*

Examples

Transparent Status Retrieving

```
Ipp8u  * pTranspBuf[TRANSP_BUF_SIZE];
... ...
Ipp8u  * pTransp = (Ipp8u *)(Ipp32u)pTranspBuf | 0x3) + 1); /* pTransp is
32-bit aligned */
Ipp32u * pTranspMB = (Ipp32u *)pTransp;/* MB transparent status pointer
*/

if ( *pTranspMB != IPP_VIDEO_TRANSPARENT ) {/* non-transparent MB */
```

```
    if ( pTransp[0] == IPP_VIDEO_TRANSPARENT ) { /* block 0 is
transparent, ... */
        /* ... ... */
    } else if ( pTransp[1] == IPP_VIDEO_TRANSPARENT ) {/* If block 1 is
transparent, ... */
        /* ... ... */
    }
}
```

MPEG-4 Video Encoder 13

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) on Intel® Personal Internet Client Architecture Processors (PCA processors) and Intel® PCA Processors with Intel® Wireless MMX™ Technology (PCA processors with MMX™), that are built to support the ISO/IEC 14496-2 MPEG-4 video encoder. MPEG-4 is a widely used coding method for video signals in various applications such as digital storage media, internet, various forms of wired or wireless communication, etc.

The Application Programming Interface (API) for the MPEG-4 encoder supports the following features:

- Progressive, non-scalable texture encoding.
- Block-based VLC encoding and zigzag scan.
- Motion vector encoding.
- Motion estimation, including modified full length search (SEA, successive elimination algorithm) and fast search (MVFAST, motion vector field adaptive search technique), including integer pixel search and half pixel search, including 16x16 search and 8x8 search.
- Block layer coefficient encoding, including intra-DC/AC prediction (for intra blocks), quantization, and DCT, with appropriate clipping on each step, also provides reconstructed data.

The remainder of this chapter provides details on the Intel® IPP MPEG-4 video encoder API. "Data Types and Structures" focuses on data types and structures used in the primitives. "MPEG-4 Encoder Primitives" describes each API definition and function.

Data Types and Structures

This section describes the data types and structures of the Intel® IPP MPEG-4 video encoder.

Video Components

Video components are defined as follows:

```
typedef enum {
    IPP_VIDEO_LUMINANCE,    /* Luminance component   */
    IPP_VIDEO_CHROMINANCE,  /* chrominance component */
    IPP_VIDEO_ALPHA         /* Alpha component       */
} IppVideoComponent;
```

Pixel Planes

The encoder's input and output is stored in pixel planes denoted by Y plane (luminance component), Cb plane and Cr plane (chrominance components).

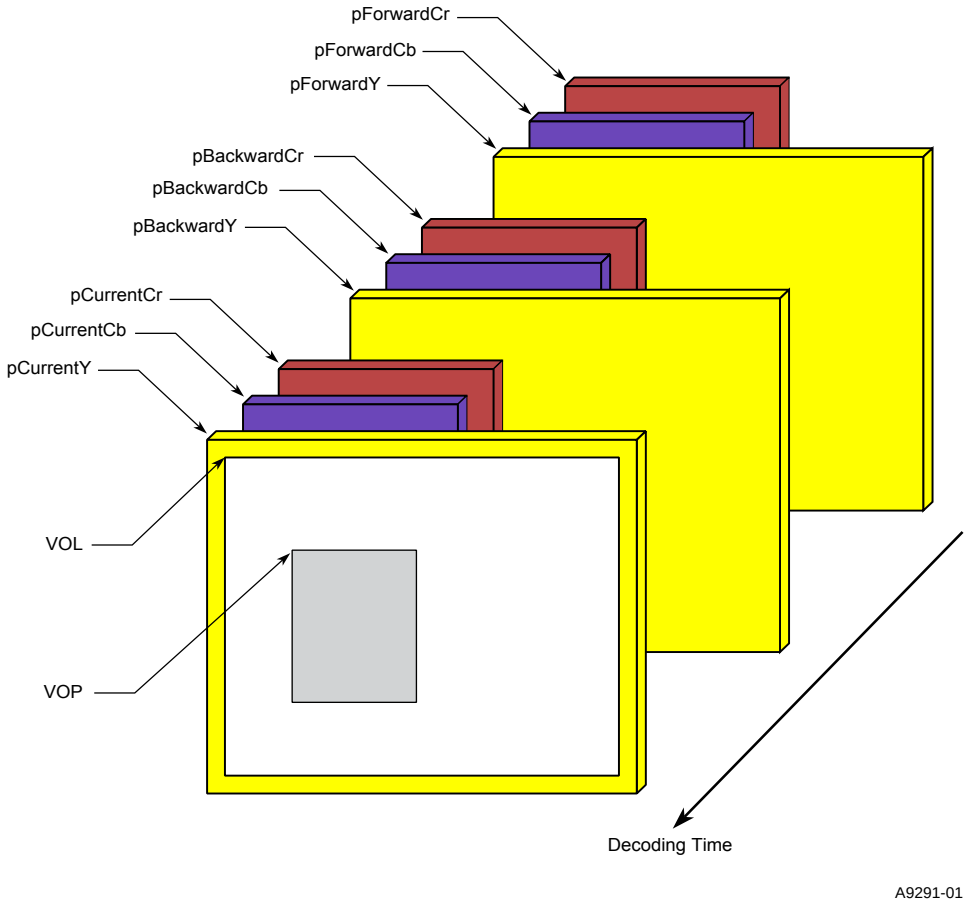
The size of Y plane relates to, but is not equal to, that of the VOL as a result of the VOP expansion. Since luminance VOP is expanded (and padded) with 16 pixels to each of the four directions, the width and height of the Y plane are 32 pixels larger than those of the VOL respectively.

The size (W, H) of Cb or Cr plane is half the size of Y plane, because chrominance VOPs are expanded with 8 pixels to each direction.

Figure 13-1 shows the relationship among pixel plane, VOL and VOP.

Allocate three sets of 32-bit word-aligned pixel planes, each consisting of a Y plane, a Cb plane and a Cr plane. These pixel plane sets are referred to as current and forward.and backward. BVOP is not supported by the encoder.

Figure 13-1 Pixel Plane, VOL, and VOP



Macroblock Types

Macroblock types in I-, P- and B-VOP are defined as follows:

```
typedef enum {
      IPP_VIDEO_INTER             = 0,    /* P picture or P-VOP */
      IPP_VIDEO_INTER_Q           = 1,    /* P picture or P-VOP */
      IPP_ VIDEO_INTER4V          = 2,    /* P picture or P-VOP */
      IPP_ VIDEO_INTRA            = 3,    /* I and P picture, I- and P-VOP */
      IPP_ VIDEO_INTRA_Q          = 4,    /* I and P picture, I- and P-VOP */
      IPP_ VIDEO_INTER4V_Q        = 5,    /* P picture or P-VOP (H.263)*/
      IPP_ VIDEO_DIRECT           = 6,    /* B picture or B-VOP (MPEG-4 only) */
      IPP_ VIDEO_INTERPOLATE      = 7,    /* B picture or B-VOP */
      IPP_ VIDEO_BACKWARD         = 8,    /* B picture or B-VOP */
      IPP_ VIDEO_FORWARD          = 9     /* B picture or B-VOP */
      IPP_VIDEO_NOTCODED          = 10    /* B picture or B-VOP */
}IppMacroblockType;
```

Motion Vector

Motion vector is defined as:

```
typedef struct {
      Ipp16s dx;
      Ipp16s dy;
}IppMotionVector;
```

Two kinds of motion vectors are used in the Intel® IPP MPEG-4 CODEC. One is for texture (in Q1 format) and the other is for shape (in Q0 format).

Transparent Status

Transparent status is a three-state value in one byte, or `Ipp8u`. The three possible states are defined as follows:

```
enum {
      IPP_VIDEO_TRANSPARENT= 0, /* Wholly transparent */
      IPP_VIDEO_PARTIAL  = 1,   /* Partially transparent */
      IPP_VIDEO_OPAQUE   = 2      /* Opaque */
};
```

Transparent status is block based in MPEG-4. Therefore:

- There is one buffer per macroblock (MB).

- Elements are block-based and contiguously stored.
- There is one byte (`Ipp8u`) per element for one block.
- There are four elements per MB.
- The first element (for block 0) must be 32-bit aligned. (which should be ensured by user).
- MB transparent status is determined by evaluating the value of the whole word.

Direction

Direction is used for predicting DC/AC and for zigzag scanning:
enum {

```
    IPP_VIDEO_NONE        = 0,
    IPP_VIDEO_HORIZONTAL  = 1,
    IPP_VIDEO_VERTICAL    = 2
};
```

Rectangle Plane

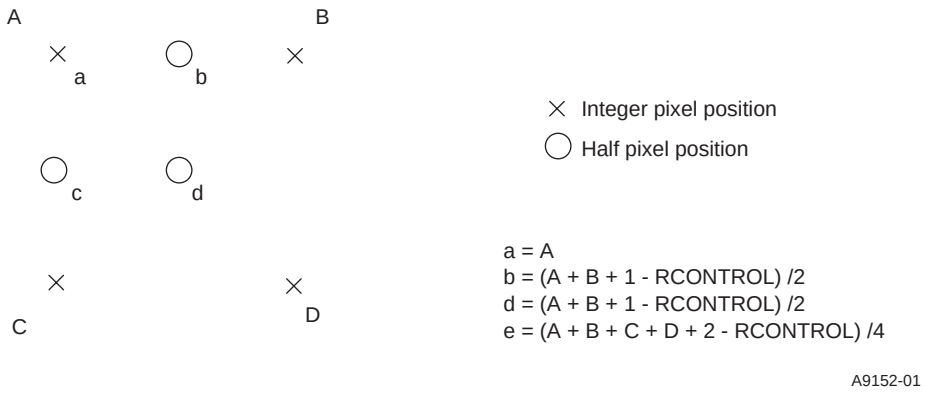
```
typedef struct {
    int x;
    int y;
    int width;
    int height;
}IppiRect;
```

Bilinear Interpolation Type

Bilinear interpolation type is used for motion estimation, compensation, and reconstruction.

```
enum {
    IPP_VIDEO_INTEGER_PIXEL   = 0,/* case "a" in Figure 13-2 */
    IPP_VIDEO_HALF_PIXEL_X    = 1,/* case "b" in Figure 13-2 */
    IPP_VIDEO_HALF_PIXEL_Y    = 2,/* case "c" in Figure 13-2 */
    IPP_VIDEO_HALF_PIXEL_XY   = 3/* case "d" in Figure 13-2 */
};
```

Figure 13-2 Halfpixed Prediction by Bilinear Interpolation



Buffers

The following sections describes buffers and their layout required by the Intel® IPP. Users must allocate and/or initialize the buffers according to these specifications.

Video Plane Buffers

Users must allocate buffers to store the raw video object (MPEG-4), consisting of texture components (Y/Cb/Cr). The presence of each component depends on the VOL shape type, as determined by `video_object_layer_shape`. Table 13-1 shows these dependencies, where "x" means required and "-" means not required.

Table 13-1 Video Plane Buffer Dependencies

VOL Shape Type	Texture	Alpha
	Y/Cb/Cr Planes	Binary Plane
Rectangular	X	–
Binary	X	X
Binary only	–	X
Grayscale	X	X

Two sets of these buffers are needed – one for the current picture/VO and the other for the previous (forward-reference) picture/VO. If bi-directional prediction (B-VOP) is supported, an additional set of buffer must be allocated.

Motion Vector for Texture (in Q1 format)

Depending on the MB type, there are from zero to eight valid motion vector(s) in a MB. A motion vector (MV) buffer contains four elements of `IppMotionVector` data. Elements are block-based and contiguously stored per each buffer. Allocate one MV buffer per MB for P-VOP. If B-VOP is supported, allocate two MV buffers for bi-directional prediction. The following vector manipulation scheme adopted by the Intel® IPP MPEG-4 CODEC, where `pMVForward` and `pMVBackward` denote the two vector buffers allocated for each MB:

- Two buffers per MB in P- or B-VOP, including `pMVForward[4]` and `pMVBackward[4]`.
- Elements are block-based, and contiguously stored per each buffer.
- In P-VOP, only `pMVForward[]` is used and valid. `pMVBackward[]` is not used.
- If MB type is "`IPP_VIDEO_INTER`"or "`IPP_VIDEO_INTER_Q` ", and if not transparent, `pMVForward[0]-[3]` must be filled with the same decoded MV.
- If MB is INTRA coded or skipped, `pMVForward[0]-[3]` must be padded with zero MVs.
- In B-VOP, `pMVForward[]` and `pMVBackward[]` may or may not be used, depending on the MB type.
- In B-VOP, if MB type is not "`IPP_VIDEO_DIRECT` ", then `pMVForward[1]-[3]` and `pMVBackward[1]-[3]` are *not* used.

Coordinates are related to the absolute coordinate system shown in Figure 7-19 of *ISO/IEC 14496-2: Information Technology - Generic Coding of Audio-Visual Objects - Part 2: Visual (FD, October 1998).*

Quantization Parameter Buffer

Allocate one "row buffer", storing the quantization parameters (QPs) for the P-VOP with one byte (Ipp8u) each element. This buffer is used for coefficient prediction. Before decoding an INTRA coded MB, the corresponding element in the QP buffer saves the QP of the MB of the upper MB row. After a MB is decoded, the corresponding element (the element storing QP of the MB above) must be updated by the QP of the current MB.

Coefficient Buffers

Allocate two coefficient buffers for Intra DC/AC prediction. One is a row buffer containing ((mb_num_per_row * 2 + 1) * 8) elements of `Ipp16s`, and the other is a column buffer containing 16 elements of `Ipp16s`.

Every eight elements of both row and column buffers, plus one element 8 units ahead in the row buffer, are used to perform DC/AC prediction for an INTRA-coded block in a MB. Each group of them contains the coefficient predictors of the neighboring block that is spatially above or to the left of the block currently to be decoded. Within every eight elements, the first element contains the DC coefficient and the other elements contain quantized AC coefficients. A negative-valued DC coefficient indicates that this neighboring block is not INTRA-coded or out of boundary, thus neither the DC nor the AC coefficients are valid.

Before decoding each VOP, initialize all DC elements in the row buffer to -1. Also, before decoding each MB row, initialize the two DC elements in the column buffer -1.

The detailed buffer layout is illustrated in Figure 13-3.

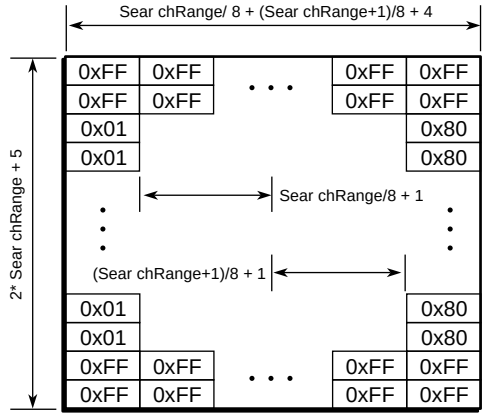
MVFAST Buffer

Allocate two buffers storing the search status for the 16*16 integer pixel search and the 8*8 integer pixel search respectively. For each buffer, there is one bit for each searching point. A bit value of 1 indicates that the pointer has been checked before or that it is out of search range. Any other value indicates that this candidate should be checked. Accordingly, the search range determines the buffer size. The buffer size is as follows:

```
Buffer size = (2*searchRange+5) * ((searchRange+1)/8+searchRange/8 + 4)
(Unit: byte)
```

NOTE. *Two buffers must be initialized before they are used. See Figure 13-3. That is, the first and last two lines should be padded with 0xFF, while the first and last column should be padded respectively with 0x01 and 0x80. In general, the searchRanges for 16*16 and 8*8 are different. The searchRange for 16*16 comes from the parameter file, while the searchRange for 8*8 is set to 2.*

Figure 13-3 Initialization of Bit Plane



MPEG-4 Encoder Primitives

This section describes all of the MPEG-4 video encoder API primitives.

BlockMatch_Integer_16x16_SEA

Prototype

```
IppStatus ippiBlockMatch_Integer_16x16_SEA (Ipp8u * pSrcRef, Ipp8u *
   pSrcCurr, Ipp16u *pSrcSumBlk, IppMotionVector *pSrcRefMV,
   IppCoordinate * pSrcPointPos, IppiRect * pSrcRefRect, int *
   pSrcDstminSAD, IppMotionVector * pDstMV, int step, int searchRange,
   int flag);
```

Description

Performs 16x16 size macro block match with Successive Elimination Algorithm (SEA), the searched motion vector have same accuracy with Exhaust Search Algorithm (ESA). This function and ippiSumNorm_VOP_MPEG4_8u16u are used in pairs typically, because the sum norm plane is one input of this function.

Input Arguments

- `pSrcRef` – pointer to the original or reconstructed reference Y plane. The pointer position is the same as the current macroblock's position in the current plane.
- `pSrcCurr` – pointer to the current original macroblock, which has been extracted from current original plane
- `pSrcSumBlk` – pointer to current macroblock in the sum plane
- `pSrcRefMV` – pointer to the predicted motion vector
- `pSrcPointPos` – pointer to the position of current macroblock in current plane.
- `pSrcRefRect` – pointer to the valid rectangular in reference plane
- `pSrcDstminSAD` – pointer to the `minSAD`, which from SAD16X16 at mv = (0,0).
- `step` – the step in reference Y-Plane
- `searchRange` – search range in 16X16 integer pixel search
- `flag` - algorithm selected flag. For implementation of PCA processors with MMX™, this parameter must be 5. For the PCA processors, the parameter flag $\in \{4, 8\}$.

Output Arguments

- `pSrcDstminSAD` – pointer to the least SAD after 16X16 integer search
- `pDstMV` – pointer to destination motion vector for the current macroblock with integer pixel definition

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - At least one of the following pointers is NULL: `pSrcRef`, `pSrcCurr`, `pSrcSumBlk`, `pSrcPointPos`, `pSrcRefRect`, `pSrcDstminSAD`, `pDstMV`.
  - At least one of the following pointers is not 64-bit aligned: `pSrcRef`, `pSrcCurr`, `pSrcSumBlk`.
  - If `pSrcRefMV` is not NULL, `*pSrcRefMV` exceeds the search range.
  - `pSrcRefRect->width < MB_SIZE` or `pSrcRefRect->height < MB_SIZE`.
  - `*pSrcPointPos` exceeds the zone indicated by `pSrcRefRect`.
  - `step < pSrcRefRect->width` or step is not a multiple of 8

— `searchRange` <= 0
— `flag` ∉ {4, 8} (for PCA processors)
— `flag` ∉ {5} (for PCA processors with MMX™)

FindMVpred_MPEG4

Prototype

```
IppStatus ippiFindMVpred_MPEG4 (IppMotionVector* pSrcMVCurMB,
   IppMotionVector* pSrcCandMV1,IppMotionVector*
   pSrcCandMV2,IppMotionVector* pSrcCandMV3,Ipp8u*
   pSrcCandTransp1,Ipp8u* pSrcCandTransp2,Ipp8u* pSrcCandTransp3, Ipp8u*
   pSrcTranspCurr, IppMotionVector* pDstMVPred, IppMotionVector*
   pDstMVPredME, int iBlk);
```

Description

Finds the vector predictor from three candidates and outputs three candidates for MVFAST, if `pDstMVPredME` is not NULL.

Input Arguments

- `pSrcMVCurMB` – pointers to the current Y macroblock buffers
- `pSrcCandMV1` – pointers to the left candidate motion vector buffers
- `pSrcCandMV2` – pointers to the top candidate motion vector buffers
- `pSrcCandMV3` – pointers to the right-top candidate motion vector buffers
- `pSrcCandTransp1`, `pSrcCandTransp2`, `pSrcCandTransp3` – pointers to the transparent status buffers of the corresponding macroblock or block
- `pSrcTranspCurr` – pointers to the transparent status buffers of the current macroblock or block
- `iBlk` – the index of block in current macroblock

Output Arguments

- `pDstMVPred` – pointer to the predicted motion vector
- `pDstMVPredME` – pointer to three motion vector candidates. This is used only to determine motion activity when MVFAST is selected.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrcMVCurMB`, `pSrcCandTransp1`, `pSrcCandTransp2`, `pSrcCandTransp3`, `pSrcTranspCurr`, `pDstMVPred`.
  - iBlk exceeds [0,3].

BlockMatch_Integer_16x16_MVFAST

Prototype

```
IppStatus ippiBlockMatch_Integer_16x16_MVFAST (Ipp8u * pSrcRef, Ipp8u *
   pSrcCurr, IppMotionVector *pSrcCanMV, IppMotionVector *pSrcRefMV,
   IppCoordinate * pSrcPointPos, IppiRect * pSrcRefRect, Ipp8u *
   pSrcSadMap, int * pFlag, int * pSrcDstSAD, IppMotionVector * pDstMV,
   int refStep, int searchRange);
```

Description

Performs 16x16 size block match with large and/or small diamond search.

Input Arguments

- `pSrcRef` – pointer to the original or reconstructed reference Y plane. The pointer position is same as current macroblock's position in current plane.
- `pSrcCurr` – pointer to the current original macroblock, which has been extracted from current original plane
- `PSrcCanMV` – pointer to the left, top and right-top reference motion vector respectively
- `pSrcRefMV` – pointer to the predicted motion vector
- `pSrcPointPos` – pointer to the position of current macroblock in current plane
- `pSrcRefRec` – pointer to the valid rectangular in reference plane
- `pSrcSadMap` – pointer to the initial address of bit plane, which is used to store the state of each search point. The buffer size for `pSrcSadMap` is (2*`SearchRange`+5)*((`SearchRange`+1)/8 + `SearchRange`/8 +4) bytes.
- `pFlag` – pointer to the flag used for SAD calculation
- `pSrcDstSAD` – pointer to the initial SAD

- refStep – the step in reference Y-Plane
- searchRange – search range in 16X16 integer pixel searchOutput Arguments.

NOTE. *For the buffer size and initialization for pSrcSadMap and pSrcBlockSadMap, see Figure 13-3 and its context.*

Output Arguments

- pSrcDstSAD – pointer to the updated SAD
- PDstMV – pointer to destination motion vector for current macroblock with integer pixel definition

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - — At least one of the following pointers is NULL: pSrcRef, pSrcCurr, pSrcrefMV, pSrcCanMV, pSrcPointPos, pSrcRefRect, pSrcSadMap, pFlag, pSrcDstSAD, pDstMV.
  - — At least one of the following parameters is not 64-bit aligned: pSrcRef, pSrcCurr, refStep.
  - — At least one of the components' absolute value (|dx| or |dy|) is greater than (2*searchRange+1): pSrcRefMV, pSrcCanMV[0], pSrcCanMV[1], pSrcCanMV[2].
  - — At least one of the following cases is true:

    pSrcRefRect->width or pSrcRefRect->height is less than or equal 16

    pSrcPointPos->x is out of range from left boarder of pSrcRefRect to its right boarder minus macroblock size.

    pSrcPointPos->y is out of range from top boarder of pSrcRefRect to its bottom boarder minus macroblock size.

    searchRange exceeds (0,1024]; *pFlag is greater than 16.

    refStep is less than or equal 16.

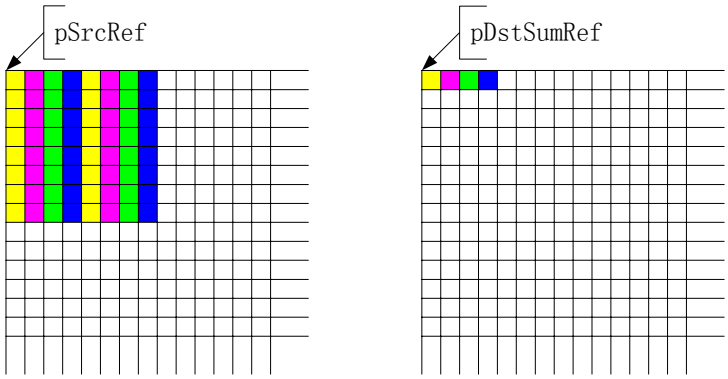
SumNorm_VOP_MPEG4_8u16u

Prototype

```
IppStatus ippiSumNorm_VOP_MPEG4_8u16u (Ipp8u * pSrcRef, IppiRect *
   pSrcRefRect, Ipp16u * pDstSumRef, int flag, int step);
```

Description

Performs summation of sub-region in a plane pointed by pSrcRef, which is a premise for motion estimation (SEA). From performance perspective, the geometry for summation is two 1x8 stripes indicated by Figure 1.1.6 1 Geometry for the calculation of the sum norm. As hinted in this figure, the result data in sum norm plane is the summation of the pixels with the same color (two 1x8 stripe) in reference plane. This summation operation will slide through the whole region specified by pSrcRefRect in reference plane. The result data is placed in sum norm plane pointed by pDstSumRef. The count of valid summation data should be (pSrcRefRect->width-4) x (pSrcRefRect ->height-7). This specific geometry for summation is very suitable for WMX instruction set implementation.



Input Arguments

- `pSrcRef` – pointer to the original or reconstructed reference Y plane, the pointer position is the top left of the padded plane
- `pSrcRefRect` – pointer to the valid rectangular in reference plane, which describes the target summation zone

- `flag` - algorithm selected flag. For implementation of PCA processors with MMX™, this parameter must be 5. For the PCA processors, the parameter flag ∈ {4, 8}.
- `step` – the step in reference Y-Plane, and reference summation plane

Output Arguments

- `pDstSumRef` – pointer to summation plane. The pointer position is the top left of the padded plane.
- Returns: ippStsNoErr – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrcRef`, `pSrcRefRect`, `pDstSumRef`
  - At least one of the following pointers is not 64-bit aligned: `pSrcRef`, `pDstSumRef`
  - `pSrcRefRect->width<=0`, `pSrcRefRect ->height<=0` or `pSrcRefRect->width` is not a multiple of 8
  - `step < pSrcRefRect->width` or step is not a multiple of 8
  - `searchRange` <= 0
  - `flag` ∉ {4, 8} (for PCA processors)
  - `flag` ∉ {5} (for PCA processors with MMX™)

QuantIntra_MPEG4_16s_I

Prototype

```
IppStatus ippiQuantIntra_MPEG4_16s_I (Ipp16s * pSrcDst, Ipp8u QP, int
   blockIndex, const int * pQMatrix);
```

Description

Performs quantization on an intra block coefficients. This function supports bits_per_pixel == 8.

Input Arguments

- `PSrcDst` – pointer to the input intra block coefficients
- QP – quantization parameter (quantiser_scale).
- `blockIndex` – block index indicating the component type and position as defined in subclause 6.1.3.8, of *ISO/IEC 14496-2*. Furthermore, indexes 6 to 9 indicate the alpha blocks spatially corresponding to luminance blocks 0 to 3 in the same macroblock.

- PQMatrix – If the second inverse quantization method is used, PQMatrix is NULL. If the first inverse quantization method is used, it points to the quantization weighting coefficients buffer (for intra MB) whose first 64 elements are the quantization weighting matrix in Q0. The second 64 elements are their reciprocals in Q21.

Output Arguments

`PSrcDst` – `pointer` to the output (quantized) intra block coefficients. The output coefficients are saturated to lie in the range:[-127,127].

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - `pSrcDst` is NULL.
  - `blockIndex` < 0 or `blockIndex` >= 10
  - QP <= 0 or QP >= 32.
  - for a non-NULL `pQMatrix, pQMatrix`[0] `* pQMatrix`[64] != (1<<21)

QuantInter_MPEG4_16s_I

Prototype

```
IppStatus ippiQuantInter_MPEG4_16s_I (Ipp16s * pSrcDst, Ipp8u QP, const
   int * pQMatrix);
```

Description

Performs quantization on an inter block coefficients. This function supports bits_per_pixel == 8.

Input Arguments

- `PSrcDst` – pointer to the input inter block coefficients
- `QP` – quantization parameter (`quantizer_scale`)
- `PQMatrix` – If the second inverse quantization method is used, PQMatrix is NULL. If the first inverse quantization method is used, it points to the quantization weighting coefficient's buffer (for inter MB) whose first 64 elements are the quantization weighting matrix in Q0. The second 64 elements are their reciprocals in Q21.

Output Arguments

`PSrcDst` – pointer to the output (quantized) inter block coefficients. The output coefficients will saturate on the interval [-127, 127].

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - `pSrcDst` is NULL.
  - `QP` <= 0 or `QP` >= 32.
  - `pQMatrix[0]*pQMatrix`[64] != (1<<21) while `pQMatrix` is not NULL

EncodeVLCZigzag_IntraDCVLC_MPEG4_16s1u
EncodeVLCZigzag_IntraACVLC_MPEG4_16s1u

Prototype

```
IppStatus ippiEncodeVLCZigzag_IntraDCVLC_MPEG4_16s1u (Ipp8u
   **ppBitStream,int *pBitOffset, Ipp16s *pQDctBlkCoef, Ipp8u
   predDir,Ipp8u pattern, IppVideoComponent videoComp);
IppStatus ippiEncodeVLCZigzag_IntraACVLC_MPEG4_16s1u (Ipp8u **
   ppBitStream,int *pBitOffset, Ipp16s * pQDctBlkCoef, Ipp8u
   predDir,Ipp8u pattern);
```

Description

Performs zigzag scanning and VLC encoding for one intra block.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream
- `pBitOffset` – pointer to the bit position in the byte pointed by *ppBitStream. Valid within 0 to 7.
- `pQDctBlkCoef` – pointer to the quantized DCT coefficient
- `predDir` – AC prediction direction, which is used to decide the zigzag scan pattern. This takes one of the following values:
  - IPP_VIDEO_NONE – AC prediction not used. Performs classical zigzag scan.

  - — IPP_VIDEO_HORIZONTAL – Horizontal prediction. Performs alternate-vertical zigzag scan.
  - — IPP_VIDEO_VERTICAL – Vertical prediction. Performs alternate-horizontal zigzag scan.
- `pattern` – block pattern which is used to decide whether this block is encoded
- `videoComp` – video component type (luminance, chrominance) of the current block

Output Arguments

- `ppBitStream` – `*ppBitStream` is updated after the block is encoded, so that it points to the current byte in the bit stream buffer.
- `pBitOffset` – `*pBitOffset` is updated so that it points to the current bit position in the byte pointed by `*ppBitStream`.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - — At least one of the following pointers is NULL: `ppBitStream`, `*ppBitStream`, `pBitOffset`, `pQDctBlkCoef`.
  - — `*pBitOffset` < 0, or `*pBitOffset` >7.
  - — `PredDir` is not one of: IPP_VIDEO_NONE, IPP_VIDEO_HORIZONTAL, or IPP_VIDEO_VERTICAL.
  - — `VideoComp` is not one component of enum `IppVideoComponent`.

EncodeVLCZigzag_Inter_MPEG4_16s1u

Prototype

```
IppStatus ippiEncodeVLCZigzag_Inter_MPEG4_16s1u (Ipp8u **ppBitStream,
   int * pBitOffset, Ipp16s *pQDctBlkCoef,Ipp8u pattern);
```

Description

Performs classical zigzag scanning and VLC encoding for one inter block.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream
- `pBitOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7
- `pQDctBlkCoef` – pointer to the quantized DCT coefficient
- `pattern` – block pattern which is used to decide whether this block is encoded

Output Arguments

- `ppBitStream` – `*ppBitStream` is updated after the block is encoded so that it points to the current byte in the bit stream buffer.
- `pBitOffset` – `*pBitOffset` is updated so that it points to the current bit position in the byte pointed by *`ppBitStream`.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the pointers: is NULL: `ppBitStream`, `*ppBitStream`, `pBitOffset`, `pQDctBlkCoef`
  - `*pBitOffset` < 0, or `*pBitOffset` >7.
  - At least one of the following pointers is NULL: `pSrc`, `pDst`.

ComputeTextureErrorBlock_SAD_8u16s

Prototype

```
IppStatus ippiComputeTextureErrorBlock_SAD_8u16s (const Ipp8u *pSrc, int
   srcStep, const Ipp8u *pSrcRef, Ipp16s * pDst, int *pDstSAD);
```

Description

Computes texture error of the block. SAD is also exported.

Input Arguments

- `pSrc` – pointer to the source plane. This should be aligned on an 8-byte boundary.
- `srcStep` – step of the source plane

- `pSrcRef` – pointer to the reference buffer, an 8x8 block. This should be aligned on an 8-byte boundary.

Output Arguments

- `PDst` – pointer to the destination buffer, an 8x8 block. This should be aligned on an 8-byte boundary.
- `pDstSAD` – pointer to the Sum of Absolute Differences (SAD) value

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrc`, `pSrcRef`, `pDst` and `pDstSAD`.
  - `pSrc` is not 8-byte aligned.
  - `SrcStep` <= 0 or `srcStep` is not a multiple of 8.
  - `pSrcRef` is not 8-byte aligned.
  - `pDst` is not 8-byte aligned.

ComputeTextureErrorBlock_8u16s

```
IppStatus ippiComputeTextureErrorBlock_8u16s (const Ipp8u *pSrc, int
   srcStep, const Ipp8u *pSrcRef, Ipp16s * pDst);
```

Description

Computes the texture error of the block.

Input Arguments

- `pSrc` – pointer to the source plane. This should be aligned on an 8-byte boundary.
- `srcStep` – step of the source plane
- `pSrcRef` – pointer to the reference buffer, an 8x8 block. This should be aligned on an 8-byte boundary.

Output Arguments

- `pDst` – pointer to the destination buffer, an 8x8 block. This should be aligned on an 8-byte boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrc`, `pSrcRef`, `pDst`.
  - `pSrc` is not 8-byte aligned.
  - `SrcStep` <= 0 or `srcStep` is not a multiple of 8.
  - `pSrcRef` is not 8-byte aligned.
  - `pDst` is not 8-byte aligned

MotionEstimation_16x16_SEA

Prototype

```
IppStatus ippiMotionEstimation_16x16_SEA(Ipp8u * pSrcRef, Ipp8u *
   pSrcReconRef, Ipp16u *pSrcSumBlk, Ipp8u * pSrcCurr, IppiRect *
   pSrcRefRect, IppCoordinate * pSrcPointPos, IppMotionVector
   *pSrcRefMV, IppMotionVector * pDstMV, Ipp8u *pPreMbtype, int
   *pDstSAD, int step, int roundingControl, int searchRange, int flag);
```

Description

Completes 16X16 size motion estimation, the core is Successive Elimination Algorithm (SEA), the function not only covers 16X16 integer and half pixel search, 8X8 integer and half pixel search, but also decides intra/inter choice and 1mv/4mv choice. At the same time, it provides the summation of current MB's residual, which is indispensable to rate control module. This function and ippiSumNorm_VOP_MPEG4_8u16u are used in pairs typically, because the sum norm plane is one input of this function.

Input Arguments

- `pSrcRef` – pointer to the original Y plane. the pointer position is same as current macroblock's position in current plane
- `pSrcReconRef` – pointer to the reconstructed reference Y plane, the pointer position is same as current macroblock's position in current plane
- `pSrcSumBlk` – pointer to the current macroblock in the sum plane

- `pSrcCurr` – pointer to the current original macroblock, which has been extracted from current original plane

NOTE. *pSrcCurr points to a continuous macroblock size buffer, which stores extracted current original pixel from current original Y plane.*

- `pSrcRefRect` – pointer to the valid rectangular in reference plane
- `pSrcPointPos` – pointer to the position of the current macroblock in current plane
- `pSrcRefMV` – pointer to the predicted motion vector generated from the neighboring motion vector
- `step` – the step in reference Y-Plane
- `roundingControl` – rounding control bit for half pixel motion estimation
- `searchRange` – search range in the 16X16 integer block match
- `flag` - algorithm selected flag. For implementation of PCA processors with MMX^{TM}, this parameter must be 5. For the PCA processors, the parameter flag $\in \{4, 8\}$.

Output Arguments

- `pDstMV` – pointer to the destination 4-motion vectors

NOTE. *pDstMV points to 4 MV buffer's first one, 4 MVs are stored continuously. If 1MV mode is selected, then 4 MV buffer stores same one.*

- `pDstSAD` – pointer to the least SAD after motion estimation
- `pPreMbtype` – pointer to pre-Mbtype, which stores the Intra/Inter, 1MV/4MV information

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrcCurr`, `pSrcSumBlk`, `pSrcReconRef`, `pDstMV`, `pDstPreMbtype`, `pDstSAD`, `pSrcRefRect`, `pSrcPointPos`
  - At least one of the following pointers is not 64-bit aligned: `pSrcCurr`, `pSrcSumBlk`, `pSrcReconRef`, `pSrcRef` (if `pSrcRef` is not NULL)
  - If `pSrcRefMV` is not NULL, `*pSrcRefMV` exceeds search range

— `pSrcRefRect->width < MB_SIZE` or `pSrcRefRect->height < MB_SIZE`
— `*pSrcPointPos` exceeds the zone indicated by `pSrcRefRect`
— `step < pSrcRefRect->width` or step is not a multiple of 8
— `searchRange` <= 0
— `flag` ∉ {4, 8} (for PCA processors)
— `flag` ∉ {5} (for PCA processors with MMX™)

MotionEstimation_16x16_MVFAST

Prototype

```
IppStatus ippiMotionEstimation_16x16_MVFAST (Ipp8u * pSrcRef,Ipp8u *
   pSrcReconRef, Ipp8u * pSrcCurr, IppMotionVector * pSrcCanMV,
   IppMotionVector * pSrcRefMV, IppCoordinate * pSrcPointPos, IppiRect *
   pSrcRefRect, Ipp8u * pSrcSadMap, Ipp8u * pSrcBlockSadMap,
   IppMotionVector *pDstMV, Ipp8u * pDstPreMBtype, int * pDstSAD, int
   step, int roundingControl, int searchRange);
```

Description

Performs fast motion estimation using the MVFAST algorithm. Refer to N3675 (Motion Vector Field adaptive Search Algorithm).

Input Arguments

- `pSrcRef` – pointer to the original reference Y plane. The pointer position is the same as current macroblock's position in current plane.
- `pSrcReconRef` – pointer to the reconstruction reference Y plane. The pointer position is the same as current macroblock's position in current plane.
- `pSrcCurr` – pointer to the current original block, which has been stored in a 16X16 size buffer

NOTE. *pSrcCurr points to a continuous macroblock size buffer, which stores extracted current originalpixel from current original Y plane.*

- pSrcCanMV – pointer to the left, top, and right-top reference motion vector respectively
- pSrcRefMV – pointer to the predicted motion vector
- pSrcPointPos – pointer to the position of the current macroblock in the current plane
- pSrcRefRect – pointer to the valid rectangular in reference plane
- pSrcSADMap – pointer to the initial address of bit plane in 16*16 block match.
- pSrcBlockSADMap – pointer to the initial address of bit plane in 8*8 block match

NOTE. *For the buffer size and initialization for pSrcSadMap and pSrcBlockSadMap, see* Figure 13-3 *and its context.*

- step – the step in reference Y-Plane
- roundingControl – rounding control bit for half pixel motion estimation
- searchRange – search range in 16X16 integer block match

Output Arguments

- pDstMV – pointer to the destination 4-motion vectors
-

NOTE. *pDstMV points to the starting address of the motion vector buffer. If 4MV mode is selected, this buffer can store 4-motion vectors. If 1MV mode is selected, the single motion vector is duplicated into 4 copies, then stored in this motion vector buffer.*

- pDstSAD – pointer to the least SAD after motion estimation
- pDstPreMBtype – pointer to the macroblock type: Inter1v, Inter4v or Intra

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments
  - — At least one of the following pointers is NULL: pSrcCurr, pSrcReconRef, pSrcCanMV, pSrcRefMV, pSrcPointPos, pSrcRefRect, pSrcSadMap, pSrcBlockSadMap, pDstMV, pDstPreMBtype, pDstSAD.
  - — At least one of the following parameters is not 64-bit aligned: pSrcCurr, pSrcRef (if available), pSrcReconRef, step.

— At least one of the components' absolute value (|dx| or |dy|) is greater than (2*searchRange+1): pSrcRefMV, pSrcCanMV[0], pSrcCanMV[1], pSrcCanMV[2].
— At least one of the following cases is true:
pSrcRefRect->width or pSrcRefRect->height is less than or equal 16;
pSrcPointPos->x is out of range from left boarder of pSrcRefRect to its right boarder minus macroblock size; pSrcPointPos->y is out of range from top boarder of pSrcRefRect to its bottom boarder minus macroblock size;
— searchRange exceeds (0,1024]; step is less than 16; rounding. Control ∉ {0,1}.

TransRecBlockCeof_intra_MPEG4

Prototype

```
IppStatus ippiTransRecBlockCeof_intra_MPEG4 (Ipp8u *pSrc,Ipp16s * pDst,
   Ipp8u * pRec, Ipp16s *pPredBufRow, Ipp16s *pPredBufCol, Ipp16s *
   pPreACPredict, int *pSumErr,int blockIndex, Ipp8u curQp, Ipp8u
   *pQpBuf, int srcStep, int dstStep,const int * pQMatrix);
```

Description

Quantizes the DCT coefficients, implements the AC/DC coefficients prediction of the intra block, and stores them into buffer. Meanwhile, the texture data are reconstructed for next frame prediction.

Input Arguments

- PSrc – pointer to the pixels of current IntraBlock
- PCoefBufRow – pointer to the coefficient row buffer
- PCoefBufCol – pointer to the coefficient column buffer
- PSumErr – pointer to the sum of difference between predicted and unpredicted coefficients
- blockIndex – block index indicating the component type and position as defined in subclause 6.1.3.8, of *ISO/IEC 14496-2*. Furthermore, indexes 6 to 9 indicate the alpha blocks spatially corresponding to luminance blocks 0 to 3 in the same macroblock.
- curQp – quantization parameter of the macroblock which the current block belongs to
- pQpBuf – pointer to the quantization parameter buffer
- srcStep – width of the source buffer

- `dstStep` – width of the reconstructed destination buffer
- `QMatrix` – If the second inverse quantization method is used, QMatrix is NULL. If the first inverse quantization method is used, it points to the quantization weighting coefficients buffer (for intra MB) whose first 64 elements are the quantization weighting matrix in Q0. The second 64 elements are their reciprocals in Q21.

Output Arguments

- `pDst` – pointer to the quantized DCT coefficients buffer
- `pRec` – pointer to the reconstructed texture
- `pCoefBufRow` – pointer to the updated coefficient row buffer
- `pCoefBufCol` – pointer to the updated coefficient column buffer
- `pPreACPredict` – pointer to the predicted coefficients buffer. The first data indicate the predicted direction of current block.
- `pSumErr` – pointer to the updated sum of the difference between predicted and unpredicted coefficients

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `pSrc`, `pDst`, `pRec`, `pCoefBufRow`, `pCoefBufCol`, `pQpBuf`, `pPreACPredict`, `pSumErr`.
  - `BlockIndex` < 0 or `blockIndex` >= 10; `curQP` <= 0 or `curQP` >= 32.
  - `SrcStep`, `dstStep` <= 0 or not a multiple of 8.
  - At least one of the following pointers is not 64-bit aligned: `pSrc`, `pDst`, `pRec`.
  - `pQMatrix[0] * pQMatrix[64]` != (1<<21) while `pQMatrix` is not NULL.

TransRecBlockCeof_inter_MPEG4

Prototype

```
IppStatus ippiTransRecBlockCeof_inter_MPEG4 (Ipp16s *pSrc, Ipp16s *
   pDst, Ipp16s * pRec, Ipp8u QP, const int * pQMatrix);
```

Description

Implements DCT, and quantizes the DCT coefficients of the inter block while reconstructing the texture residual. There is no boundary check for the bit stream buffer.

Input Arguments

- `pSrc` – For the current InterBlock, points to the residuals to be encoded.
- `QP` – quantization parameter.
- `pQMatrix` – If the second inverse quantization method is used, pQMatrix is NULL. If the first inverse quantization method is used, it points to the quantization weighting coefficients buffer (for intra MB) whose first 64 elements are the quantization weighting matrix in Q0. The second 64 elements are their reciprocals in Q21.

Output Arguments

- `pDst` – pointer to the quantized DCT coefficients buffer
- `pRec` – pointer to the reconstructed texture residuals

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL or is not 64-bit aligned: `pSrc`, `pDst`, `pRec`.
  - QP <= 0 or QP >= 32.
  - `pQMatrix[0] * pQMatrix[64]` != (1<<21) while `pQMatrix` is not NULL.

EncodeMV_MPEG4_8u16s

Prototype

```
IppStatus ippiEncodeMV_MPEG4_8u16s (Ipp8u **ppBitStream, int
   *pBitOffset, IppMotionVector * pMVCurMB, IppMotionVector *
   pSrcMVLeftMB, IppMotionVector * pSrcMVUpperMB, IppMotionVector *
   pSrcMVUpperRightMB, Ipp8u * pTranspCurMB, Ipp8u * pTranspLeftMB,
   Ipp8u * pTranspUpperMB, Ipp8u* pTranspUpperRightMB, int fcodeForward,
   IppMacroblockType MBType);
```

Description

Finds the prediction MV and encodes the difference.

Input Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream buffer
- `pBitOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`. Valid within 0 to 7.
- `pMVCurMB` – pointer to the current macroblock motion vector
- `pSrcMVLeftMB` – pointer to the source left macroblock motion vector
- `pSrcMVUpperMB` – pointer to source upper macroblock motion vector
- `pSrcMVUpperRightMB` – pointer to source upper right MB motion vector
- `pTranspCurMB` – pointer to the transparent status buffers of the current macroblock
- `pTranspLeftMB` – pointer to the transparent status buffers of the source left macroblock
- `pTranspUpperMB` – pointer to the transparent status buffers of the source upper macroblock
- `pTranspUpperRightMB` – pointer to the transparent status buffers of the source upper macroblock
- `fcodeForward` – an integer with values from 1 to 7. This is used in encoding motion vectors related to search range.
- `MBType` – macro block type, taking values from 0 to 9

Output Arguments

- `ppBitStream` – pointer to the pointer to the current byte in the bit stream buffer
- `pBitOffset` – pointer to the bit position in the byte pointed by `*ppBitStream`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - At least one of the following pointers is NULL: `ppBitStream`, `*ppBitStream`, `pBitOffset`, `pMVCurMB`, `pTranspLeftMB`, `pTranspUpperMB`, `pTransUpperRightMB`, `pTranspCurMB`.
  - `*pBitOffset` < 0, or `*pBitOffset` >7.
  - `fcodeForward` < 0, or `fcodeForward` > 7, or `MBType` < 0.

GSM-AMR Speech CODEC 14

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) 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 7.2.1 Release 1998 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 14-1 shows the functional grouping of the GSM-AMR primitives.

Table 14-1 Summary and Functional Groupings of the GSM-AMR CODEC Primitives

Function Subset	Function Name
LP analysis	<ippsAutoCorr_GSMAMR_16s32s()>
	<ippsLevinsonDurbin_GSMAMR()>
	<ippsLPCToLSP_GSMAMR_16s()>
	<ippsLSPToLPC_GSMAMR_16s()>
	<ippsLSPQuant_GSMAMR_16s()>
	<ippsQuantLSPDecode_GSMAMR_16s()>

continued

Table 14-1 Summary and Functional Groupings of the GSM-AMR CODEC Primitives (continued)

Adaptive-codebook search	<ippsImpulseResponseTarget_GSMAMR_16s()>
	<ippsOpenLoopPitchSearchNonDTX_GSMAMR_16s()>
	<ippsOpenLoopPitchSearchDTXVAD1_GSMAMR_16s()>
	<ippsOpenLoopPitchSearchDTXVAD2_GSMAMR()>
	<ippsAdaptiveCodebookSearch_GSMAMR_16s()>
	<ippsAdaptiveCodebookDecode_GSMAMR_16s()>
Fixed-codebook search	<ippsAlgebraicCodebookSearch_GSMAMR_16s()>
	<ippsFixedCodebookDecode_GSMAMR_16s()>
	<ippsAlgebraicCodebookSearchEX_GSMAMR_16s()
Discontinuous transmission	<ippsVAD1_GSMAMR_16s()>
	<ippsVAD2_GSMAMR_16s()>
	<ippsEncDTXSID_GSMAMR_16s()>
	<ippsEncDTXHandler_GSMAMR_16s()>
	<ippsEncDTXBuffer_GSMAMR_16s()>
	<ippsDecDTXBuffer_GSMAMR_16s()>
Post-processing	<ippsPostFilter_GSMAMR_16s()>

The organization of this chapter is:

- "CODEC Architecture" provides a summary of the GSM-AMR CODEC architecture.
- "Definitions and Data Structures" describes GSM-AMR API definitions and data structures.
- Sections "LP Analysis and Quantization Primitives" through "Post Processing" provide details on the primitives that implement each of the GSM-AMR functional blocks summarized in Table 14-1
- "LP Analysis and Quantization Primitives" describes LP analysis primitives.
- Sections "Adaptive Codebook Primitives" and "Fixed Codebook Search" give details on primitives provided for the adaptive and fixed codebook search procedures, respectively.
- "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.
- "Post Processing" describes post-processing primitives.

CODEC Architecture

This section briefly describes the major functional blocks that comprise the GSM-AMR 06.90 vocoder. The section is organized as:

- "CELP Analysis-by-Synthesis" describes important features of the core CELP analysis-by-synthesis.
- Sections "Voice Activity Detection (VAD)" through "Comfort Noise Generation (CNG)", respectively, are concerned with Voice Activity Detection (VAD), frame muting and substitution, discontinuous transmission (DTX), and comfort noise generation (CNG).

CELP Analysis-by-Synthesis

The GSM-AMR CODEC core is based on the technique generally known as analysis-by-synthesis Code-Excited Linear Prediction (CELP). In this approach, coded speech is represented in terms of the parameters of a source-system model. The relatively slow time-varying parameters of the source-system model are estimated on short analysis frames of 20 millisecond (ms) duration. The compact parameter sets extracted for each frame are transmitted to the receiver and are used to synthesize output speech. The source-system model consists of two components: an all-pole digital filter (the system) and a filter excitation sequence (the source). Parameters of the digital filter, which essentially models the spectral shaping characteristics of the upper vocal tract, are estimated on each analysis interval using linear prediction. Parameters of the filter excitation, which models the behavior of the lower vocal tract (for example, pitch periodicity, voiced/unvoiced/mixed characteristics, etc.) are estimated using a closed-loop analysis-by-synthesis approach, in which a perceptually weighted error is minimized, and the selected excitation sequence is ultimately represented using a combination of vector quantization (VQ) and long-term prediction.

The spectral shaping characteristics of the upper vocal tract are captured in a 10th-order linear prediction (LP), or short-term, synthesis filter given by

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + \sum_{i=1}^{10} a_i z^{-i}}$$

where a_i, $i = 1, 2,\ldots, 10$, are the direct-form predictor coefficients. The short-term linear prediction associated with this filter tends to capture the spectral envelope or speech formants that are generated by the resonant modes of the upper vocal tract. Following the short-term prediction, long-term prediction is applied to extract the fine or pitch-induced spectral energy of the input speech. To accomplish this, a long-term or pitch synthesis filter of the form

$$\frac{1}{B(z)} = \frac{1}{1 - g_p z^{-T}}$$

is used, where the parameter T represents a pitch lag, and the parameter g_p represents the pitch gain.

At the transmitter, the fixed and adaptive codebook sequences that comprise the excitation are carefully chosen on each analysis interval. The optimum code words are identified using an analysis-by-synthesis search that minimizes explicitly a perceptually weighted error between the original and synthesized speech. An efficient representation of the vector excitation components is achieved using gain-shape VQ. The VQ parameters are transmitted to the receiver, where an excitation sequence is constructed by combining the fixed and adaptive code words. Then, the short-term synthesis filter generates output speech by filtering the combined fixed and adaptive codewords.

The analysis-by-synthesis perceptual weighting filter is given by

$$W(z) = \frac{A(z/\gamma_1)}{A(z/\gamma_2)}$$

where $A(z)$ is the synthesis filter, and the parameters $0 < \gamma_1 < \gamma_2 \leq 1$ are the perceptual weighting factors. The idea is to emphasize the matching error in those spectral regions where it is most likely to be perceived (regions of low energy), while simultaneously de emphasizing the importance of spectral matching for those regions in which they are least likely to be perceived (that is, in the vicinity of spectral peaks).

The encoder operates on 20 ms frames, which corresponds to 160 samples at the sampling frequency of 8000 Hz. On each frame, the input speech is analyzed to extract a complete set of parameters, including LP coefficients, adaptive and fixed-codebook indices, and codebook gains. These parameters are encoded and transmitted. At the decoder, the parameters are decoded and speech is synthesized by filtering the reconstructed excitation signal through the LP synthesis filter.

Voice Activity Detection (VAD)

For each input frame, the transmitter employs a VAD algorithm to determine whether or not a signal is present that requires transmission (that is, speech, music, or tones). The output of the VAD algorithm is a Boolean flag indicating presence or absence of such information-bearing signals. The idea is that other, non-critical inputs such as background noise can be represented with very few parameters and hence very few bits.

Frame Muting

Whenever possible, frame substitution is applied to conceal lost frames. In extreme cases, however (that is, multiple consecutive lost frames), frame muting is instead applied to indicate channel breakdown to the user and to avoid generating the annoying audible artifacts that could result from excessive frame substitution.

Discontinuous Transmission (DTX)

The discontinuous transmission (DTX) mechanism allows the radio transmitter (encoder) to be switched off most of the time during speech pauses. This serves the dual purposes of saving power for the radio transmitter, as well as reducing the overall and over-the-air interferences created by the transmitter.

Comfort Noise Generation (CNG)

A problem intrinsic to the use of DTX is that background acoustic noise, which is transmitted together with speech when VAD is asserted, suddenly disappears when radio transmission is stopped. As a result, perceptually annoying background noise discontinuities are created at the receiver.

The CNG algorithm mitigates this problem by generating synthetic noise at the receiver that is similar to the actual acoustical background noise present at the transmitter. Comfort noise parameters are continuously estimated at the encoder and transmitted to the receiver, such that the noise estimate is able to adapt smoothly to acoustical noise changes at the encoder.

Definitions and Data Structures

This section describes the header files, data types, and structures of the Intel® IPP GSM-AMR API.

Header Files

The header files <ippdefs.h> and <ippSC.h> must be included in order to link against any of the GSM-AMR 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

Most of the GSM-AMR primitives support multiple bit rates. As a result, the API includes an enumerated set of rate specifiers, shown in Table 14-2. The set contains one specifier for each of the supported bit rates. Primitives requiring a rate specifier expect to receive an input of the type IppSpchBitRate.

Table 14-2 IppSpchBitRate Definition and Associated Bit Rates

Data Type	Definition	Associated Bit Rate
GSM-AMR bit rates	typedef enum {	
(IppSpchBitRate)	IPP_SPCHBR_4750,	4.75 kbps
	IPP_SPCHBR_5150,	5.15 kbps
	IPP_SPCHBR_5900,	5.9 kbps
	IPP_SPCHBR_6700,	6.7 kbps
	IPP_SPCHBR_7400,	7.4 kbps
	IPP_SPCHBR_7950,	7.95 kbps
	IPP_SPCHBR_10200,	10.2 kbps
	IPP_SPCHBR_12200,	12.2 kbps
	IPP_SPCHBR_DTX,	Discontinuous TX mode
	IPP_SPCHBR_5300,	5.3 kbps
	IPP_SPCHBR_6300,	6.3 kbps
	} IppSpchBitRate;	

LP Analysis and Quantization Primitives

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_16s32s

Prototype

```
IppStatus ippsAutoCorr_GSMAMR_16s32s(const Ipp16s * pSrcSpch, Ipp32s *
   pDstAutoCorr, IppSpchBitRate mode);
```

Description

Estimates the autocorrelation sequence for a block of 240 samples (30 ms). For the 12.2 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, as follows.

$$w_1(n) = \begin{cases} 0.54 - 0.46\cos(\frac{n\pi}{159}), n = 0,1,...,159 \\ 0.54 + 0.46\cos(\frac{(n-160)\pi}{79}), n = 160,161,...,239 \end{cases}$$

and

$$w_2(n) = \begin{cases} 0.54 - 0.46\cos(\frac{2n\pi}{463}), n = 0,1,...,231 \\ \cos(\frac{2(n-232)\pi}{31}), n = 232,233,...,239 \end{cases}$$

Neither w1 nor w2 has any look ahead. For all bit rates other than 12.2 kbps, a window centered on the current frame is applied, as follows,

$$w_3(n) = \begin{cases} 0.54 - 0.46\cos(\frac{2n\pi}{399}), n = 0,1,...,199 \\ \cos(\frac{2(n-200)\pi}{159}), n = 200,201,...,239 \end{cases}$$

2. Estimation of autocorrelation lags – autocorrelation lags are estimated from the windowed speech samples $s(i)$, $i = 0, 1, \ldots, 239$, as follows

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k), k = 0,1,...,10$$

3. Bandwidth expansion – the following binomial lag window is applied to the autocorrelation sequence(s) obtained in step 2

$$bi(i) = \exp[-0.5 \times (\frac{2\pi f_0 i}{f_s})^2], i = 0,1,...,10$$

where $f_0 = 60$ Hz and $f_s = 8000$ Hz.

Input Arguments

- `pSrcSpch` – pointer to the input speech vector (240 samples), represented using Q15.0. This should be aligned on an 8-byte boundary.
- `mode` – bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Output Arguments

- `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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments
  - at least one of the following pointers: `pSrcSpch` or `pDstAutoCorr` is NULL
  - mode is not a valid element of the enumerated type IppSpchBitRate. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200.
  - `pSrcSpch` is not aligned at 8-byte boundary

LevinsonDurbin_GSMAMR

Prototype

```
IppStatus ippsLevinsonDurbin_GSMAMR(const Ipp32s * pSrcAutoCorr, Ipp16s
   * pSrcDstLpc);
```

Description

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, $i = 1, 2, \ldots, 10$, , as follows.

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k), k = 1,2,...,10$$

2. The Levinson-Durbin algorithm performs the following recursion to obtain the prediction coefficients:

$E^{[0]} = r(0)$

for $i = 1$ to 10

$a_0^{[i-1]} = 1$

$k_i = -[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j)] / E^{[i-1]}$

$a_i^{[i]} = k_i$

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, (that is, 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.

Input Arguments

- `pSrcAutoCorr` – pointer to the autocorrelation coefficients, a vector of length of 11
- `pSrcDstLpc` – pointer to the LP coefficients associated with the previous frame, a vector of length 11, represented using Q3.12

Output Arguments

`pSrcDstLpc` – pointer to the LP coefficients associated with the current frame, a vector of length 11, represented using Q3.12

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when `pSrcAutoCorr` or `pSrcDstLpc` is NULL

LPCToLSP_GSMAMR_16s

Prototype

```
IppStatus ippsLPCToLSP_GSMAMR_16s(const Ipp16s * pSrcLpc, const Ipp16s *
   pSrcPrevLsp, Ipp16s * pDstLsp);
```

Description

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

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i), i = 0,1,...,4$$

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

$$C_1(\omega) = \cos(5\omega) + f_1(1) \times \cos(4\omega) + f_1(2) \times \cos(3\omega) + f_1(3) \times \cos(2\omega)$$
$$+ f_1(4) \times \cos(\omega) + f_1(5)/2$$

$$C_2(\omega) = \cos(5\omega) + f_2(1) \times \cos(4\omega) + f_2(2) \times \cos(3\omega) + f_2(3) \times \cos(2\omega)$$
$$+ f_2(4) \times \cos(\omega) + f_2(5)/2$$

3. Evaluate $F_1(z)$ and $F_2(z)$ on 60 points equally spaced between 0 and p, 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.

Input Arguments

- `pSrcLpc` – pointer to 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

Output Arguments

`pDstLsp` – pointer to the ten-element LSP coefficient vector, represented using Q0.15

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when `pSrcLpc`, `pSrcPrevLsp` or `pDstLsp` is NULL

LSPToLPC_GSMAMR_16s

Prototype

```
IppStatus ippsLSPToLPC_GSMAMR_16s(const Ipp16s * pSrcLsp, Ipp16s *
   pDstLpc);
```

Description

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

$$f_1(j) = f_1(j) - 2q_{2i-1} \times f_1(j-1) + f_1(j-2)$$

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 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

$$a_i = \begin{cases} 0.5 \times f_1'(i) + 0.5 \times f_2'(i), i = 1,2,...,5 \\ 0.5 \times f_1'(11-i) - 0.5 \times f_2'(11-i), i = 6,7,...,10 \end{cases}$$

Input Arguments

`pSrcLsp` – pointer to the ten-element LSP coefficient vector, represented using Q0.15

Output Arguments

`pDstLpc` – pointer to the eleven-element LP coefficient vector, represented using Q3.12

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, `pSrcLsp` or `pDstLpc` is NULL

LSPQuant_GSMAMR_16s

Prototype

```
IppStatus ippsLSPQuant_GSMAMR_16s(const Ipp16s * pSrcLsp, Ipp16s *
   pSrcDstPrevQLsfResidual, Ipp16s * pDstQLsp, Ipp16s * pDstQLspIndex,
   IppSpchBitRate  mode);
```

Description

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 relation

$$f_i = \frac{f_s}{2\pi}\arccos(q_i)$$

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:

$$r^{(1)}(n) = z^{(1)}(n) - 0.65\hat{r}^{(2)}(n-1),\ r^{(2)}(n) = z^{(2)}(n) - 0.65\hat{r}^{(2)}(n-1)$$

   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:

$$r_j(n) = z_j(n) - \alpha_j \hat{r}_j(n-1) \quad j = 0,...,9$$

   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

$$w_i = \begin{cases} 3.347 - \dfrac{1.547}{450} d_i, & d_i < 450 \\ 1.8 - \dfrac{0.8}{1050}(d_i - 450), & otherwise \end{cases}$$

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.

Input Arguments

- `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
- `mode` – bit rate specifier. The enumerated values of `IPP_SPCHBR_4750` to IPP_SPCHBR_12200 are valid

Output Arguments

- `pSrcDstPrevQLsfResidual` – pointer 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 previous discussion of SMQ and SVQ for the various modes).

Returns

- `ippStsNoErr` – no error

- ippStsBadArgErr – bad arguments, returned when pSrcLsp, pSrcDstPrevLsfResidual, pDstQLsp, pDstQLspIndex is NULL; or mode is not a valid element of the enumerated type IppSpchBitRate. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

QuantLSPDecode_GSMAMR_16s

Prototype

```
IppStatus ippsQuantLSPDecode_GSMAMR_16s(const Ipp16s * pSrcQLspIndex,
   Ipp16s * pSrcDstPrevQLsfResidual, Ipp16s * pSrcDstPrevQLsf, Ipp16s *
   pSrcDstPrevQLsp, Ipp16s * pDstQLsp, Ipp16s bfi, IppSpchBitRate mode);
```

Description

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:

$$\begin{cases} lsf\_q1(i) = lsf\_q2(i) = \alpha \times past\_lsf\_q(i) + (1-\alpha) \times mean\_lsf(i) & 12.2\,\text{kbit/s mode} \\ lsf\_q(i) = \alpha \times past\_lsf\_q(i) + (1-\alpha) \times mean\_lsf(i) & otherwise \end{cases}$$

   where $i = 0, 1, \ldots, 9$, a = 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 than12.2 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.

Input Arguments

- `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
- `pSrcDstPrevQLsf` – pointer to the ten-element quantized LSF vector from the previous frame, represented using Q0.15
- `pSrcDstPrevQLsp` – pointer to the ten-element quantized LSP vector from the previous frame, represented 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.

Output Arguments

- `pSrcDstPrevQLsfResidual` – pointer to the ten-element updated LSF residual, represented using Q0.15
- `pSrcDstPrevQLsf` – pointer to the ten-element updated quantized LSF vector, represented using Q0.15
- `pSrcDstPrevQLsp` – pointer 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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments detected, returned when `pSrcQLspIndex`, `pSrcDstPrevLsfResidual`, `pSrcDstPrevQLsf`, `pSrcDstPrevQLsp`, `pDstQLsp`, is NULL, or mode is not a valid element of the enumerated type `IppSpchBitRate`. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

Adaptive Codebook Primitives

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

The section is organized as follows. First, "Adaptive Codebook Primitives" describes the three open-loop pitch search primitives, including those appropriate for Non-DTX, VAD1, and VAD2-encoded frames. Then, sections "ImpulseResponseTarget_GSMAMR_16s" through "AdaptiveCodebookSearch_GSMAMR_16s", respectively, describe the primitives for impulse response and target signal computation, adaptive codebook search, and adaptive codebook vector decoding.

Open-Loop Pitch Search (OLP)

Three primitives are provided for OLP search. Use of these primitives is mutually exclusive. That is, only one is appropriate for an application during any given frame type. The appropriate choice of an OLP search primitive depends upon the state of the DTX and VAD modules. The OLP search primitives should be applied as follows:

Encoder Mode	appropriate primitive
DTX disabled	ippsOpenLoopPitchSearchNonDTX_GSMAMR_16s()
DTX, VAD 1 enabled	ippsOpenLoopPitchSearchDTXVAD1_GSMAMR_16s()
DTX, VAD 2 enabled	ippsOpenLoopPitchSearchDTXVAD2_GSMAMR()

The OLP search primitives 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 employe 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, as follows.

$$R(k) = w(k) \times \sum_{n=0}^{79} sw(n) \times sw(n-k), \quad 20 \le k \le 143$$

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, as follows.

$$wn(k) = \begin{cases} cw(|T_{old} - k| + 20) & weightflag = 1 \\ 1 & otherwise \end{cases}$$

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

$$g = \frac{\sum_{n=0}^{79} sw(n) \times sw(n - T_{op})}{\sum_{n=0}^{79} sw(n) \times sw(n)} - 0.4$$

where,

$$v = \begin{cases} 1 & g > 0 \\ 0.9v & otherwise \end{cases}$$

and

$$weightflag = \begin{cases} 1 & v > 0.3 \\ 0 & otherwise \end{cases}$$

If $g > 0$, the previous pitch lag buffer and 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, denoted by $\hat{M}_i, i = 0,1,2$ are identified on three different bitrate-specific search intervals 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. Next, normalized correlation maxima, M_i, are obtained using the relation:

$$M_i = \hat{M}_i / \sqrt{\sum_{n=0}^{length-1} sw^2(n - t_i)}$$

where the correlation lags associated with the maxima M_i are denoted by the lag parameter t_i, for $i = 0,1,2$.

c. Finally, the best open-loop lag, T_{op}, is selected from among the three candidates t_i using the following pitch multiple avoidance procedure:

$$T_{op} = T_1, \quad M(T_{op}) = M_1$$
$$if\ (M_2 > 0.85\, M(T_{op}))$$
$$M(T_{op}) = M_2, \quad and \quad T_{op} = T_2$$
$$if\ (M_3 > 0.85\, M(T_{op}))$$
$$T_{op} = T_3$$

Each of the OLP search primitives implements the rate-dependent search algorithms described above. Next, details are given for non-DTX, VAD1, and VAD2 OLP search primitives.

OpenLoopPitchSearchNonDTX_GSMAMR_16s

Prototype

```
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);
```

Description

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)".

Input Arguments

- `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. This argument is used only for 10.2 kbps frames.
- pValResultVvalue – pointer to the adaptive parameter *v* described in "Open-Loop Pitch Search (OLP)", represented using Q0.15. This argument is used 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. 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
- mode – bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid

Output Arguments

- pValResultPrevMidPitchLag – pointer 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. This argument is valid only for 10.2 kbps frames.
- pSrcDstPrevPitchLag – pointer to the updated five-element vector of pitch lags associated with the five previous voiced half-frames. This argument is valid only for 10.2 kbps frames.
- pSrcDstPrevWgtSpch – pointer 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

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad arguments, returned when any of the input or output pointers is NULL; or whenever the mode is not a valid element of the enumerated type IppSpchBitRate. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

OpenLoopPitchSearchDTXVAD1_GSMAMR_16s

Prototype

```
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);
```

Description

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 the "Open-Loop Pitch Search (OLP)" section 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

$$toneflag >>= 1$$

$$DelayEnergy = \sum_{n=0}^{79} sw^2(n - T_{op})$$

$$MaxCorr = \sum_{n=0}^{79} sw(n - T_{op}) \times sw(n)$$

$$if\ (0.325 \times DelayEnergy < MaxCorr) \quad toneflag = toneflag\ |\ 0x4000$$

   b. On the second OLP search for the current frame, find the maximum of the high-pass filtered autocorrelations, as follows.

$$maxhpcorr = \max(R(k) \times 2 - R(k-1) - R(k+1)\ |\ k = 142,...,24)$$

   Then, *maxhpcorr* is normalized by *NormFactor* = *frameEnergy* - *frameCorr*:

$$framenergy = \sum_{n=0}^{79} sw^2(n), \qquad framecorr = \sum_{n=0}^{79} sw(n) \times sw(n-1)$$

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:

$$toneflag = toneflag >> 1, \quad toneflag = toneflag \mid 0x2000$$

   b. After finding three open-loop pitch candidates, update the tone flag as follows:

$$if\,(DelayEnergy \times 0.65 < MaxCorr) \quad toneflag = toneflag \mid 0x4000$$

   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.

Input Arguments

- `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. This argument is valid only for 10.2 kbps frames.
- `pValResultVvalue` – pointer to the 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. 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.
- `mode` – bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Output Arguments

- `pValResultToneFlag` – pointer to the updated tone flag for the VAD module
- `pValResultPrevMidPitchLag` – pointer 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 updated adaptive parameter *v*, represented using Q0.15. This argument is valid only for 10.2 kbps frames
- `pSrcDstPrevPitchLag` – pointer 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 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

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when any of the input or output pointers is NULL; or mode is not a valid element of the enumerated type `IppSpchBitRate`. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

OpenLoopPitchSearchDTXVAD2_GSMAMR

Prototype

```
IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR(const Ipp16s *
   pSrcWgtLpc1, const Ipp16s * pSrcWgtLpc2, const Ipp16s * pSrcSpch,
   Ipp16s * pValResultPrevMidPitchLag, pp16s * pValResultVvalue, Ipp16s
   * pSrcDstPrevPitchLag, Ipp16s * pSrcDstPrevWgtSpch, Ipp32s *
   pResultMaxCorr, Ipp32s pResultWgtEnergy, Ipp16s * pDstOpenLoopLag,
   Ipp16s * pDstOpenLoopGain, IppSpchBitRate mode);
```

Description

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 the 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).

Input Arguments

- `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. This argument is valid only for 10.2 kbps frames.
- `pValResultVvalue` – pointer to the 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. 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
- `mode` – bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Output Arguments

- `pValResultPrevMidPitchLag` – pointer 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 updated adaptive parameter *v*, represented using Q0.15. This argument is valid only for 10.2 kbps frames.
- `pSrcDstPrevPitchLag` – pointer 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 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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when any of the input or output pointers is NULL; or mode is not a valid element of the enumerated type `IppSpchBitRate`. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

ImpulseResponseTarget_GSMAMR_16s

Prototype

```
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);
```

Description

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, $H(z)W(z) = A(z/\gamma_1)/[\hat{A}(z)A(z/\gamma_2)]$, 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, $1/\hat{A}(z)$, 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, as follows,

$$res_{LP}(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i s(n-i)$$

Input Arguments

- `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

Output Arguments

- `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

Returns

- `ippStsNoErr` – no error

- `ippStsBadArgErr` – bad arguments, returned when pSrcSpch, pSrcWgtLpc1, `pSrcWgtLpc2`, `pSrcQLpc`, `pSrcSynFltState`, `pSrcWgtFltState`, `pDstImpulseResponse`, `pDstLpResidual`; or `pDstAdptTarget` is NULL

AdaptiveCodebookSearch_GSMAMR_16s

Prototype

```
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);
```

Description

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, as follows,

$$R(k)_t = \sum_{i=0}^{3} R(k-i) b_{24}(t + i \cdot 6) + \sum_{i=0}^{3} R(k+1+i) b_{24}(6 - t + i \cdot 6)$$

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, as follows,

$$v(n) = \sum_{i=0}^{9} u(n-k-i) b_{60}(t + i \cdot 6) + \sum_{i=0}^{9} u(n-k+1+i) b_{60}(6 - t + i \cdot 6)$$

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.

Input Arguments

- `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.
- `subFrame` – subframe index
- mode – bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

Output Arguments

- `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

Returns

- `ippStsNoErr` – no error
- ippStsBadArgErr – bad arguments, pSrcTarget, pSrcImpulseResponse, pSrcOpenLoopLag, pValResultPrevIntPitchLag, pSrcDstExcitation, pResultFracPitchLag, pResultAdptIndex or pDstAdptVector is NULL; or mode is not a valid element of the enumerated type IppSpchBitRate, or subFrame is not in the range of [0, 3]; or pSrcImpulseResponse or pSrcTarget is not aligned on an 8-byte boundary. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

AdaptiveCodebookDecode_GSMAMR_16s

Prototype

```
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);
```

Description

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 4^{th} 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.

Input Arguments

- `valAdptIndex` – adaptive codebook index
- `pValResultPrevIntPitchLag` – pointer to the previous integer pitch lag
- `pValResultLtpLag` – pointer to the LTP-Lag value.
- `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.
- `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.

Output Arguments

- `pValResultPrevIntPitchLag` – pointer to the previous integral pitch lag
- `pValResultLtpLag` – pointer to the LTP-Lag value.
- `pSrcDstExcitation` – pointer to the 194-element excitation vector. Elements 0 ~ 153 contain the past excitation, represented using Q15.0. 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

Returns

- `ippStsNoErr` – no error
- ippStsBadArgErr – bad arguments, returned when pValResultPrevIntPitchLag, pValResultLtpLag, pSrcDstExcitation, pResultIntPitchLag, pDstAdptVector is NULL; or subFrame is not in the range of [0, 3]; or mode is not a valid element of the enumerated type IppSpchBitRate. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

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_16s

Prototype

```
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);
```

Description

These two 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_16s` uses an internal working buffer pointed by `pBuffer` allocated by user, but `ippsAlgebraicCodebookSearch_GSMAMR_16s` allocates this internal working buffer in stack.

ippsAlgebraicCodebookSearch_GSMAMR_16s can't be used on an operating system that cannot allocate more than 4K stack space for one process.

Input Arguments

- `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.
- `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

Output Arguments

- `pSrcDstImpulseResponse` – pointer 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.
- `pBuffer` – pointer to internal working buffer, in length of 1K 32-bit integer unit. This pointer should be aligned at the 8-byte boundary.

Returns

- `ippStsNoErr` – no error

- ippStsBadArgErr – bad arguments, returned when any of the input and output pointers is NULL; or mode is not a valid element of the enumerated type IppSpchBitRate; or pSrcFixedTarget or pSrcDstImpulseResponse is not aligned at 8-byte boundary. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

FixedCodebookDecode_GSMAMR_16s

Prototype

```
IppStatus ippsFixedCodebookDecode_GSMAMR_16s(const Ipp16s *
   pSrcFixedIndex, Ipp16s * pDstFixedVector, Ipp16s subFrame,
   IppSpchBitRate mode);
```

Description

This function decodes the fixed codebook vector from the received fixed codebook index.

Input Arguments

- `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.
- `subFrame` – subframe index
- mode – bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Output Arguments

- `pDstFixedVector` – pointer to the 40-element fixed codebook vector

Returns

- `ippStsNoErr` – no error
- ippStsBadArgErr – bad arguments, returned when pSrcFixedIndex, pDstFixedVector is NULL or subFrame is not in the range [0, 3]; or mode is not a valid element of the enumerated type IppSpchBitRate. Use a value in the range of between IPP_SPCHBR_4750 to IPP_SPCHBR_12200 for mode.

Discontinuous Transmission (DTX)

This section describes primitives that are concerned with discontinuous transmission (DTX), including primitives that perform the following functions:

- 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.

VAD1_GSMAMR_16s

Prototype

```
IppStatus ippsVAD1_GSMAMR_16s(const Ipp16s pSrcSpch, IppGSMAMRVad1State
   * pValResultVad1State, Ipp16s * pResultVadFlag, Ipp16s maxHpCorr,
   Ipp16s toneFlag);
```

Description

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.

Input Arguments

- `pSrcSpch` – pointer to the input speech signal, in the length of 160, in Q0.
- `pValResultVad1State` – pointer to the VAD Option 1 history variables. Its structure type `IppGSMAMRVad1State` is defined later.
- `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.

Output Arguments

- `pValResultVad1State` – pointer to the updated VAD Option 1 history variables. The structure `IppGSMAMRVad1State` is defined later.
- `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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when any of the input and output pointers is NULL

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 5^{th} order filters of filter bank.
Ipp16s pThirdFltState[5];	The history state of the five 3^{rd} 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.

Structure definition of IppGSMAMRVad1State:

typedef struct{	**Description**
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.

Notes

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.

VAD2_GSMAMR_16s

Prototype

```
IppStatus ippsVAD2_GSMAMR_16s(const Ipp16s * pSrcSpch,
   IppGSMAMRVad2State * pValResultVad2State, Ipp16s * pResultVadFlag,
   Ipp16s ltpFlag);
```

Description

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.

Input Arguments

- `pSrcSpch` – pointer to the input speech frame, in the length of 160, in Q0
- `pValResultVad2State` – pointer to the history variables of VAD Option 2. The structure `IppGSMAMRVad2State` is defined below.

- `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*.

Output Arguments

- `pValResultVad2State` – pointer to the history variables of VAD Option 2. 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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when any of the input and output pointers is NULL

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 $\bar{E}_{dB}$, 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.
Ipp16s hangOverCount;	Hangover counter *h*(*m*) used in the 10ms half-frame's VAD Decision.
Ipp32s frameCount;	Half-frame counter.
}IppGmrVad2State;	

Notes

All elements in the structure should be initialized to zero whenever encoder is reset.

EncDTXSID_GSMAMR_16s

Prototype

```
IppStatus ippsEncDTXSID_GSMAMR_16s(const Ipp16s * pSrcLspBuffer, const
   Ipp16s * pSrcLogEnergyBuffer, Ipp16s * pValResultLogEnergyIndex,
   Ipp16s * pValResultDtxLsfRefIndex, Ipp16s * pSrcDstQLsfIndex, Ipp16s
   * pSrcDstPredQErr, Ipp16s * pSrcDstPredQErrMR122, Ipp16s sidFlag);
```

Description

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:

$$EnLogIndex = \frac{1}{8}\sum_{n=0}^{7} EnLog(t-n)$$

   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:

$$PredErr(i) = EnLogIndex$$
$$\Pr edErrMR122(i) = (EnLogIndex)/(20\times\log 10(2))$$
$$i = 0,...,3$$

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:

$$Lsfref\_Index = \min_{index}\left(\sum_{n=0}^{9}(Lsf(n) - Lsf\_ref_{index}(n))^2 \mid index = 0,...,8\right)$$

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.

Input Arguments

- `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 log energy index of last frame, in Q2.13
- `pValResultDtxLsfRefIndex` – pointer to the LSF quantization reference index of last frame
- `pSrcDstDtxQLsfIndex` – pointer to the LSF residual quantization indices of last 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.
- `pSrcDstPredQErrMR122` – pointer to the fixed gain prediction error of four previous subframes for 12.2 Kbps, 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.

Output Arguments

- `pValResultLogEnergyIndex` – pointer to log energy index of current frame, in Q2.13
- `pValResultDtxLsfRefIndex` – pointer to the LSF quantization reference index of current DTX frame
- `pSrcDstDtxQLsfIndex` – pointer to the LSF residual quantization indices of current frame, in the length of 3
- `pSrcDstPredQErr` – pointer 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 updated fixed gain prediction error for 12.2 Kbps mode, in the length of 4, in Q5.10

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when any of the input and output pointers is NULL

EncDTXHandler_GSMAMR_16s

Prototype

```
IppStatus ippsEncDTXHandler_GSMAMR_16s(Ipp16s * pValResultHangOverCount,
   Ipp16s *pValResultDtxElapseCount, Ipp16s * pValResultUsedMode, Ipp16s
   * pResultSidFlag, Ipp16s vadFlag);
```

Description

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 :

   $$DTX\_HangOver\_Count + DTX\_ElaspedCount < 30$$

   The SID flag is set to 0, and the transmission mode is set to “MRDTX”.

Input Arguments

- `pValResultHangOverCount` – pointer to the DTX hangover count. When initialized or reset, it is set to 0.
- `pValResultDtxElaspCount` – pointer to elapsed frame count since last non-DTX frame. When initialized or reset, it is set 0.
- `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.
- `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.

Output Arguments

- `pValResultHangOverCount` – pointer to the updated DTX hangover count
- `pValValResultDtxElaspedCount` – pointer to the updated elapsed frame count since last non-DTX frame
- `pValResultUsedMode` – pointer to the transmission mode. 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.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when any of the input and output pointers is NULL

EncDTXBuffer_GSMAMR_16s
DecDTXBuffer_GSMAMR_16s

Prototype

```
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);
```

Description

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 below:

$$EnLog = \log 2(\frac{1}{N}\sum_{n=0}^{N-1} s^2(n))$$

Where N is the frame length, and s is the input speech signal.

Input Arguments

- `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. It is a value circularly increased between 0 and 7.
- `pSrcDstLspBuffer` – pointer to the LSP 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

Output Arguments

- `pValResultUpdateIndex` – pointer the current memory update index. It is circularly increased between 0 and 7
- `pSrcDstLspBuffer` – pointer to the LSP coefficients of eight most recent frames (including current frame), in the length of 80, in Q0.15
- `pSrcDstLogEnergyBuffer` – pointer to the log energy coefficients of eight most recent frames (including current frame), in the length of 8, in Q5.10

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, returned when any of the input and output pointers is NULL

Post Processing

PostFilter_GSMAMR_16s

Prototype

```
IppStatus ippsPostFilter_GSMAMR_16s(const Ipp16s * pSrcQLpc, const Ipp16s *
   pSrcSpch, Ipp16s * pValResultPrevResidual, Ipp16s *
   pValResultPrevScalingGain, Ipp16s * pSrcDstFormantFIRState, Ipp16s *
   pSrcDstFormantIIRState, Ipp16s * pDstFltSpch, IppSpchBitRate mode);
```

Description

Filters the synthesized speech to enhance reconstruction quality. This primitive implements the following procedure:

1. Get the weighted LP coefficients of the formant postfilter:

$$H_f(z) = \frac{\hat{A}(z/\gamma_n)}{\hat{A}(z/\gamma_d)}$$

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:

$$k_1' = \frac{r_h(1)}{r_h(0)} \quad , r_h(i) = \sum_{j=0}^{21-i} h(j) \times h(j+i)$$

Then compute the tilt factor: $\mu = \gamma_t k_1'$

For 12.2 and 10.2 Kbps mode,

$$\gamma_t = \begin{cases} 0.8 & k_1' > 0, \\ 0 & otherwise \end{cases}$$

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: $HI(z) = \frac{1}{\hat{A}(z/\gamma_d)}$

3. Compute the adaptive gain scaling factor:

$$\gamma_{sc} = \sqrt{\frac{\sum_{n=0}^{39} \hat{s}(n)}{\sum_{n=0}^{39} \hat{s}_f(n)}}$$

Here, $\hat{s}(n)$ is the synthesized speech, and $\hat{s}_f(n)$ is the post-filtered speech. The output speech is given by:

$$\hat{s}'(n) = \beta_{sc}(n)\hat{s}_f(n)$$

G.723.1 Speech CODEC 15

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) that can be combined to construct a bit-exact implementation of the International Telecommunications Union (ITU) ITU-T G.723.1 speech CODEC. The primitives are primarily concerned with the well-defined, computationally expensive operations that comprise the core of the G.723.1 algorithm.

The G.723.1 speech CODEC is a fixed-rate algorithm that represents efficient telephone-bandwidth digital speech using compressed data rates 5.3 and 6.3 kilobits per second (kbps). Algebraic-Code-Excited-Linear-Prediction (ACELP) is used at the lower rate of 5.3 kbps, and Multipulse-Maximum-Likelihood-Quantization (MP-MLQ) is used during 6.3 kbps operation.

Table 15-1 shows the functional groupings of the G.723.1 primitives.

Table 15-1 Summary and Functional Groupings of the G.723.1 CODEC Primitives

Function Subset	Function Name
LP Analysis	<ippsAutoCorr_G723_16s()>
	<ippsLevinsonDurbin_G723_16s()>
	<ippsLPCToLSF_G723_16s()>
	<ippsLSFToLPC_G723_16s()>
	<ippsLSFQuant_G723_16s32s()>
Adaptive-codebook search	<ippsOpenLoopPitchSearch_G723_16s()>
	<ippsHarmonicSearch_G723_16s()>
	<ippsAdaptiveCodebookSearch_G723()>
	<ippsDecodeAdaptiveVector_G723_16s()>

continued

Table 15-1 Summary and Functional Groupings of the G.723.1 CODEC Primitives (continued)

Fixed-codebook search	<ippsToeplizMatrix_G723_16s()>
	<ippsMPMLQFixedCodebookSearch_G723()>
	<ippsACELPFixedCodebookSearch_G723_16s()>
Filtering	<ippsSynthesisFilter_G723_16s()>
	<ippsPitchPostFilter_G723_16s()>

CODEC Architecture

This section briefly describes the major functional blocks of the G.723.1 vocoder.

The G723.1 algorithm is designed to operate with a digital signal obtained by first performing telephone bandwidth filtering (ITU-T Recommendation G.712) of the analog input, then sampling at 8000 Hz and then converting to 16-bit linear PCM for the input to the encoder. The output of the decoder should be converted back to an analog signal by a similar means. Other input/output characteristics, such as those specified by ITU-T Recommendation G.711 for 64 kbit/s PCM data, should be converted to 16-bit linear PCM before encoding or from 16-bit linear PCM to the appropriate format after decoding.

The coder is based on the principles of linear prediction analysis-by-synthesis coding and attempts to minimize a perceptually weighted error signal. The encoder operates on frames of 240 samples, or 30 msec at an 8 kHz sampling rate. Each frame is first high pass filtered to remove the DC component and then divided into four subframes of 60 samples each. For every subframe, a 10th order LPC filter coefficients are computed using the unprocessed input signal. The LPC filter for the last subframe is quantized using Predictive Split Vector Quantizer (PSVQ). The unquantized LP coefficients are used to construct the short-term perceptual weighting filter, which is used to filter the entire frame and to obtain the perceptually weighted speech signal.

For every two subframes (120 samples), the open loop pitch period is computed using the weighted speech signal. This pitch estimation is performed on blocks of 120 samples. The pitch period is searched in the range from 18 to 142 samples. For all remaining functions, speech is processed in 60-sample subframe blocks.

Using the estimated pitch period computed previously, a harmonic noise-shaping filters is constructed. The combination of the LPC synthesis filter, the formant perceptual weighting filter, and the harmonic noise-shaping filter is used to create an impulse response. The impulse response is then used for further computations.

Using the pitch period estimation and the impulse response, a closed loop pitch predictor is computed and a fifth-order pitch predictor is used. The pitch period is computed as a small differential value around the open loop pitch estimate. The contribution of the pitch predictor is then subtracted from the initial target vector. Both the pitch period and the differential value are transmitted to the decoder.

Finally the non-periodic component of the excitation is approximated. For the high bit rate MP-MLQ excitation is used, and for the low bit rate, an ACELP is used.

The decoder operation is also performed on a frame-by-frame basis. First the quantized LPC indices are decoded, then the decoder constructs the LPC synthesis filter. For every subframe, both the adaptive codebook excitation and fixed codebook excitation are decoded and input to the synthesis filter. The adaptive post-filter consists of a formant and a forward-backward pitch post-filter. The excitation signal is input to the pitch post-filter, which in turn is input to the synthesis filter whose output is input to the formant post-filter. A gain-scaling unit maintains the energy at the input level of the formant post-filter.

Encoder and decoder block diagrams appear, respectively, in Figure 15-1 and Figure 15-2.

Figure 15-1 G.723.1 Encoder

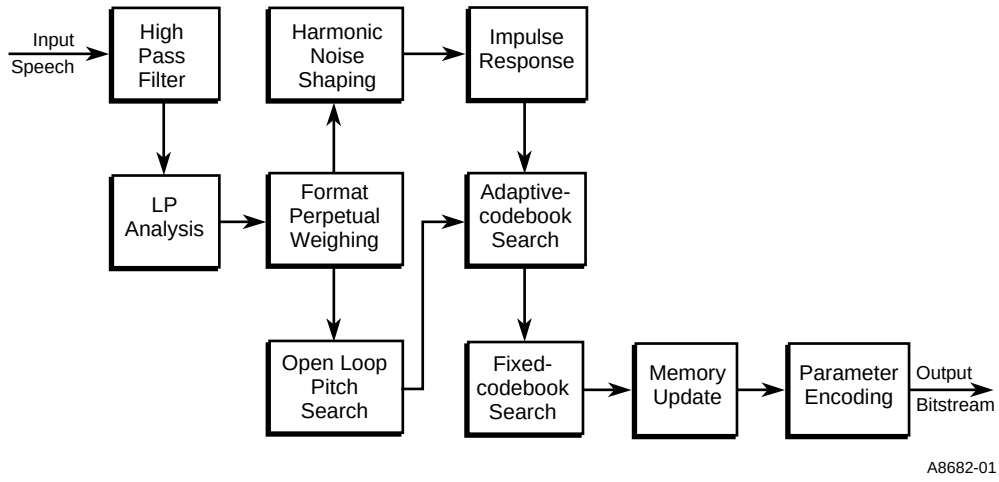
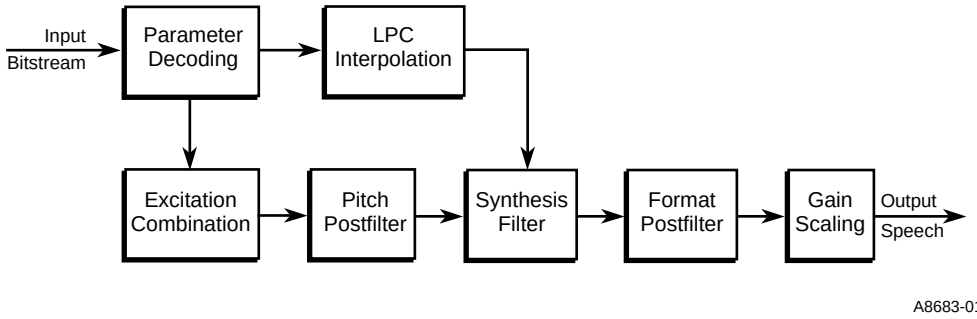


Figure 15-2 723.1 Decoder



Definitions and Data Structures

This section describes the definitions and header files of the G.723.1 API.

Header Files

The header files <ippdefs.h> and <ippSC.h> must be included in order to link against any of the G.723.1 primitives, as shown in the following example code:

```
#include "ippdefs.h"
#include "ippSC.h"
int main()
{
    ...
    /* call G.732.1 IPP functions *
    ippsAutoCorr_G723_16s(pSrcSpch, pResultAutoCorrExp, pDstAutoCorr);
    ...
}
```

Data Structures

A few of the G.723.1 primitives support multiple bit rates. As a result, the API includes an enumerated set (Table 15-2) of rate specifiers. The set contains one specifier for each of the supported bit rates. Primitives requiring a rate specifier expect to receive an input of the type `IppSpchBitRate`.

Table 15-2 IppSpchBitRate Definition and Associated Bit Rates

Data Type	Definition	Associated Bit Rate
G.723.1 bit rates (IppSpchBitRate)	typedef enum {	
	IPP_SPCHBR_4750,	4.75 kbps
	IPP_SPCHBR_5150,	5.15 kbps
	IPP_SPCHBR_5900,	5.9 kbps
	IPP_SPCHBR_6700,	6.7 kbps
	IPP_SPCHBR_7400,	7.4 kbps
	IPP_SPCHBR_7950,	7.95 kbps
	IPP_SPCHBR_10200,	10.2 kbps
	IPP_SPCHBR_12200,	12.2 kbps
	IPP_SPCHBR_DTX,	Discontinuous TX mode
	IPP_SPCHBR_5300,	5.3 kbps
	IPP_SPCHBR_6300,	6.3 kbps
	} IppSpchBitRate;	

LP Analysis Primitives

The following sections describe the G.723.1 LP Analysis primitives.

- Autocorrelation
- Levinson-Durbin Algorithm
- LPC to LSF Conversion
- LSF to LPC Conversion
- LSF Quantization

AutoCorr_G723_16s

Prototype

```
IppStatus ippsAutoCorr_G723_16s(const Ipp16s * pSrcSpch, Ipp16s *
   pResultAutoCorrExp, Ipp16s * pDstAutoCorr);
```

Description

This function calculates the first 11 elements of autocorrelation coefficients of speech samples. It is applied in subframes to 180 speech samples centered on the current subframe.

1. Add the speech samples with Hamming window. The Hamming window is given by the following formula.

$$w(i) = 0.54 + 0.46\cos[(\frac{2i}{179} - 1)\pi], i = 0,1,...179$$

2. Calculate the autocorrelation of the windowed speech samples using the formula below.

$$r(k) = \sum_{i=k}^{179} s(i) \times s(i-k), k = 0,1,...,10$$

3. Add the autocorrelation with a binomial window.

$$bi(0) = \frac{1025}{1024}$$

$$bi(i) = \exp[-0.5 \times (\frac{2\pi f_0 i}{f_s})^2], i = 1,...,10$$

where f_o = 42.1 Hz and f_s = 8000 Hz.

Input Arguments

`pSrcSpch` – pointer to the input speech signal in the length of 180

Output Arguments

- `pResultAutoCorrExp` – pointer to the exponential of the autocorrelation coefficients in the length of 1

- `pDstAutoCorr` – pointer to the autocorrelation coefficients in the length of 11

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments: `pSrcSpch`, `pResultAutoCorrExp` or `pDstAutoCorr` are NULL

LevinsonDurbin_G723_16s

Prototype

```
IppStatus ippsLevinsonDurbin_G723_16s(const Ipp16s * pSrcAutoCorr,
   Ipp16s * pValResultSineDtct, Ipp16s * pResultResidualEnergy, Ipp16s *
   pDstLpc);
```

Description

This function calculates the 10-order LP coefficients from the autocorrelation coefficients using Levinson-Durbin algorithm. Also it detects the sine circumstance. It is applied in subframes.

1. Levinson-Durbin algorithm is to solve the equation as below to obtain the LP coefficients ai, i = 1, 2, …, 10.

$$\sum_{i=1}^{10} a_i \times r(|\, i-k\,|) = r(k), k = 1,2,...,10$$

2. The detailed Levinson-Durbin algorithm uses the following recursion.

$$E^{[0]} = r(0)$$

for i = 1 to 10

$$a_0^{[i-1]} = 1$$

$$k_i = -[a_0^{[i-1]} \times r(i) - \sum_{j=1}^{i-1} a_j^{[i-1]} \times r(i-j)] / E^{[i-1]}$$

$$a_i^{[i]} = k_i$$

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. If the LPC filter in this algorithm is unstable, that is $k_i \geq 1$ in the recursion, just force the remaining LP coefficients to 0.
4. The sine circumstance is updated when i = 1 in the recursion. The formulas below show the algorithm.

   `SineDtct` = `SineDtct` << 1

   If $k_1 > 0.95$, then `SineDtct` = `SineDtct` + 1

Input Arguments

- `pSrcAutoCorr` – pointer to the autocorrelation coefficients in the length of 11
- `pValResultSineDtct` – pointer to the sine circumstance in the length of 1

Output Arguments

- `pDstLpc` – pointer to the LP coefficients in the length of 10, in Q2.13
- `pValResultSineDtct` – pointer to the updated sine circumstance in the length of 1
- `pResultResidualEnergy` – pointer to the residual energy in the length of 1

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, `pSrcAutoCorr`, `pValResultSineDtct`, `pDstLpc` or `pResultResidualEnergy` are NULL

LPCToLSF_G723_16s

Prototype

```
IppStatus ippsLPCToLSF_G723_16s(const Ipp16s * pSrcLpc, const Ipp16s *
   pSrcPrevLsf, Ipp16s * pDstLsf);
```

Description

This function converts a set of 10-order LP coefficients to LSF coefficients. The functionality is as below.

1. A bandwidth expansion of 7.5 Hz is applied to the LP coefficients.
2. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursion relations as below.

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i), i = 0,1,...,4$$

   where $f_1(0) = f_2(0) = 1.0$.
3. Use Chebyshev polynomials to find the root of $F_1(z)$ and $F_2(z)$ and obtain the LSF coefficients. The Chebyshev polynomials are as below.

$$C_1(\omega) = \cos(5\omega) + f_1(1) \times \cos(4\omega) + f_1(2) \times \cos(3\omega) + f_1(3) \times \cos(2\omega)$$
$$+ f_1(4) \times \cos(\omega) + f_1(5)/2$$

$$C_2(\omega) = \cos(5\omega) + f_2(1) \times \cos(4\omega) + f_2(2) \times \cos(3\omega) + f_2(3) \times \cos(2\omega)$$
$$+ f_2(4) \times \cos(\omega) + f_2(5)/2$$

4. If could not find 10 roots for LSF coefficients, just use the previous set of LSF coefficients.

Input Arguments

- `pSrcLpc` – pointer to LP coefficients in the length of 10, in Q2.13
- `pSrcPrevLsf` – pointer to previous normalized LSF coefficients in the length of 10, in Q15

Output Arguments

`pDstLsf` – pointer to the normalized LSF coefficients in the length of 10, in Q15

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, `pSrcLpc`, `pSrcPrevLsf` or `pDstLsf` are NULL.

LSFToLPC_G723_16s

Prototype

```
IppStatus ippsLSFToLPC_G723_16s(const Ipp16s * pSrcLsf, Ipp16s *
   pDstLpc);
```

Description

This function converts a set of 10-order LSF coefficients to LP coefficients. The functionality is as below.

1. Convert the LSF coefficients to LSP coefficients, using the formula below.

$$q_i = \cos(\omega_i), i = 0,1,...,9$$

2. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursion relations as below.

   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

   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}.

3. $F_1(z)$ and $F_2(z)$ are then multiplied by $1+z^{-1}$ and $1-z^{-1}$ respectively to obtain $F_1'(z)$ and $F_2'(z)$ using the formula below.

$$f_1'(i) = f_1(i) + f_1(i-1)$$

$$f_2'(i) = f_2(i) - f_2(i-1), i = 1,2,...5$$

4. The LP coefficients are computed from the polynomials of $F_1'(z)$ and $F_2'(z)$ by the formula below.

$$a_i = \begin{cases} 0.5 \times f_1'(i) + 0.5 \times f_2'(i), i = 1,2,...,5 \\ 0.5 \times f_1'(11-i) - 0.5 \times f_2'(11-i), i = 6,7,...,10 \end{cases}$$

Input Arguments

`pSrcLsf` – pointer to the normalized LSF coefficients in the length of 10, in Q15

Output Arguments

`pDstLpc` – pointer to LP coefficients in the length of 10, in Q2.13

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments: `pSrcLsf` or `pDstLpc` are NULL

LSFQuant_G723_16s32s

Prototype

```
IppStatus ippsLSFQuant_G723_16s32s(const Ipp16s * pSrcLsf, const Ipp16s *
   pSrcPrevLsf, Ipp32s * pResultQLsfIndex);
```

Description

This function quantizes the LSF coefficients to obtain the codebook indexes using PSVQ. The functionality is as below.

1. Calculate the diagonal weighting matrix $\boldsymbol{W}$, determined from the unquantized LSF coefficients. The elements are defined by the following formulas.

$$w_{1,1} = 1/(\omega_2 - \omega_1)$$

$$w_{10,10} = 1/(\omega_{10} - \omega_9)$$

$$w_{j,j} = 1/\min(\omega_j - \omega_{j-1}, \omega_{j+1} - \omega_j), j = 2,3,...9$$

2. Calculate the prediction LSF coefficients using the formula below.
   $\omega_p = (\omega - \omega_{DC}) - 0.375 \times (\omega_{-1} - \omega_{DC})$
   where ω is the vector of the current LSF coefficients, is ω_{-1} the vector of previous LSF coefficients, and ω_{DC} is the vector of DC LSF coefficients.
3. Search the codebook vector to minimize the error defined below.
   $E_l = (\omega_p - \omega_l)^T \boldsymbol{W} (\omega_p - \omega_l)$
   where ω_l is the *l*-th code vector in the code book. Notice that here use 3-3-4 split vector quantization.

Input Arguments

- `pSrcLsf` – pointer to LSF coefficients in the length of 10, in Q15
- `pSrcPrevLsf` – pointer to previous LSF coefficients in the length of 10, in Q15

Output Arguments

`pResultQLsfIndex` – pointer to combined index of quantized LSF coefficients

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments, `pSrcLsf`, `pSrcPrevLsf` or `pResultQLsfIndex` are NULL

Adaptive Codebook Search Primitives

The following sections describe the G.723.1 LP Adaptive Codebook Search primitives

- open loop pitch search
- harmonic search
- adaptive-codebook search
- adaptive vector decode

OpenLoopPitchSearch_G723_16s

Prototype

```
IppStatus ippsOpenLoopPitchSearch_G723_16s(const Ipp16s * pSrcWgtSpch,
   Ipp16s * pResultOpenDelay);
```

Description

This function extracts the open loop pitch from the weighted speech signal. It is applied in half-frames and the functionality is as below.

1. A cross correlation criterion, $C_{ol}(j)$, maximization method is used to determine the pitch period, using the formula below.

$$C_{ol}(j) = (\sum_{i=0}^{119} s(i) \times s(i-j))^2 / (\sum_{i=0}^{119} s(i-j) \times s(i-j)), j = 18,19,...,142$$

   The index j which maximize the cross correlation, $C_{ol}(j)$, is selected as the open loop pitch for the half-frame.

2. While searching for the best index, some preference is given to smaller pitch periods to avoid choosing pitch multiples. Maximums of $C_{ol}(j)$ are searched beginning with j = 18. For every maximum $C_{ol}(j)$ found, its value is compared to the best previous maximum found, $C_{ol}(j')$. If the difference between indices j and j' is less than 18 and $C_{ol}(j) > C_{ol}(j')$, the new maximum is selected. If the difference between the indices is greater than or equal to 18, the new maximum is selected only if $C_{ol}(j)$ is greater than $C_{ol}(j')$ by 1.25 dB.

Input Arguments

`pSrcWgtSpch` – pointer to the weighted speech in the length of 265. The pointer points to the location of 146.

Output Arguments

`pResultOpenDelay` – pointer to the open loop pitch in the length of 1, in Q0

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments: `pSrcWgtSpch` or `pResultOpenDelay` are NULL

HarmonicSearch_G723_16s

Prototype

```
IppStatus ippsHarmonicSearch_G723_16s(Ipp16s valOpenDelay, const Ipp16s
   * pSrcWgtSpch, Ipp16s * pResultHarmonicDelay, Ipp16s *
   pResultHarmonicGain);
```

Description

This function searches the harmonic delay and harmonic gain for the harmonic noise shaping filter from the weighted speech and open loop pitch. It is applied in subframes and the functionality is as below.

1. Find the maximum of $C_{pw}(j)$ and the optimal index L using the formula below.

$$C_{pw}(j) = (\sum_{i=0}^{59} s(i) \times s(i-j))^2 / \sum_{i=0}^{59} s(i-j) \times s(i-j), j = L_1, ..., L_2$$

   where L_1 is obtained as open delay minus 3, and L_2 is obtained as open delay add 3. L is the harmonic delay.

2. Calculate the optimal filter gain, G_{opt} using the formula below.

$$G_{opt} = \sum_{i=0}^{59} s(i) \times s(i-j) / \sum_{i=0}^{59} s(i-j) \times s(i-j)$$

   And then limit G_{opt} to the range [0, 1].

3. Calculate the energy of weighted speech samples using the formula below.

$$E = \sum_{i=0}^{59} s^2(i)$$

4. Calculate the harmonic gain, β using the formula below.

$$\beta = \begin{cases} 0.3125 \times G_{opt}, if \ -10\log_{10}(1 - \frac{C_L}{E}) \geq 2.0 \\ 0.0, otherwise \end{cases}$$

Input Arguments

- `valOpenDelay` – open loop pitch, in Q0

- `pSrcWgtSpch` – pointer to the weighted speech in the length of 205. The pointer points to the location of 146.

Output Arguments

- `pResultHarmonicDelay` – pointer to the harmonic delay in the length of 1, in Q0
- `pResultHarmonicGain` – pointer to the harmonic gain in the length of 1, in Q15

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments: `pSrcWgtSpch`, `pResultHarmonicDelay` or `pResultHarmonicGain` are NULL, or `valOpenDelay` is not in the range of [18, 145].

AdaptiveCodebookSearch_G723

Prototype

```
IppStatus ippsAdaptiveCodebookSearch_G723(Ipp16s valBaseDelay, const
   Ipp16s * pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse, const
   Ipp16s * pSrcPrevExcitation, const Ipp32s * pSrcPrevError, Ipp16s *
   pResultCloseLag, Ipp16s * pResultAdptGainIndex, Ipp16s subFrame,
   Ipp16s sineDtct, IppSpchBitRate bitRate);
```

Description

This function searches for the close loop pitch and adaptive gain index. It is applied in subframes and the introduction is as below.

1. For subframes 0 and 2, the close loop pitch lag is selected from around the appropriate open loop pitch in the range of ±1. For subframes 1 and 3, the close loop pitch lag is selected from appropriate open loop pitch in the range of [-1, 3].
2. Denote the close loop pitch as L_i, i = 0, 1, 2 and 3. The pitch predictor gains are vector quantized using two codebooks with 85 and 170 entries for the 6.3 Kbps bit rate and 170 entries for the 5.3 Kbps bit rate. The 170-entry codebook is the same for both rates. For the 6.3 Kbps bit rate, if L_0 is less than 58 for subframes 0 and 1 or if L_2 is less than 58 for subframes 2 and 3, then the 85-entry codebook is used for pitch gain quantization. Otherwise, the pitch gain is quantized using the 170-entry codebook.

Input Arguments

- `valBaseDelay` – base delay, in Q0
- `pSrcAdptTarget` – pointer to the adaptive target signal in the length of 60
- `pSrcImpulseResponse` – pointer to the impulse response in the length of 60
- `pSrcPrevExcitation` – pointer to the previous excitation in the length of 145
- `pSrcPrevError` – pointer to the previous error in the length of 5, in 32-bit format
- `subFrame` – subframe number, in Q0, from 0 to 3
- `bitRate` – transmit bit rate, IPP_SPCHBR_6300 stands for 6.3 Kbps and IPP_SPCHBR_5300 stands for 5.3 Kbps.
- `sineDtct` – sine circumstance

Output Arguments

- `pResultCloseLag` – pointer to the lag of close pitch in the length of 1, in Q0
- `pResultAdptGainIndex` – pointer to the index of adaptive gain in the length of 1, in Q0

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments: `pSrcAdptTarget`, `pSrcImpulseResponse`, `pSrcPrevExcitation`, `pSrcPrevError`, `pResultCloseLag` or `pResultAdptGainIndex` are NULL, or `valBaseDelay` is not in the range of [18, 145], or `subFrame` is not in the range of [0, 3], or `bitRate` is not IPP_SPCHBR_6300 or IPP_SPCHBR_5300

DecodeAdaptiveVector_G723_16s

Prototype

```
IppStatus ippsDecodeAdaptiveVector_G723_16s(Ipp16s valBaseDelay, Ipp16s
   valCloseLag, Ipp16s valAdptGainIndex, const Ipp16s *
   pSrcPrevExcitation, Ipp16s * pDstAdptVector, IppSpchBitRate bitRate);
```

Description

This function decodes the adaptive vector from excitation, close loop pitch, adaptive gain index and bit rate. It is applied in subframes and the functionality is as below.

1. Obtain the pitch use the formula below.

$$L = L_o + L_c - 1$$

where L_o is the open loop pitch and L_c is the close loop pitch lag.

2. Obtain the residue from the previous excitation using the formulas below.

$$res(0) = res(1) = 0$$

$$res(i+2) = exci((i\%L) - L), i = 0,1,...,61$$

3. Select the adaptive codebook, G_b, between 85 or 170 entry codebooks.
4. Obtain the adaptive vector using the formula below.

$$c(i) = \sum_{j=0}^{4} res(i+j) \times G_b(20 \times k + i), i = 0,1,...59$$

where k is the adaptive gain index.

Input Arguments

- `valBaseDelay` – base delay, in Q0
- `valCloseLag` – lag of close pitch, in Q0
- `valAdptGainIndex` – index of adaptive gain, in Q0
- `pSrcPrevExcitation` – pointer to the previous excitation in the length of 145
- `bitRate` – transmit bit rate, IPP_SPCHBR_6300 stands for 6.3 Kbps and IPP_SPCHBR_5300 stands for 5.3 Kbps.

Output Arguments

`pDstAdptVector` – pointer to the adaptive vector in the length of 60

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments: `pSrcPrevExcitation` or `pDstAdptVector` are NULL, or `valBaseDelay` is not in the range of [18, 145], or `valCloseLag` is not in the range of [-1, 3], or `valAdptGainIndex` is not in the range of [0, 170), or `bitRate` is not IPP_SPCHBR_6300 or IPP_SPCHBR_5300.

Fixed-codebook Search Primitives

The following sections describe the G.723.1 Fixed-codebook Search primitives

- Toepliz Matrix Calculation
- MP-MLQ Fixed-codebook Search

- ACELP fixed-codebook search

ToeplizMatrix_G723_16s

Prototype

```
IppStatus ippsToeplizMatrix_G723_16s(const Ipp16s * pSrcImpulseResponse,
   Ipp16s * pDstMatrix);
```

Description

This function computes the 416 elements of the Toepliz matrix for the fixed-codebook search.

$$\Phi(i,j) = \sum_{n=j}^{59} h(n-i) \times h(n-j), i \le j, 0 \le i \le 59$$

where $h(i)$, $i = 0, 1, \ldots, 59$ is the impulse response.

Input Arguments

`pSrcImpulseResponse` – pointer to the impulse response in the length of 60

Output Arguments

`pDstMatrix` – pointer to the elements of Toepliz matrix in the length of 416

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments: `pSrcImpulseResponse` or `pDstMatrix` are NULL

NOTE. *Only 416 elements in the Toepliz Matrix, F (i, j), would be calculated in this function. And the arrangement of pointer pDstMatrix is as below.*

1. $\Phi(m_i, m_j)$, (i = 0, 1, 2, 3), 4×8 = 32, start location: 0.
   $\Phi(0,0)$, $\Phi(8,8)$, ..., $\Phi(56, 56)$, $\Phi(2, 2)$, $\Phi(10, 10)$, ..., $\Phi(58, 58)$, $\Phi(4,4)$, $\Phi(12,12)$, ...,$\Phi(60, 60)$, $\Phi(6, 6)$, $\Phi(14, 14)$, ..., $\Phi(62,62)$.
2. $\Phi(m_0, m_1)$, 8×8 = 64, start location: 32.
   $\Phi(0, 2)$, ..., $\Phi(0, 58)$, $\Phi(8, 2)$, ..., $\Phi(8, 58)$, $\Phi(16, 2)$, ..., $\Phi(56,58)$.
3. $\Phi(m_0, m_2)$, 8×8 = 64, start location: 96.
   $\Phi(0, 4)$, ..., $\Phi(0, 60)$, $\Phi(8, 4)$,..., $\Phi(8, 60)$, $\Phi(16, 4)$, ..., $\Phi(56, 60)$.
4. $\Phi(m_0, m_3)$, 8×8 = 64, start location: 160.
   $\Phi(0, 6)$, ..., $\Phi(0, 62)$, $\Phi(8, 6)$, ..., $\Phi(8, 62)$, $\Phi(16, 6)$, ..., $\Phi(56, 62)$.
5. $\Phi(m_1, m_2)$, 8×8 = 64, start location: 224.
   $\Phi(2, 4)$, ..., $\Phi(2, 60)$, $\Phi(10, 4)$, ..., $\Phi(10, 60)$, $\Phi(18, 4)$, ..., $\Phi(58, 60)$.
6. $\Phi(m_1, m_3)$, 8×8 = 64, start location: 288.
   $\Phi(2, 6)$, ..., $\Phi(2, 62)$, $\Phi(10, 6)$, ..., $\Phi(10, 62)$, $\Phi(18, 6)$, ..., $\Phi(58, 62)$.
7. $\Phi(m_2, m_3)$, 8×8 = 64, start location: 352.
   $\Phi(4, 6)$, ..., $\Phi(4, 62)$, $\Phi(12, 6)$, ..., $\Phi(12, 62)$, $\Phi(20, 6)$, ..., $\Phi(60, 62)$.

MPMLQFixedCodebookSearch_G723

Prototype

```
IppStatus ippsMPMLQFixedCodebookSearch_G723(Ipp16s valBaseDelay, const
   Ipp16s * pSrcImpulseResponse, const Ipp16s * pSrcResidualTarget,
   Ipp16s * pDstFixedVector, Ipp16s * pResultGrid, Ipp16s *
   pResultTrainDirac, Ipp16s * pResultAmpIndex, Ipp16s *
   pResultAmplitude, Ipp32s * pResultPosition,  Ipp16s subFrame);
```

Description

This function searches for the MP-MLQ fixed-codebook for the excitation for 6.3 Kbps bit rate. It is applied in subframes and the functionality is as below.

1. The error vector is obtained using the formula below.

$$err(n) = res(n) - G \times \sum_{k=0}^{M-1} \alpha_k \times h(n - m_k)$$

where G is the gain factor, α_k, $k = 0, 1, \ldots, M\text{-}1$ and m_k, $k = 0, 1, \ldots, M\text{-}1$ are the signs and the positions of fixed vector respectively and M is the number of pulses.

2. Search for the parameters, G, α_k, $k = 0, 1, \ldots, M\text{-}1$, and m_k, $k = 0, 1, \ldots, M\text{-}1$ that minimize the mean square of the error signal $err(n)$, $n = 0, 1, \ldots, 59$.

3. The fixed-codebook gain is obtained using the formulas below.

$$d(j) = \sum_{n=j}^{59} c(n) \times h(n - j), 0 \le j \le 59$$

$$G_m = \frac{\max\{|d(j)|\}_{j=0,1,\ldots,59}}{\sum_{n=0}^{59} h(n) \times h(n)}$$

where $c(n)$, $n = 0, 1, \ldots, 59$ is the fixed vector.

Input Arguments

- `valBaseDelay` – base delay, in Q0
- `pSrcImpulseResponse` – pointer to the impulse response in the length of 60
- `pSrcResidualTarget` – pointer to the residue target signal in the length of 60
- `subFrame` – subframe number, in Q0, from 0 to 3

Output Arguments

- `pDstFixedVector` – pointer to the fixed codebook vector in the length of 60
- `pResultGrid` – pointer to the begin grid location, in Q0, 0 or 1
- `pResultTrainDirac` – pointer to the flag if train Dirac function used, 0: unused, 1: used
- `pResultAmpIndex` – pointer to the index of quantized amplitude, in Q0
- `pResultAmplitude` – pointer to the amplitude of the fixed codebook vector
- `pResultPosition` – pointer to the position of fixed codebook vector, which amplitude is not equal to 0, in Q0

Returns

- `ippStsNoErr` – no error

- `ippStsBadArgErr` – bad arguments: `pSrcImpulseResponse`, `pSrcResidualTarget`, `pDstFixedVector`, `pResultGrid`, `pResultTrainDirac`, `pResultAmpIndex`, `pResultAmplitude` or `pResultPosition` are NULL, or `valBaseDelay` is not in the range of [18, 145], or `subFrame` is not in the range of [0, 3]

ACELPFixedCodebookSearch_G723_16s

Prototype

```
IppStatus ippsACELPFixedCodebookSearch_G723_16s(const Ipp16s *
   pSrcFixedCorr, const Ipp16s * pSrcMatrix, Ipp16s * pDstFixedSign,
   Ipp16s * pDstFixedPosition, Ipp16s * pResultGrid, Ipp16s *
   pDstFixedVector, Ipp16s * pSearchTimes);
```

Description

This function searches for the ACELP fixed codebook for the excitation for 5.3 Kbps bit rate. It is applied in subframes and the functionality is as below.

1. The fixed vector contains four non-zero pulses and could be shown as below.

$$c(n) = \sum_{k=0}^{3} \alpha_k \times \delta(n - m_k), n = 0,1,...,59$$

   where α_k, $k = 0, 1, \ldots, 3$ and m_k, $k = 0, 1, \ldots, 3$ are the signs and the positions of fixed vector respectively.

2. Search for the parameters, α_k, $k = 0, 1, \ldots, 3$, and m_k, $k = 0, 1, \ldots, 3$ that minimize error shown below.

$$\varepsilon = \Phi(m_0, m_0) + \Phi(m_1, m_1) + 2\alpha_0\alpha_1\Phi(m_0, m_1)$$

$$+ \Phi(m_2, m_2) + 2[\alpha_0\alpha_2\Phi(m_0, m_2) + \alpha_1\alpha_2\Phi(m_1, m_2)]$$

$$+ \Phi(m_3, m_3) + 2[\alpha_0\alpha_3\Phi(m_0, m_3) + \alpha_1\alpha_3\Phi(m_1, m_3) + \alpha_2\alpha_3\Phi(m_2, m_3)]$$

   where $\Phi(i, j)$ is the Toepliz Matrix.

Input Arguments

- `pSrcFixedCorr` – pointer to the correlation between residue and impulse response, in the length of 60

- pSrcMatrix – pointer to the elements of Toepliz matrix in the length of 416
- pSearchTimes – pointer to the maximum search time, in Q0

Output Arguments

- pDstFixedSign – pointer to the sign of the fixed vector, in the length of 4
- pDstFixedPosition – pointer to the position of the fixed vector in the length of 4
- pResultGrid – pointer to the begin grid location, in Q0
- pDstFixedVector – pointer to the fixed vector in the length of 60
- pSearchTimes – pointer to the maximum search time for the remaining subframes, in Q0

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – bad argument: pSrcFixedCorr, pSrcMatrix, pSearchTimes, pDstFixedSign, pDstFixedPosition, or pDstFixedVector are NULL.

Filtering Primitives

The following sections describe the G.723.1 Filtering primitives.

- Synthesis Filter
- Pitch Post Filter

SynthesisFilter_G723_16s

Prototype

```
IppStatus ippsSynthesisFilter_G723_16s(const Ipp16s * pSrcLpc, const
   Ipp16s * pSrcResidual, Ipp16s * pSrcDstIIRState, Ipp16s * pDstSpch);
```

Description

This function implements the LPC synthesis filter. It is applied in subframes and the functionality is shown in the formula below.

$$s(n) = u(n) - \sum_{i=1}^{10} a_i \times s(n-i), n = 0,1,...,59$$

Input Arguments

- pSrcLpc – pointer to the LP coefficients in the length of 10, in Q2.13
- `pSrcResidual` – pointer to the residual signal in the length of 60
- `pSrcDstIIRState` – pointer to the history of synthesized speech signal in the length of 10

Output Arguments

- `pDstSpch` – pointer to the output speech signal in the length of 60
- `pSrcDstIIRState` – pointer to the updated history of synthesized speech signal in the length of 10

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments: `pSrcLpc`, `pSrcResidual`, `pSrcDstIIRState` or `pDstSpch` are NULL

PitchPostFilter_G723_16s

Prototype

```
IppStatus ippsPitchPostFilter_G723_16s(Ipp16s valBaseDelay, const Ipp16s
   * pSrcResidual, Ipp16s * pResultDelay, Ipp16s * pResultPitchGain,
   Ipp16s * pResultScalingGain, Ipp16s subFrame, IppSpchBitRate
   bitRate);
```

Description

This function calculates the coefficients of the pitch post filter. It is applied in subframes and the functionality is as below.

1. Search for the forward pitch lag M_f from the cross correlation shown below.

$$C_f = \sum_{i=0}^{59} res(i) \times res(i + M_f), M_1 \le M_f \le M_2$$

where M_1 is obtained as base delay minus 3, and M_2 is obtained as base delay add 3.

2. Search for the backward pitch lag M_b from the cross correlation shown below.

$$C_b = \sum_{i=0}^{59} res(i) \times res(i - M_b), M_1 \le M_b \le M_2$$

also M_1 is obtained as base delay minus 3, and M_2 is obtained as base delay add 3.

3. In (1) and (2), if one of the maximum, C_f or C_b is negative or if for some i in the range [0,59], there is no residue samples $res(i+ M_f)$ available, then the corresponding weight and delay are set to 0. This makes four possible cases: (a) both maxima are negative and no pitch postfilter weights need to be computed, (b) only the forward maximum is positive, so it is selected, (c) only the backward maximum is positive, so it is selected, (d) both maxima are positive and the one making the larger contribution is selected.

4. Calculate the relevant residue energy, T_{en}, D_f and D_b using the formulas below.

$$D_f = \sum_{i=0}^{59} res(i + M_f) \times res(i + M_f)$$

$$D_b = \sum_{i=0}^{59} res(i - M_b) \times res(i - M_b)$$

$$T_{en} = \sum_{i=0}^{59} res(i) \times res(i)$$

5. According to the decision in (3), obtain the optimal gain using the formula below.

$$g = C / D$$

Notice that the decision in (3) (d) is to maximum the result of C^2/D.

6. Calculate the scaling gain using the formula below.

$$g_s = \sqrt{\frac{\sum_{i=0}^{59} res^2(i)}{\sum_{i=0}^{59} res'^2(i)}}$$

where $res'(i)$, $i = 0, 1, \ldots, 59$ is the post filtered residue.

7. Calculate the total gain of the post filter using the formula below.

$$g_p = \gamma_{ltp} \times g_s \times g$$

Input Arguments

- `valBaseDelay` – base delay, in Q0

- pSrcResidual – pointer to the residual signal in the length of 365. The pointer points to the location of 146.
- subFrame – subframe number, in Q0, from 0 to 3
- bitRate – transmit bit rate, IPP_SPCHBR_6300 stands for 6.3 Kbps and IPP_SPCHBR_5300 stands for 5.3 Kbps

Output Arguments

- pResultDelay – point to the delay of the pitch post filter, in Q0
- pResultPitchGain – point to the gain of the pitch post filter, in Q15
- pResultScalingGain – point to the scaling gain of the pitch post filter, in Q15

Returns

ippStsNoErr – no error

- ippStsBadArgErr – bad arguments: pSrcResidual, pResultDelay, pResultPitchGain or pResultScalingGain are NULL, or valBaseDelay is not in the range of [18, 145], or subFrame is not in the range of [0, 3], or bitRate is not IPP_SPCHBR_6300 or IPP_SPCHBR_5300

Image Processing 16

This chapter describes general image processing primitives of the Intel® Integrated Performance Primitives (Intel® IPP) for the Intel® Integrated Performance Primitives on Intel® PCA Processors with Intel® Wireless MMX™ Technology (PCA processors with MMX™). The IPP image primitives provide the following function sets:

- initialization
- arithmetic and logical
- filtering
- linear transforms
- color space conversion
- morphological
- threshold
- statistics

In the interest of simplicity and consistency, the mathematical expressions given in this chapter to describe the behavior of each primitive represent the particular case of the non-in-place and non–scaled function variable (the so-called “default” function version). The user should be aware that the behavior of any scaled and/or in-place variables can be understood easily by applying to the default behavioral specification the generic in-place and scaled function behavioral rules that are given in Chapter 1, “Introduction”

The remainder of this chapter is organized as follows. First, the “Image Data Types” section describes the data type used in image primitives. The “Image Processing Models” section describes the major image operation models. Finally, sections “Image Initialization Primitives” through “Image Statistical Primitives” detail initial, Arithmetic and Logical Operations, FIR Filtering, Linear Transforms, Color Space Conversion, Base Morphological Operations, Threshold, and Statistics primitives. Each section also provides example 'C' language source listings that demonstrate the usage of the Intel® IPP image primitives.

Image Data Types

IPP supports only absolute color images in which each pixel is represented by its channel intensities. The data storage for an image can be either pixel-oriented (or called interleave format) or plane-oriented (or called planar format). For images in pixel-oriented, all channel values for each pixel are clustered and stored consecutively, for example, RGBRGBRGB..... for an RGB image. The number of channels in a pixel-order image can be one or three (the fourth channel alpha channel is not currently supported) and can be identified by the function name descriptor C1 or C3. The name RGB also indicates that the data are stored in RGB order. For example, in function IppStatus ippiRGBToYUV_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst, int dstStep, IppiSize roiSize), both input and output are in C3 format. The input pointer pSrc will point to data formatted as RGBRGBRGB....., and similarly output pointer pDst will point to data formatted as YVUYUVYUV....

For images in plane-oriented, all image data for each channel is stored contiguously. One channel followed by the next channel, for example, RRR...GGG...BBB. Primitives that operate on planar images are identified by the presence of P3 descriptor in their names. In this case, three pointers (one for each plane) are specified. For an example, in function ippiRGBToYUV_8u_C3P3R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst[3], int dstStep, IppiSize roiSize), descriptor C3P3 means that input is in pixel-oriented format of three channel and output is in plane-oriented format of three channel. Therefore input pointer pSrc will point to data block RGBRGBRGB..... and output pointer pDst[0] will point to data block YYY....., pDst[1] will point to UUU....., and pDst[2] will point to VVV.

The image data type is determined by the pixel depth in bits per channel, or bit depth. Bit depth for each channel can be 8, 16 or 32 and is included in the function name as one of these numbers. The data type may be signed (s) or unsigned (u). All channels in an image must have the same data type.

For example, in an absolute color 24-bit RGB image, three consecutive bytes (24 bits) per pixel represent the three channel intensities in pixel mode. This data type is identified in function names as 8u_C3 descriptor, where 8u represents 8-bit unsigned data for each channel and C3 represents three channels.

The range of values that can be represented by each data type lies between the lower and upper bounds. Table 16-1 lists data ranges and constant identifiers used in IPP to denote the respective range bounds:

Table 16-1 Data Ranges and Identifiers

	Data Types Identifier	Lower Bound Value	Identifier	Upper Bound Value
8s	IPP_MIN_8s	-2^7	IPP_MAX_8s	2^7-1
8u		0	IPP_MAX_8u	2^8-1
16s	IPP_MIN_16s	-2^{15}	IPP_MAX_16s	$2^{15}-1$
16u		0	IPP_MAX_16u	$2^{16}-1$
32s	IPP_MIN_32s	-2^{31}	IPP_MAX_32s	$2^{31}-1$
32u		0	IPP_MAX_32u	$2^{32}-1$
32f	IPP_MINABS_32f	$1.17549435.10^{-38}$	IPP_MAXABS_32f	$3.402823466.10^{38}$

Image Processing Models

Most IPP image processing primitives perform identical and independent operations on all channels of the processed image. It means that the same operation is applied to each channel, and the computed results do not depend upon values of other channels. The only exceptions are the `ippiFilterMedianColor` function and color conversion primitives, which process three channels together.

The IPP image processing primitives can be broken into two major models of operation. The first model includes primitives that operate on one pixel to compute the result (also known as point operations), for example, `ippiAdd`. The second model includes primitives that operate on a group of pixels (also referred to as a neighborhood), for example, `ippiFilterBox`.

Neighborhood Operations

The result of a neighborhood operation is based on values of a certain group of pixels, located near a given input pixel. The set of neighboring pixels is typically defined by the size of rectangular mask (or kernel) and anchor cell, specifying the mask alignment with respect to the position of the input pixel.

The IPP primitives that process a neighborhood operate on the assumption that all referred points of the image are available. To support this mode, the application must check that ROI parameters passed to the function have such values that all processed neighborhood pixels actually exist in the image.

Only the following filtering and morphological primitives perform Neighborhood operations.

- `ippiFilterBox`
- `ippiFilterRow`

- `ippiFilterColumn`
- `ippiFilter`
- `ippiFilterMedian`
- `ippiFilterColorMedian`
- `ippiErode` and `ippiDilate`

Rectangle or Region of Interest in IPP

Most IPP image processing primitives can operate not only on entire images but also on a part of the image. The Region of Interest or Rectangle of Interest (ROI) are rectangular areas which may be either some part of the image or the whole image.

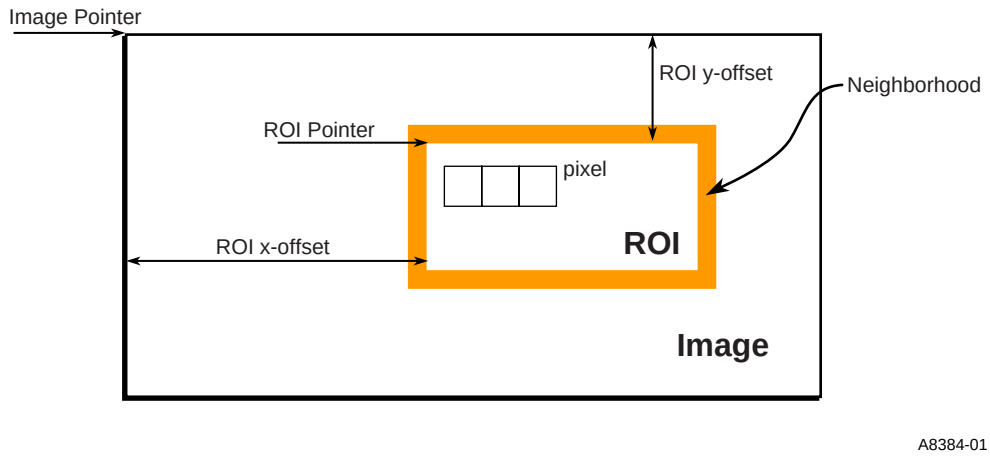
IPP primitives with ROI support are distinguished by the presence of an R descriptor in their names. ROI of an image is defined by the size and offset from the image's origin as shown in Figure 16-1. The origin of an image is implied to be in the top left corner, with x values increasing from left to right and y values increasing downwards.

Both the source and destination images can have a rectangle of interest. In such cases, the sizes of ROIs are assumed to be the same while offsets may differ. The image processing is then performed on data of the source ROI, and the results are written to the destination ROI. In function call sequences, an ROI is specified by:

- roiSize argument of the IppiSize type
- pSrc and pDst pointers to the starts of source and destination ROI buffers
- srcStep and dstStep arguments which are equal to distances in bytes between the starts of consecutive lines in source and destination images, respectively.

Thus, the arguments srcStep, dstStep set steps in bytes through image buffers to start processing a new line in the ROI of an image.

Figure 16-1 Image, ROI, and Offsets



Example 16-1 Use of the dstStep Parameter in Function Calls

```
IppStatus alignedLine( void ) {
   Ipp8u x[8*3] = {0};
   IppiSize imgSize = {5,3};
   // The image is of size 5x3. Width 8 has been
   // chosen by the user to align every line of the image
   return ippiSet_8u_C1R(7, x, 8, imgSize);
}
```

The resultant image x contains the following data:

```
07 07 07 07 07 00 00 00
07 07 07 07 07 00 00 00
07 07 07 07 07 00 00 00
```

If ROI is present,

- source and destination images can have different sizes
- image may have padding at the end for aligning the line sizes
- application must correctly define the pSrc, pDst and roiSize arguments

The pSrc and pDst arguments are the shifted pointers to the image data. For example, in case of ROI operations on 3-bytes-per-pixel image data (8u_C3R), pSrc points to the start of the source ROI buffer and can be interpreted as follows:

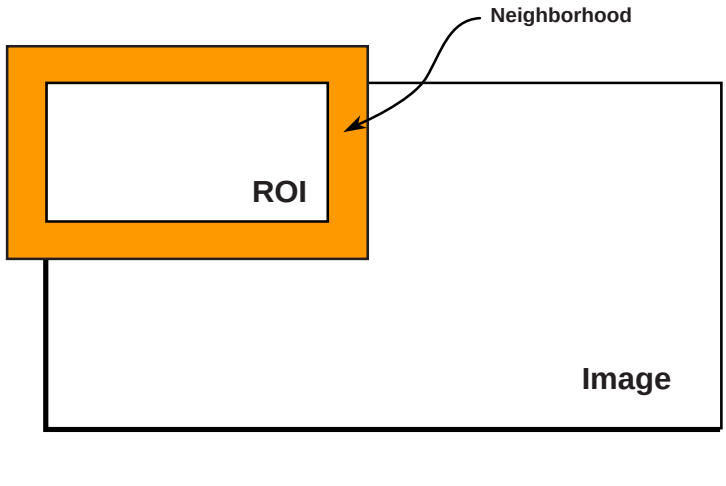
```
pSrc = pSrcImg
      +3*(srcImgSize.width * srcRoiOffset.y + srcRoiOffset.x)
```

where

- `pSrcImg` points to the start of the source image buffer;
- `srcImgSize` is the image size in pixels (of the IppiSize type);
- `srcRoiOffset` determines an offset of ROI relative to the start of the image as shown in the Figure 16-1.

For primitives using ROI with a neighborhood, you should correctly use values of the pSrc and roiSize parameters. These primitives assume that the points in the neighborhood exist and that therefore pSrc is almost never equal to `pSrcImg`. Figure 16-2 illustrates when neighborhood pixels can fall outside the source image.

Figure 16-2 Using ROI with a Neighborhood



The following example shows how to process an image with ROI:

```
IppStatus roi( void ) {
   Ipp8u x[ 8*3] = {0};
   IppiSize roiSize = {3,2};
   IppiPoint roiPoint = {2,1};
   // place the pointer to the ROI start position
   return ippiSet_8u_C1R( 7, x+8*roiPoint.y+roiPoint.x, 8, roiSize );
}
```

The destination image x will contain the following data

```
00 00 00 00 00 00 00 00
00 00 07 07 07 00 00 00
00 00 07 07 07 00 00 00
```

The rest of this chapter contains sub-sections that describe primitives of similar functionality. The description will be focused on each function's API with the C prototype and definition, the input/output arguments, return value, and notes to the definition. Each function set accompanies an example(s) of usage. The examples were compiled by the Microsoft* Embedded Visual Tools v 3.0 and run under Microsoft* Windows* CE environment.

Image Initialization Primitives

This section describes IPP image processing Primitives that perform image data set and initialization operations.

Set_8u_C1R

Prototype

```
IppStatus ippiSet_8u_C1R(Ipp8u value, Ipp8u*pDst, int dstStep, IppSize
   roiSize);
```

Description

Fills pixels in the ROI's `pSrc` of the size `roiSize` of one channel image with value.

Input Arguments

- `value` – constant value to set each pixel in the destination ROI
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal to zero
- `ippStsSizeErr` – illegal data size values

Set_8u_C3R

Prototype

```
IppStatus ippiSet_8u_C3R(const Ipp8u value[3], Ipp8u* pDst, int dstStep,
   IppiSize roiSize);
```

Description

Fills pixels with the corresponding number value[3] in the ROI region of the three–channel image.

Input Arguments

- `value` – constant value pointer to set each pixel in the destination ROI
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Example 16-2 Use of Set Function

```
#include "extools.h" /* The tool for print/display and listed in the
appendix*/
#include "ipp.h"

int MAIN()
{
   // Allocate enough space
   Ipp8u img[8*3] = {0};

   // The image is of size 5x3. But width 8 has been
   // chosen by the user to align every line of the image
   IppiSize imgSize = {5,3};
   ippiSet_8u_C1R(7, img, 8, imgSize);

   IppiSize showSize = {8,3};
   // print result of (8,3) size
   PrintROI_C1(_T("Use of the dstStep parameter"), img, 8, showSize);

   return 0;
}
```

Copy_8u_C1R
Copy_8u_C3R

Prototype

```
IppStatus ippiCopy_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst,
   int dstStep, IppiSize roiSize);
IppStatus ippiCopy_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst,
   int dstStep, IppiSize roiSize);
```

Description

Copy pixel values from the ROI of the source image pointed `pSrc` to the ROI of the destination image `pDst`.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

The following example describes a simple initialization routine.

Example 16-3 Initializing the Image

```
#include "extools.h" /* The tool for print/display and listed in the
appendix*/
#include "ipp.h"

int MAIN()
{
   // One-channel image
   const int SIDE = 32;
   Ipp8u img[SIDE*SIDE];

   // Specify ROI location and size
   IppiPoint roiLocation = {1,1};
   IppiSize roiSize = {7,7};
   // Initialize ROI by 0xF value
   ippiSet_8u_C1R(0xF, AddressOf(img,SIDE, roiLocation), SIDE, roiSize);

   // Print result
   IppiSize showSize = {9,9};
   PrintROI_C1(_T("Example: ippiSet"), img, SIDE, showSize);

   return 0;
}

//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
   return pImg +sLine*aLocation.y +aLocation.x;
}
```

The following example describes how to copy one image to another.

Example 16-4 Copying One Image to Another

```
#include "extools.h" /* The tool for print/display and listed in the
appendix*/
#include "ipp.h"

int MAIN()
{
   // One-channel source and destination images
   const int SIDE = 16;
   Ipp8u src[SIDE*SIDE];
   Ipp8u dst[SIDE*SIDE];
   IppiSize imgSize = {SIDE,SIDE};

   // Init source image
   ippiSet_8u_C1R(0xF, src,SIDE,imgSize);

   // Specify ROI location and size
   IppiPoint srcROIlocation = {2,2};
   IppiPoint dstROIlocation = {1,1};
   IppiSize  dstROIsize = {7,7};
   // Copy source ROI to the destination
   ippiCopy_8u_C1R(AddressOf(src,SIDE,srcROIlocation), SIDE,
                   AddressOf(dst,SIDE,dstROIlocation), SIDE, dstROIsize);

   // Print result
   IppiSize showSize = {9,9};
   PrintROI_C1(_T("Example: ippiCopy"), dst, SIDE, showSize);

   return 0;
}
//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
   return pImg +sLine*aLocation.y +aLocation.x;
}
```

Image Arithmetic and Logic Primitives

This section describes IPP image processing primitives that performs Arithmetic and Logic operations on an image.

Table 16-2 Arithmetic and Logic Definitions

Operation	Description
Add AddC	Adding operations with scaling and saturation: $z_{n,m} = SAT_{8u}((x_{n,m} + y_{n,m}) * 2^{-S})$ $z_{n,m} = SAT_{8u}((x_{n,m} + c) * 2^{-S})$
Sub SubC	Subtract operations with scaling and saturation: $z_{n,m} = SAT_{8u}((x_{n,m} - y_{n,m}) * 2^{-S})$ $z_{n,m} = SAT_{8u}((x_{n,m} - c) * 2^{-S})$
Mul MulC	Multiply operations with scaling and saturation: $z_{n,m} = SAT_{8u}((x_{n,m} * y_{n,m}) * 2^{-S})$ $z_{n,m} = SAT_{8u}((x_{n,m} * c) * 2^{-S})$
Sqr	Square operation with scaling and saturation: $z_{n,m} = SAT_{8u}((x_{n,m} * x_{n,m}) * 2^{-S})$
MulScale MulCScale	Scaled multiply operations: $z_{n,m} = (x_{n,m} * y_{n,m}) / 2^8$ $z_{n,m} = (x_{n,m} * c) / 2^8$
And AndC	Logical And operations: $z_{n,m} = x_{n,m} \,\&\, y_{n,m}$ $z_{n,m} = x_{n,m} \,\&\, c$
Or OrC	Logical Or operations: $z_{n,m} = x_{n,m} \mid y_{n,m}$ $z_{n,m} = x_{n,m} \mid c$

continued

Table 16-2 Arithmetic and Logic Definitions (continued)

Xor XorC	Logical Xor operations: $z_{n,m} = x_{n,m} \wedge y_{n,m}$ $z_{n,m} = x_{n,m} \wedge c$
Not	Logical Not operation: $z_{n,m} = \tilde{} x_{n,m}$
LShiftC	Logical Left Shift operation: $z_{n,m} = x_{n,m} * 2^{c}$
RShiftC	Logical Right Shift operation: $z_{n,m} = x_{n,m} * 2^{-c}$

Add_8u_C1RSfs
Mul_8u_C1RSf
Sub_8u_C1RSfs
Add_8u_C3RSfs
Mul_8u_C3RSfs
Sub_8u_C3RSf

Prototype

```
IppStatus ippiAdd_8u_C1RSfs(const Ipp8u* pSrc1, int src1Step, const
   Ipp8u* pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize
   roiSize, int scaleFactor);
IppStatus ippiMul_8u_C1RSfs(const Ipp8u* pSrc1, int src1Step, const
   Ipp8u* pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize
   roiSize, int scaleFactor);
IppStatus ippiSub_8u_C1RSfs(const Ipp8u* pSrc1, int src1Step, const
   Ipp8u* pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize
   roiSize, int scaleFactor);
```

```
IppStatus ippiAdd_8u_C3RSfs(const Ipp8u* pSrc1, int src1Step, const
   Ipp8u* pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize
   roiSize, int scaleFactor);
IppStatus ippiMul_8u_C3RSfs(const Ipp8u* pSrc1, int src1Step, const
   Ipp8u* pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize
   roiSize, int scaleFactor);
IppStatus ippiSub_8u_C3RSfs(const Ipp8u* pSrc1, int src1Step, const
   Ipp8u* pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize
   roiSize, int scaleFactor);
```

Description

Makes corresponding arithmetic operation with each element of two source images `pSrc1` and `pSrc2` and places the scaled result in the destination image `pDst`.

NOTE. *For Sub operation* `pSrc2` *is pointer to subtrahend,* `pSrc1` *is pointer to subtracter.*

Input Arguments

- `pSrc1` – pointer to the first source ROI (minuend)
- `src1Step` – step in bytes through the first source image
- `pSrc2` – pointer to the second source ROI (subtrahend)
- `src2Step` – step in bytes through the second source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `scaleFactor` – scale factor value

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

AddC_8u_C1RSfs
SubC_8u_C1RSf
MulC_8u_C1RSfs
AddC_8u_C3RSfs
SubC_8u_C3RSfs
MulC_8u_C3RSfs

Prototype

```
IppStatus ippiAddC_8u_C1RSfs(const Ipp8u* pSrc, int srcStep, Ipp8u value
   Ipp8u* pDst, int dstStep, IppiSize roiSize, int scaleFactor);
IppStatus ippiSubC_8u_C1RSfs(const Ipp8u* pSrc, int srcStep, Ipp8u value
   Ipp8u* pDst, int dstStep, IppiSize roiSize, int scaleFactor);
IppStatus ippiMulC_8u_C1RSfs(const Ipp8u* pSrc, int srcStep, Ipp8u value
   Ipp8u* pDst, int dstStep, IppiSize roiSize, int scaleFactor);
IppStatus ippiAddC_8u_C3RSfs(const Ipp8u* pSrc, int srcStep, const Ipp8u
   value[3] Ipp8u* pDst, int dstStep, IppiSize roiSize, int
   scaleFactor);
IppStatus ippiSubC_8u_C3RSfs(const Ipp8u* pSrc, int srcStep, const Ipp8u
   value[3] Ipp8u* pDst, int dstStep, IppiSize roiSize, int
   scaleFactor);
IppStatus ippiMulC_8u_C3RSfs(const Ipp8u* pSrc, int srcStep, const Ipp8u
   value[3] Ipp8u* pDst, int dstStep, IppiSize roiSize, int scaleFactor;
```

Description

Makes corresponding arithmetic operation with a constant and each element of image and places the scaled result in the same image.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `value` – constant for operation
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

- `scaleFactor` – scale factor value

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

MulScale_8u_C1R
MulScale_8u_C3R

Prototype

```
IppStatus ippiMulScale_8u_C1R(const Ipp8u* pSrc1, int src1Step, const
   Ipp8u* pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize
   roiSize);
IppStatus ippiMulScale_8u_C3R(const Ipp8u* pSrc1, int src1Step, const
   Ipp8u* pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize
   roiSize);
```

Description

Makes scaled multiply operation with each elements of two source images and places the result in the destination image.

Input Arguments

- `pSrc1` – pointer to the first source ROI
- `src1Step` – step in bytes through the first source image
- `pSrc2` – pointer to the second source ROI
- `src2Step` – step in bytes through the second source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

MulCScale_8u_C1R
MulCScale_8u_C3R

Prototype

```
IppStatus ippiMulCScale_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u
   value, Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiMulCScale_8u_C3R(const Ipp8u* pSrc, int srcStep, const
   Ipp8u value[3], Ipp8u* pDst, int dstStep, IppiSize roiSize);
```

Description

Makes scaled multiply operation with a constant and each element of image and places the scaled result in the same image.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `value` – constant for operation
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Sqr_8u_C1RSf
Sqr_8u_C3RSfs

Prototype

```
IppStatus ippiSqr_8u_C1RSfs(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst,
   int dstStep, IppiSize roiSize, int scaleFactor);
IppStatus ippiSqr_8u_C3RSfs(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst,
   int dstStep, IppiSize roiSize, int scaleFactor);
```

Description

Performs a square operation on each element of the input image and places the scaled result in the output image.

Input Arguments

- pSrc – pointer to the source ROI
- srcStep – step in bytes through the source image
- dstStep – step in bytes through the destination image
- roiSize – size of the source and destination ROI in pixels
- scaleFactor – scale factor value

Output Arguments

pDst – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error

- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Example 16-5 Use of Arithmetic Function

```
#include "extools.h" /* The tool for print/display and listed in the appendix*/
#include "ipp.h"

int MAIN()
{
   // One-channel source and destination images
   const int SIDE = 16;
   Ipp8u src[SIDE*SIDE];
   Ipp8u dst[SIDE*SIDE];
   IppiSize imgSize = {SIDE,SIDE};

   // Init source image
   ippiSet_8u_C1R(0x05, src,SIDE,imgSize);

 // Specify ROI location and size
   IppiPoint srcLocation = {2,2};
   IppiPoint dstLocation = {1,1};
   IppiSize roiSize = {7,7};
```

Example 16-5 Use of Arithmetic Function (continued)

```
   // Inverse part of the src image and place the result to dst
   ippiNot_8u_C1R(AddressOf(src,SIDE,srcLocation), SIDE,
                  AddressOf(dst,SIDE,dstLocation), SIDE, roiSize);

   // Print result
   IppiSize showSize = {9,9};
   PrintROI_C1(_T("Example: ippiNot"), dst, SIDE, showSize);

   return 0;
}
//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
   return pImg +sLine*aLocation.y +aLocation.x;
}
```

And_8u_C1R
Or_8u_C1R
Xor_8u_C1R
Not_8u_C1R
And_8u_C3R
Or_8u_C3R
Xor_8u_C3R
Not_8u_C3R

Prototype

```
IppStatus ippiAnd_8u_C1R(const Ipp8u* pSrc1, int src1Step, const Ipp8u*
   pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiOr_8u_C1R(const Ipp8u* pSrc1, int src1Step, const Ipp8u*
   pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiXor_8u_C1R(const Ipp8u* pSrc1, int src1Step, const Ipp8u*
   pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiNot_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst, int
   dstStep, IppiSize roiSize)
IppStatus ippiAnd_8u_C3R(const Ipp8u* pSrc1, int src1Step, const Ipp8u*
   pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiOr_8u_C3R(const Ipp8u* pSrc1, int src1Step, const Ipp8u*
   pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiXor_8u_C3R(const Ipp8u* pSrc1, int src1Step, const Ipp8u*
   pSrc2, int src2Step, Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiNot_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst, int
   dstStep, IppiSize roiSize);
```

Description

Makes corresponding logic operations between corresponding elements of two source images and places the scaled result in the destination image at the corresponding location.

Input Arguments

- `pSrc1` – pointer to the first source ROI

- src1Step – step in bytes through the first source image
- pSrc2 – pointer to the second source ROI
- src2Step – step in bytes through the second source image
- dstStep – step in bytes through the destination image
- roiSize – size of the source and destination ROI in pixels

Output Arguments

pDst – pointer to the destination ROI

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointer error
- ippStsStepErr – step value is less or equal zero
- ippStsSizeErr – illegal data size values

AndC_8u_C1R
OrC_8u_C1R
XorC_8u_C1R
AndC_8u_C3R
OrC_8u_C3R
XorC_8u_C3R

Prototype

```
IppStatus ippiAndC_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u* value,
   Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiOrC_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u* value,
   Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiXorC_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u* value,
   Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiAndC_8u_C3R(const Ipp8u* pSrc, int srcStep, const Ipp8u*
   value[3], Ipp8u* pDst, int dstStep, IppiSize roiSize);
```

```
IppStatus ippiOrC_8u_C3R(const Ipp8u* pSrc, int srcStep, const Ipp8u*
   value[3], Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiXorC_8u_C3R(const Ipp8u* pSrc, int srcStep, const Ipp8u*
   value[3], Ipp8u* pDst, int dstStep, IppiSize roiSize);
```

Description

Makes corresponding logic operation between the constant and each element of image and places the scaled result in the destination image at the corresponding location.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `value` – constant for operation
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – wrong value of data size

LShiftC_8u_C1R
RShiftC_8u_C1R
LShiftC_8u_C3R
RShiftC_8u_C3R

Prototype

```
IppStatus ippiLShiftC_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u value,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiRShiftC_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u value,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiLShiftC_8u_C3R(const Ipp8u* pSrc, int srcStep, const Ipp8u
   value[3], Ipp8u* pDst, int dstStep, IppiSize roiSize);
IppStatus ippiRShiftC_8u_C3R(const Ipp8u* pSrc, int srcStep, const Ipp8u
   value[3], Ipp8u* pDst, int dstStep, IppiSize roiSize);
```

Description

Makes a logical shift operation (left or right) on each element of the image and places the result in the destination image.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `value` – constant for operation
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero

- `ippStsSizeErr` – illegal data size values

Image Filtering Primitives

This section describes IPP image processing Primitives that performs linear and non–linear filtering operations on an image. Table 16-3 and Table 16-4 provide detailed mathematical definitions of all filter operations support by IPP.

Table 16-3 FIR Filtering Definitions

Operation	Description
Pure row (horizontal) filtering	Filters an image using row (horizontal) convolution kernel. Basic equation is the following: $y_{n,m} = \Psi_{row}(h,a,x) \equiv \sum_{i=0}^{W-1} h_i \cdot x_{n,m+a-i}$ where: h kernel of width W with anchor location a $x_{n,m}$ and $y_{n,m}$ are input and output signals respectively
Pure column (vertical) filtering	Filters an image using column (vertical) convolution kernel. Basic equation is the following: $y_{n,m} = \Psi_{column}(h,a,x) \equiv \sum_{i=0}^{H-1} h_i \cdot x_{n+a-i,m}$ where: h kernel of height H width anchor location a $x_{n,m}$ and $y_{n,m}$ are input and output signals respectively
Simple blur filtering	Blurs an image using simple box filter (applies an average filter). Basic equation of average filter is the following: $y_{n,m} = \Psi_{box}(W,H,x) \equiv \frac{1}{H \cdot W} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} x_{n+ay-i,m+ax-j}$ where: mask (aperture) has $W \times H$ size with anchor location (ay, ax) $x_{n,m}$ and $y_{n,m}$ are input and output signals respectively

Table 16-3 FIR Filtering Definitions (continued)

General FIR filtering	Filters an image using a general rectangular convolution kernel. Basic equation of general 2D filter is the following: $y_{n,m} = \Psi_{2D}(h,a,x) \equiv \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} h_{i,j} \cdot x_{n+ay-i,m+ax-j}$ where: h 2D kernel of $W \times H$ size with anchor location (ay, ax) $x_{n,m}$ and $y_{n,m}$ are input and output signals respectively

Table 16-4 Median Filtering Definitions

Operation	Description
single–channel (gray) image	Filters an image using a median filter. Basic equation is the following: $y_{n,m} = x_{n-ay+r,m-ax+c} \equiv \underset{(W \times H)}{MED}(a,x)$ $c = 0,\ldots,W-1,$ $r = 0,\ldots,H-1$ if $\Psi = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \left\| x_{n-ay+r,m-ax+c} - x_{n-ay+i,m-ax+j} \right\|$ has minimum value where: $W \times H$ is the (aperture) size of median filter width anchor location (ay, ax) $x_{n,m}$ and $y_{n,m}$ are input and output signals correspondingly
multi–channel image	Filters multi–channel image separately (channel–by–channel) using a median filter. Basic equation is the following: $y_{chan} = \underset{(W \times H)}{MED}(a, x_{chan})$ x_{chan} and y_{chan} are channel slice of input and output signals correspondingly.

Table 16-4 Median Filtering Definitions (continued)

"color" filtering	Filters an image using a "color" median filter. Basic equation is the following: $y_{n,m} = x_{n-ay+r,m-ax+c} \equiv \underset{(W\times H)}{MEDC}(a,x)$ $c = 0,\ldots,W-1,$ $r = 0,\ldots,H-1$ if $\Psi = \sum_{chan=0}^{nChans-1}\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}\left\lvert x_{(chan)n-ay+r,m-ax+c} - x_{(chan)n-ay+i,m-ax+j}\right\rvert$ has minimum value where: $W \times H$ is the (aperture) size of median filter width anchor location (ay, ax) $x_{(chan)n,m}$ and $y_{(chan)n,m}$ are channel slice input and output signals correspondingly

FilterBox_8u_C1R
FilterBox_8u_C3R

Prototype

```
IppStatus ippiFilterBox_8u_C1R(const Ipp8u* pSrc,int srcStep, Ipp8u*
   pDst,int dstStep,IppiSize roiSize, IppiSize maskSize, IppiPoint
   anchor);
IppStatus ippiFilterBox_8u_C3R(const Ipp8u* pSrc,int srcStep, Ipp8u*
   pDst,int dstStep,IppiSize roiSize, IppiSize maskSize, IppiPoint
   anchor);
```

Description

Perform blurring (applies an average filter) of the ROI of the source image pointed to by `pSrc` using the simple box filter of size maskSize and location anchor. Place the result into the ROI of the destination image pointed to by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI

- srcStep – step in bytes through the source image
- dstStep – step in bytes through the destination image
- roiSize – size of the source and destination ROI in pixels
- maskSize – size of the mask in pixels
- anchor – anchor cell specifying the mask alignment with respect to the position of the input pixel

Output Arguments

pDst – pointer to the destination ROI

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointer error
- ippStsStepErr – step value is less or equal zero
- ippStsSizeErr – illegal data size values
- ippStsMaskSizeErr – invalid mask size
- ippStsAnchorErr – the anchor point is outside mask

FilterColumn_8u_C1R
FilterColumn_8u_C3R

Prototype

```
IppStatus ippiFilterColumn_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, const Ipp32s* pKernel, int
   kernelSize, int anchor, int divider);
IppStatus ippiFilterColumn_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, const Ipp32s* pKernel, int
   kernelSize, int anchor, int divider);
```

Description

Filters an ROI in the source image using an integer column convolution kernel. The value of the output pixel is calculated as:

$$SAT_{8u}\left(\frac{1}{divider}\Psi_{column}(h,a,x)\right)$$

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `pKernel` – pointer to the FIR filter taps
- `kernelSize` – length of the FIR filter
- `anchor` – anchor cell specifying the array of filter taps alignment with respect to the position of the input pixel
- `divider` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `ippStsAnchorErr` – the anchor point is outside mask
- `ippStsScaleRangeErr` – scale bounds is out of range
- `ippStsMemAllocErr` – not enough memory for the operation

FilterRow_8u_C1R
FilterRow_8u_C3R

Prototype

```
IppStatus ippiFilterRow_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, const Ipp32s* pKernel, int
   kernelSize, int anchor, int divider);
IppStatus ippiFilterRow_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, const Ipp32s* pKernel, int
   kernelSize, int anchor, int divider);
```

Description

Performs filtering of the ROI of the source image pointed to by pSrc using an integer row (Wx1 size) convolution kernel. The value of the output pixel is normalized by the divider and saturated. This can be calculated as:

$$SAT_{8u}\left(\frac{1}{divider}\Psi_{row}(h,a,x)\right)$$

Place the result into the ROI of the destination image pointed to by `pDst`.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `pDst` – pointer to the destination ROI
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `pKernel` – pointer to the FIR filter taps
- `kernelSize` – length of the FIR filter
- `anchor` – anchor cell specifying the array of filter taps alignment with respect to the position of the input pixel
- `divider` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `ippStsAnchorErr` – the anchor point is outside mask
- `ippStsScaleRangeErr` – scale bounds is out of range
- `ippStsMemAllocErr` – not enough memory for the operation

Filter_8u_C1R
Filter_8u_C3R

Prototype

```
IppStatus ippiFilter_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst,
   int dstStep, IppiSize roiSize, const Ipp32s* pKernel, IppiSize
   kernelSize, IppiPoint anchor, int divider);
IppStatus ippiFilter_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst,
   int dstStep, IppiSize roiSize, const Ipp32s* pKernel, IppiSize
   kernelSize, IppiPoint anchor, int divider);
```

Description

Performs filtering of the ROI of the source image pointed to by `pSrc` using a general rectangular (WxH size) convolution kernel. The value of the output pixel is normalized by the divider and saturated as:

$$SAT_{8u}\left(\frac{1}{divider}\Psi_{2D}(h,a,x)\right)$$

Place the result into the ROI of the destination image pointed to by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image

- `roiSize` – size of the source and destination ROI in pixels
- `pKernel` – pointer to the 2D FIR filter taps
- `kernelSize` – size of the FIR filter
- `anchor` – anchor cell specifying the array of filter taps alignment with respect to the position of the input pixel
- `divider` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `ippStsAnchorErr` – the anchor point is outside mask
- `ippStsScaleRangeErr` – scale bounds is out of range
- `ippStsMemAllocErr` – not enough memory for the operation

Example 16-6 Use FIR Filters

```
#include <stdlib.h>
#include "extools.h" /* The tool for print/display and listed in
the appendix*/
#include "ipp.h"

int MAIN()
{
   // One-channel source and destination images
   const int SIDE = 16;
   Ipp8u src[SIDE*SIDE];
   Ipp8u dst[SIDE*SIDE];

   // Fill source image by any values
   for(int i=0; i<SIDE*SIDE; i++) src[i] = (Ipp8u)(rand()&0xF);

   // 3x3 filter with anchor at the middle
   const int FSIDE  = 3;
   IppiSize filterSize = {FSIDE,FSIDE};
   IppiPoint anchor = {1,1};
   Ipp32s filter[FSIDE][FSIDE] = {
      {1,1,1},
      {1,1,1},
      {1,1,1}
   };

   // Specify ROI location and size
   IppiPoint srcLocation = {1,1};
   IppiPoint dstLocation = {0,0};
   IppiSize roiSize = {7,7};
   // Apply 2D averaging FIR Filter
   ippiFilter_8u_C1R(AddressOf(src,SIDE,srcLocation),SIDE,
                     AddressOf(dst,SIDE,dstLocation),SIDE,roiSize,
                      (Ipp32s*)filter,filterSize,anchor,
FSIDE*FSIDE);
```

Example 16-6 Use FIR Filters

```
    // Print result
    IppiSize showSize = {9,9};
    PrintROI_C1(_T("Example: ippiFilter (source)"),      src, SIDE,
showSize);
    PrintROI_C1(_T("Example: ippiFilter (destination)"), dst, SIDE,
roiSize);

    return 0;
}

//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
    return pImg +sLine*aLocation.y +aLocation.x
```

FilterMedian_8u_C1R
FilterMedian_8u_C3R

Prototype

```
IppStatus ippiFilterMedian_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, IppiSize maskSize, IppiPoint
   anchor);
IppStatus ippiFilterMedian_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, IppiSize maskSize, IppiPoint
   anchor);
```

Description

Performs median filtering of the ROI of the source image pointed to by `pSrc` using the median filter of the size `maskSize` and location `anchor`. Place the result into the ROI of the destination image pointed to by `pDst`.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `maskSize` – size of the mask in pixels
- `anchor` – anchor cell specifying the mask alignment with respect to the position of the input pixel

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `ippStsMaskSizeErr` – invalid mask size
- `ippStsAnchorErr` – the anchor point is outside mask

FilterMedianColor_8u_C3R

Prototype

```
IppStatus ippiFilterMedianColor_8u_C3R(const Ipp8u* pSrc, int srcStep,
   Ipp8u* pDst, int dstStep, IppiSize roiSize, IppiMaskSize mask);
```

Description

Performs “color” median filtering of the ROI of the source image pointed to by `pSrc` using Color Median Filter of predefined size `mask`. Place the result into the ROI of the destination image pointed to by `pDst`.

The `IppiMaskSize` enumeration defines the neighborhood area for Color Median Filter function:

```
typedef enum {
```

```
   mskSize1x3 = 13,
   mskSize1x5 = 15,
   mskSize3x1 = 31,
   mskSize3x3 = 33,
   mskSize5x1 = 51,
   mskSize5x5 = 55
} IppiMaskSize;
```

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `mask` – size of the predefined mask in pixels

Output Arguments

- `pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `ippStsMaskSizeErr` – invalid mask size

Linear Transform Primitives

This section describes the IPPs that perform 2D linear image Fast Fourier Transform (FFT) and Discrete Cosine Transform (DCT) transforms that are implemented in the library.

Table 16-5 Linear Transform Definitions

Operation	Description

Table 16-5 Linear Transform Definitions (continued)

Forward DFT of a real image	The basic forward DFT equation is the following: $X_{uv} = FFT(x_{nm}) \equiv K_{fwd} \cdot \sum_{n=0}^{H-1} \sum_{m=0}^{W-1} x_{nm} \cdot e^{-j\frac{2\pi}{H}nu} e^{-j\frac{2\pi}{W}mv}$ $u = 0,\ldots,H-1, v = 0,\ldots,W-1$
Inverse FFT to a real image	The basic inverse FFT equation is the following: $x_{nm} = FFT^{-1}(X_{uv}) \equiv K_{inv} \cdot \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} X_{uv} \cdot e^{j\frac{2\pi}{H}un} e^{j\frac{2\pi}{W}vm}$ $m = 0,\ldots,H-1, n = 0,\ldots,W-1$
Forward DCT of an image	The basic forward DCT equation is the following: $X_{uv} = \frac{2 \cdot c(u)c(v)}{\sqrt{H \cdot W}} \sum_{m=0}^{H-1} \sum_{n=0}^{W-1} x_{mn} \cdot \cos\left[\frac{(2m+1)u\pi}{2H}\right] \cdot \cos\left[\frac{(2n+1)v\pi}{2W}\right]$ $u = 0,\ldots,H-1, v = 0,\ldots,W-1$ $c(k) = \frac{1}{\sqrt{2}}$ if $k = 0$ otherwise $= 0$
Inverse DCT of an image	The basic inverse DCT equation is the following: $x_{mn} = \frac{2 \cdot}{\sqrt{H \cdot W}} \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} c(u) \cdot c(v) \cdot X_{uv} \cdot \cos\left[\frac{(2m+1)u\pi}{2H}\right] \cdot \cos\left[\frac{(2n+1)v\pi}{2W}\right]$ $m = 0,\ldots,H-1, n = 0,\ldots,W-1$ $c(k) = \frac{1}{\sqrt{2}}$ if $k = 0$ otherwise $= 0$

Fast Fourier Transform Data Forma

The FFT of any real 2D signal is conjugate–symmetric. Therefore, it can be fully specified by storing only half the output data. A special format called Pack is provided for this purpose.

The ippiFFTFwd_RtoPack() function transforms an image and produces the Fourier coefficients in the Pack format. To complement this, function `ippiFFTInv_PackToR( )` uses its input in Pack format, and produces its output as a real–valued image.

Fourier coefficients have the following relationship:

$$X_{0,v} = conj\left(X_{0,W-v}\right)$$

$$X_{u,0} = conj\left(X_{H-u,0}\right)$$

$$X_{u,v} = conj\left(X_{H-u,W-v}\right), u = 1,...,H-1, v = 1,...,W-1$$

To reconstruct the $W \times H$ complex coefficients X_{uv}, it is enough to store only $W \times H$ real values. The Fourier transform primitives actually use $u = 0,...,H-1, v = 0,...,W/2$. Other Fourier coefficients are founded using complex–conjugate relations.

Table 16-6 and Table 16-7 show the Pack format arrangement:

- Re corresponds to Real
- Im corresponds to Imaginary parts. Table 16-8 is an example of output samples storage for H =4 and W =4.

Table 16-6 ippFFTFwd_RToPack() Output in Pack Format (even H)

$\mathrm{Re}\,X_{0,0}$	$\mathrm{Re}\,X_{0,1}$	$\mathrm{Im}\,X_{0,1}$	⋯	$\mathrm{Re}\,X_{0,(W-1)/2}$	$\mathrm{Im}\,X_{0,(W-1)/2}$	$\mathrm{Re}\,X_{0,W/2}$
$\mathrm{Re}\,X_{1,0}$	$\mathrm{Re}\,X_{1,1}$	$\mathrm{Im}\,X_{1,1}$	⋯	$\mathrm{Re}\,X_{1,(W-1)/2}$	$\mathrm{Im}\,X_{1,(W-1)/2}$	$\mathrm{Re}\,X_{1,W/2}$
$\mathrm{Im}\,X_{1,0}$	$\mathrm{Re}\,X_{2,1}$	$\mathrm{Im}\,X_{2,1}$	⋯	$\mathrm{Re}\,X_{2,(W-1)/2}$	$\mathrm{Im}\,X_{2,(W-1)/2}$	$\mathrm{Im}\,X_{1,W/2}$
.....	.....	.....	...	.....	.....	.....
$\mathrm{Re}\,X_{H/2-1,0}$	$\mathrm{Re}\,X_{H-3,1}$	$\mathrm{Im}\,X_{H-3,1}$	⋯	$\mathrm{Re}\,X_{H-3,(W-1)/2}$	$\mathrm{Im}\,X_{H-3,(W-1)/2}$	$\mathrm{Re}\,X_{H/2-1,W/2}$
$\mathrm{Im}\,X_{H/2-1,0}$	$\mathrm{Re}\,X_{H-2,1}$	$\mathrm{Im}\,X_{H-2,1}$	⋯	$\mathrm{Re}\,X_{H-2,(W-1)/2}$	$\mathrm{Im}\,X_{H-2,(W-1)/2}$	$\mathrm{Im}\,X_{H/2-1,W/2}$
$\mathrm{Re}\,X_{H/2,0}$	$\mathrm{Re}\,X_{H-1,1}$	$\mathrm{Im}\,X_{H-1,1}$	⋯	$\mathrm{Re}\,X_{H-1,(W-1)/2}$	$\mathrm{Im}\,X_{H-1,(W-1)/2}$	$\mathrm{Re}\,X_{H/2,W/2}$

Table 16-7 ippiFFTFwd_RToPack() Output in Pack Format (odd H)

$\mathrm{Re}\,X_{0,0}$	$\mathrm{Re}\,X_{0,1}$	$\mathrm{Im}\,X_{0,1}$	…	$\mathrm{Re}\,X_{0,(W-1)/2}$	$\mathrm{Im}\,X_{0,(W-1)/2}$	$\mathrm{Re}\,X_{0,W/2}$
$\mathrm{Re}\,X_{1,0}$	$\mathrm{Re}\,X_{1,1}$	$\mathrm{Im}\,X_{1,1}$	…	$\mathrm{Re}\,X_{1,(W-1)/2}$	$\mathrm{Im}\,X_{1,(W-1)/2}$	$\mathrm{Re}\,X_{1,W/2}$
$\mathrm{Im}\,X_{1,0}$	$\mathrm{Re}\,X_{2,1}$	$\mathrm{Im}\,X_{2,1}$	…	$\mathrm{Re}\,X_{2,(W-1)/2}$	$\mathrm{Im}\,X_{2,(W-1)/2}$	$\mathrm{Im}\,X_{1,W/2}$
……	……	……	…	……	……	……
$\mathrm{Re}\,X_{H/2,0}$	$\mathrm{Re}\,X_{H-2,1}$	$\mathrm{Im}\,X_{H-2,1}$	…	$\mathrm{Re}\,X_{H-2,(W-1)/2}$	$\mathrm{Im}\,X_{H-2,(W-1)/2}$	$\mathrm{Re}\,X_{H/2,W/2}$
$\mathrm{Im}\,X_{H/2,0}$	$\mathrm{Re}\,X_{H-1,1}$	$\mathrm{Im}\,X_{H-1,1}$	…	$\mathrm{Re}\,X_{H-1,(W-1)/2}$	$\mathrm{Im}\,X_{H-1,(W-1)/2}$	$\mathrm{Im}\,X_{H/2,W/2}$

Table 16-8 is an example of output samples storage for H =4 and W =4.

Table 16-8 ippiFFTFwd_RToPack() Output in Pack Format (H=4,W=4)

$\mathrm{Re}\,X_{0,0}$	$\mathrm{Re}\,X_{0,1}$	$\mathrm{Im}\,X_{0,1}$	$\mathrm{Re}\,X_{0,2}$
$\mathrm{Re}\,X_{1,0}$	$\mathrm{Re}\,X_{1,1}$	$\mathrm{Im}\,X_{1,1}$	$\mathrm{Re}\,X_{1,2}$
$\mathrm{Im}\,X_{1,0}$	$\mathrm{Re}\,X_{2,1}$	$\mathrm{Im}\,X_{2,1}$	$\mathrm{Im}\,X_{1,2}$
$\mathrm{Re}\,X_{20}$	$\mathrm{Re}\,X_{3,1}$	$\mathrm{Im}\,X_{3,1}$	$\mathrm{Re}\,X_{2,2}$

FFTGetSpecSize_R_8u

Prototype

```
IppStatus ippiFFTGetSpecSize_R_8u(int xOrder, int yOrder,
   IppHintAlgorithm hint, int *pSize);
```

Description

Gets size of FFT specification structure (that is, special structure for FFT calculations) in bytes. FFT will be `xOrder` in X direction, `yOrder` in Y direction. This returns the FFT specification structure size in `pSize`.

Input Arguments

- `xOrder` – the FFT order in X direction. Specify the input signal width $W = 2^{xOrder}$. (Implementation limit: xOrder<=10)
- `yOrder` – the FFT order in Y direction. Specify the input signal height $H = 2^{yOrder}$. (Implementation limit: yOrder<=10)

Output Arguments

`pSize` – pointer to the size of FFT specification structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsFftOrderErr` – invalid value of the FFT order parameter

FFTInit_R_8u

Prototype

```
IppStatus ippiFFTInit_R_8u(IppiFFTSpec_R_8u* pSpec, int xOrder, int
   yOrder, int flag, IppHintAlgorithm hint);
```

Description

Initializes FFT specification structure (that is, special structure for FFT calculations). FFT will be `xOrder` in X direction, `yOrder` in Y direction. The result will depend on the `flag` value. This returns the FFT specification structure address in `pSpec`.

Input Arguments

- `xOrder` – the FFT order in X direction. Specify the input signal width $W = 2^{xOrder}$. (Implementation limit: xOrder<=10)

- `yOrder` – the FFT order in Y direction. Specify the input signal height $H = 2^{yOrder}$. (Implementation limit: yOrder<=10)
- flag – specifies the method of the result normalization. The results normalizing depend on flag value:

	K_{fwd}	K_{inv}
IPP_FFT_DIV_BY_SQRT_N	$\frac{1}{\sqrt{W \cdot H}}$	$\frac{1}{\sqrt{W \cdot H}}$
IPP_FFT_DIV_FWD_BY_N	$\frac{1}{W \cdot H}$	1
IPP_FFT_DIV_INV_BY_N	1	$\frac{1}{W \cdot H}$
IPP_FFT_NODIV_BY_ANY	1	1

Recommend using a specific code for the transform

Output Arguments

`pSpec` – Address of pointer to the created 2D FFT specification structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsFftOrderErr` – invalid value of the FFT order parameter

FFTInitAlloc_R_8u

Prototype

```
IppStatus ippiFFTInitAlloc_R_8u(IppiFFTSpec_R_8u** pSpec, int xOrder,
   int yOrder, int flag, IppHintAlgorithm hint);
```

Description

Allocates and initializes FFT specification structure (that is, special structure for FFT calculations). FFT will be `xOrder` in X direction, `yOrder` in Y direction. The result will depend on the `flag` value. This returns the FFT specification structure address in `pSpec`.

Input Arguments

- `xOrder` – the FFT order in X direction. Specify the input signal width $W = 2^{xOrder}$. (Implementation limit: xOrder<=10)
- `yOrder` – the FFT order in Y direction. Specify the input signal height $H = 2^{yOrder}$. (Implementation limit: yOrder<=10)
- `flag` – specifies the method of the result normalization. The results normalizing depend on flag value:

	K_{fwd}	K_{inv}
IPP_FFT_DIV_BY_SQRT_N	$\frac{1}{\sqrt{W \cdot H}}$	$\frac{1}{\sqrt{W \cdot H}}$
IPP_FFT_DIV_FWD_BY_N	$\frac{1}{W \cdot H}$	1
IPP_FFT_DIV_INV_BY_N	1	$\frac{1}{W \cdot H}$
IPP_FFT_NODIV_BY_ANY	1	1

Output Arguments

`pSpec` – address of pointer to the created 2D FFT specification structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsFftOrderErr` – invalid value of the FFT order parameter
- `ippStsMemAllocErr` – not enough memory for the operation

FFTFree_R_8u

Prototype

```
IppStatus ippiFFTFree_R_8u(IppiFFTSpec_R_8u* pSpec);
```

Description

Release memory previously allocated for FFT specification structure with `ippiFFTInitAlloc_R_8u` function.

Input Arguments

`pSpec` – pointer to the FFT specification structure

Output Arguments

None

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsContextMatchErr` – the context parameter does not match the operation

FFTGetBufSize_R_8u

Prototype

```
IppStatus ippiFFTGetBufSize_R_8u(const IppiFFTSpec_R_8u* pSpec, int*
   pSize);
```

Description

Get the size of the FFT work buffer in bytes.

Input Arguments

`pSpec` – pointer to the FFT specification structure

Output Arguments

`pSize` – pointer to the FFT work buffer size value

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsContextMatchErr` – the context parameter doesn't match to the operation

FFTFwd_RToPack_8u32s_C1R
FFTFwd_RToPack_8u32s_C3R

Prototype

```
IppStatus ippiFFTFwd_RToPack_8u32s_C1R(const Ipp8u* pSrc, int srcStep,
   Ipp32s* pDst, int dstStep, const IppiFFTSpec_R_8u* pSpec, int
   scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippiFFTFwd_RToPack_8u32s_C3R(const Ipp8u* pSrc, int srcStep,
   Ipp32s* pDst, int dstStep, const IppiFFTSpec_R_8u* pSpec, int
   scaleFactor, Ipp8u* pBuffer);
```

Description

Computes the forward Fast Fourier Transform of a real signal. Place the scaled result into the ROI of the destination image pointed to by pDst. All general parameters (for example, FFT order and result normalization type) should be defined before the call.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `pDst` – pointer to the destdstStep – step in bytes through the destination image
- `pSpec` – pointer to the FFT specification structure
- `scaleFactor` – output scale factor value
- `pBuffer` – pointer to the work buffer

Output Arguments

pDst – pointer to the destination ROI. Function performs output values according to $FFT(x) \cdot 2^{-scaleFactor}$ Results are in Pack format.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – null pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsContextMatchErr` – the context parameter doesn't match to the operation

FFTInv_PackToR_32s8u_C1RSfs
FFTInv_PackToR_32s8u_C3RSfs

Prototype

```
IppStatus ippiFFTInv_PackToR_32s8u_C1RSfs(const Ipp8u* pSrc, int
   srcStep, Ipp32s* pDst, int dstStep, const IppiFFTSpec_R_8u* pSpec,
   int scaleFactor, Ipp8u* pBuffer);
IppStatus ippiFFTInv_PackToR_32s8u_C3RSfs(const Ipp8u* pSrc, int
   srcStep, Ipp32s* pDst, int dstStep, const IppiFFTSpec_R_8u* pSpec,
   int scaleFactor, Ipp8u* pBuffer);
```

Description

Computes the inverse Fast Fourier transform to a real signal. Place the scaled and saturated result into the ROI of the destination image pointed to by pDst. All general parameters (for example, FFT order and result normalization types) should be defined before they are called.

Input Arguments

- `pSrc` – pointer to the source ROI. Input data are in Pack format
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `pSpec` – pointer to the FFT specification structure
- `scaleFactor` – output scale factor value
- `pBuffer` – pointer to the work buffer

Output Arguments

`pDst` – pointer to the destination ROI. Function performs output values according to:

$$SAT_{8u}\left(FFT^{-1}(X)\cdot 2^{-scaleFactor}\right)$$

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – null pointer error
- `ippStsStepErr` – step value is less or equal zero

- `ippStsContextMatchErr` – the context parameter doesn't match to the operation

Example 16-7 describes all steps required for the Fourier Transform of an image.

Example 16-7 Use of FFT Primitives

```
*_____________________________________________
// Result of ippiFFTFwd_RToPack_8u32s_C1R function:
// 89 10  -7 -9
// -1  8 -21 13
//  6 10   1  2
// -3 -8   3  3
//

int MAIN()
{
   // One-channel source and destination images
   const int SIDE = 4;
   const Ipp8u src[SIDE][SIDE] = {
      {9, 7, 4, 1},
      {7, 5, 1, 7},
      {6, 6, 1, 9},
      {3,10, 9, 4}
   };
   Ipp32s dst[SIDE][SIDE];
   IppiSize imgSize = {SIDE,SIDE};

  // Allocate and initialize FFT Specification
   IppiFFTSpec_R_8u* pSpec;
   ippiFFTInitAlloc_R_8u(&pSpec, 2,2,IPP_FFT_DIV_INV_BY_N,
ippAlgHintNone);
```

Example 16-7 Use of FFT Primitives (continued)

```
    // Get length of the temporary buffer and allocate one
     int bufSize;
     ippiFFTGetBufSize_R_8u(pSpec, &bufSize);
     Ipp8u* pBuffer = ippsMalloc_8u(bufSize);

     // Forward FFT
     ippiFFTFwd_RToPack_8u32s_C1R((Ipp8u*) src, SIDE,
                                  (Ipp32s*)dst, SIDE*sizeof(Ipp32s),
                                  pSpec, 0, pBuffer);

     // Print result
     PrintROI_C1(_T("Example: ippiFFT"), (Ipp32s*)dst, SIDE, imgSize);

     // Release temporary buffer and FFT Specification
     ippsFree(pBuffer);
     ippiFFTFree_R_8u(pSpec);

     return 0;
}
//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
     return pImg +sLine*aLocation.y +aLocation.x;
}
```

DCT8x8Fwd_16s_C1

Prototype

```
IppStatus ippiDCT8x8Fwd_16s_C1(const Ipp16s* pSrc, Ipp16s* pDst);
```

Description

Performs a forward DCT on a 2D array pSrc of 8x8 size, and outputs the results to pDst.

Input Arguments

`pSrc` – pointer to the source ROI

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error

DCT8x8Inv_16s_C1

Prototype

```
IppStatus ippiDCT8x8Inv_16s_C1(const Ipp16s* pSrc, Ipp16s* pDst);
```

Description

Performs an inverse DCT on a 2D array pSrc of 8x8 size, and outputs the results to pDst.

Input Arguments

pSrc – pointer to the source ROI

Output Arguments

pDst – pointer to the destination ROI

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointer error

Image Color Model Conversion Primitives

This section describes IPP image processing primitives perform color model conversion operations on an image. Table 16-9 presents mathematical descriptions of the color conversions primitives that are presented in this section. Color conversion subsampling conventions are summarized in Table 16-10.

Table 16-9 Color Model Conversions

Functions	Mathematical Descriptions
Color Twist	Linear transform of an original Color model (RGB) to another (ABC) by user defined transformation T MT_0 $$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} T_{00}T_{01}T_{02} \\ T_{10}T_{11}T_{12} \\ T_{20}T_{21}T_{22} \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} T_{03} \\ T_{13} \\ T_{23} \end{bmatrix}$$ Where $T = \begin{bmatrix} T_{00}T_{01}T_{02} \\ T_{10}T_{11}T_{12} \\ T_{20}T_{21}T_{22} \end{bmatrix}$ and $T_0 = \begin{bmatrix} T_{03} \\ T_{13} \\ T_{23} \end{bmatrix}$
RGB to XYZ	RGB Color model transformed to XYZ Color model as follows (reference to Color Twist): Uses $T = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix}$ and $T_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$
XYZ to RGB	XYZ Color model transformed to RGB Color model as follows (reference to Color Twist): Uses $T = \begin{bmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix}$ and $T_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$

Table 16-9 Color Model Conversions (continued)

RGB to YCC	Normalized (values of 0 to 1) RGB Color model transformed to YCC Color model as follows: $luma = 0.299R' + 0.587G' + 0.114B' = Y$ $chroma1 = -0.299R' - 0.587G' + 0.886B' = B' - Y$ $chroma2 = 0.701R' - 0.587G' - 0.114B' = R' - Y$ Where R', G', B' are gamma-corrected R, G and B values. Providing 8-bit range of YCC achieved as follows: $luma = (255/1.402)luma$ $chroma1 = (111.40chroma1\_ + 156$ $chroma2 = (135.64chroma2) + 137$
YCC to RGB	YCC Color model transformed RGB Color model as follows: $L = 1.3584(luma)$ $C1 = 2.2178(chroma1 - 156)$ $C2 = 1.8215(chromf\,2 - 137)$ $R = L + C2$ $G = L - 0.194C1 - 0.509C2$ $B = L + C1$
RGB to YUV	RGB Color model transformed to YUV Color model as follows: $Y = 0.299R' + 0.587G' + 0.114B'$ $U = -0.147R' - 0.298G' + 0.436B' = 0.492(B' - Y)$ $V = 0.616R' - 0.515G' - 0.100B' = 0.877(R' - Y)$ Where R', G', B' are gamma-corrected R, G and B values
YUV to RGB	YUV Color model transformed to RGB Color model as follows: $R = Y + 1.140V$ $G = Y - 0.394U - 0.581V$ $B = Y + 2.032U$

Table 16-9 Color Model Conversions (continued)

RGB to YCbCr	RGB Color model transformed to YCbCr Color model as follows: $Y = 0.257R + 504G + 0.098B + 16$ $Cb = -0.148R - 0.291G + 0.439G + 128$ $Cr = 0.439R - 0.368G - 0.071B + 128$
YcbCr to RGB	YCbCr Color model transformed to RGB Color model as follows: $R = 1.164(Y - 16) + 1.596(Cr - 128)$ $G = 1.164(Y - 16) - 0.813(Cr - 128)$ $B = 1.164(Y - 16) + 2.017(Cb - 128)$

Table 16-9 Color Model Conversions (continued)

RGB to HSV	RGB Color model transformed to HSV Color model as follows:

$$M = \max(G, G, B)$$
$$m = \min(R, G, B)$$
$$r = (M - R)/(M - m)$$
$$g = (M - G)/(M - m)$$
$$b = (M - G)/(M - m)$$

Value-component:

$$V = \max(R, G, B)$$

Saturation-component:

$$S = \begin{cases} (M - m)/M, if \ \ M \neq 0 \\ 0, if \ \ M = 0 \end{cases}$$

Hue-component:

$$H = \begin{cases} 180, if \ \ M = 0 \\ 60(b - g), if \ \ R = M \\ 60(2 + r - b), if \ \ G = M \\ 60(4 + g - r), if \ \ B = M \end{cases}$$

Hue-component should be of 0 to 360 range

$$H = \begin{cases} H - 360, if \ \ H \geq 360 \\ H + 360, if \ \ H < 0 \end{cases}$$

Table 16-9 Color Model Conversions (continued)

HSV to RGB

HSV Color model transformed to RGB Color model as follows:

non chromatic case (S=0)

$$R = V$$
$$G = V$$
$$B = V$$

chromatic case (S!=0)

$$h = H / 60$$
$$i = \lceil h \rceil$$
$$f = h - i$$
$$p = V(1 - S)$$
$$q = V(1 - (S \cdot f))$$
$$t = V(1 - S(1 - f))$$

$$(R, G, B) = \begin{cases} (V, t, p), if \ \ i = 0 \\ (q, V, p), if \ \ i = 1 \\ (p, V, t), if \ \ i = 2 \\ (p, q, V), if \ \ i = 3 \\ (t, p, V), if \ \ i = 4 \\ (V, p, q), if \ \ i = 5 \end{cases}$$

Table 16-9 Color Model Conversions (continued)

RGB to HLS	RGB Color model transformed to HSV Color model as follows: $M = \max(G, G, B)$ $m = \min(R, G, B)$ $r = (M - R)/(M - m)$ $g = (M - G)/(M - m)$ $b = (M - G)/(M - m)$ Lightness-component: $L = (M + m)/2$ Saturation-component: $S = \begin{cases} 0, M = m \\ (M - m)/(M + m), L \le 0.5 \\ (M - m)/(2 - M - m), L > 0.5 \end{cases}$ Hue-component: $H = \begin{cases} 60(b - g), if \ \ R = M \\ 60(2 + r - b), if \ \ G = M \\ 60(4 + g - r), if \ \ B = M \\ 180, if \ \ M = m \end{cases}$ Hue-component should be of 0 to 360 range $H = \begin{cases} H - 360, if \ \ H \ge 360 \\ H + 360, if \ \ H < 0 \end{cases}$

Table 16-9 Color Model Conversions (continued)

HLS to RGB	HSV Color model transformed to RGB Color model as follows: non chromatic case (S=0) $R = L$ $G = L$ $B = L$ chromatic case (S!=0) $h = H / 60$ $i = \lceil h \rceil$ $f = h - i$ $M = \begin{cases} L(1+S), if \;\; L \le 0.5 \\ L + S - LS, if \;\; L > 0.5 \end{cases}$ $N = 2L - M$ $t = N + (M - N)f$ $q = M - (M - N)f$ $(R,G,B) = \begin{cases} (M,t,N), if \;\; i = 0 \\ (q,M,N), if \;\; i = 1 \\ (N,M,t), if \;\; i = 2 \\ (N,q,M), if \;\; i = 3 \\ (t,N,M), if \;\; i = 4 \\ (M,N,q), if \;\; i = 5 \end{cases}$

Table 16-9 Color Model Conversions (continued)

Gamma Correction	Forward: $R' = \begin{cases} 4.5R, if\ \ R < 0.018 \\ 1.099R^{0.45} - 0.99, if\ \ R \geq 0.018 \end{cases}$ $G' = \begin{cases} 4.5G, if\ \ G < 0.018 \\ 1.099G^{0.45} - 0.99, if\ \ G \geq 0.018 \end{cases}$ $B' = \begin{cases} 4.5B, if\ \ B < 0.018 \\ 1.099B^{0.45} - 0.99, if\ \ B \geq 0.018 \end{cases}$ Inverse: $R = \begin{cases} R'/0.45, if\ \ R' < 0.0812 \\ ((R' + 0.999)/1.099)^{2.2}, if\ \ R' \geq 0.0812 \end{cases}$ $G = \begin{cases} G'/0.45, if\ \ G' < 0.0812 \\ ((G' + 0.999)/1.099)^{2.2}, if\ \ G' \geq 0.0812 \end{cases}$ $B = \begin{cases} B'/0.45, if\ \ B' < 0.0812 \\ ((B' + 0.999)/1.099)^{2.2}, if\ \ B' \geq 0.0812 \end{cases}$

Table 16-10 Color Conversion Subsampling Conventions

Image Type	Downsampling	Description
YUV	None	Y, U, V sampled on every pixel. 8 bits per component = 24 bits per pixel.
4:2:2 YCbCr	2:1 horizontal	Y sampled on every pixel; U (Cb) and V (Cr) sampled every 2 pixels horizontally. 8 bits per component = 32 bits per pixel pair.
4:1:1 YCbCr	4:1 horizontal	Y sampled on every pixel; U (Cb) and V (Cr) sampled every 4 pixels horizontally. 8 bits per component = 48 bits for four pixels.

Table 16-10 Color Conversion Subsampling Conventions (continued)

Image Type	Downsampling	Description
4:2:0 YCbCr	2:1 horizontal, 2:1 vertical	Y sampled on every pixel, U (Cb) and V (Cr) sampled once on each 2x2 pixel block. 8 bits per component = 48 bits for four pixels.

RGBToXYZ_8u_C3R
XYZToRGB_8u_C3R

Prototype

```
IppStatus ippiRGBToXYZ_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiXYZToRGB_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from XYZ color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image pointed to by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

RGBToYCC_8u_C3R
YCCToRGB_8u_C3R

Prototype

```
IppStatus ippiRGBToYCC_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCCToRGB_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from YCC color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image pointed by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

RGBToLUV_8u_C3R
LUVToRGB_8u_C3R

Prototype

```
IppStatus ippiRGBToLUV_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiLUVToRGB_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from LUV color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image pointed to by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

RGBToHSV_8u_C3R
HSVToRGB_8u_C3R

Prototype

```
IppStatus ippiRGBToHSV_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiHSVToRGB_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from HSV color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image pointed to by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

RGBToHLS_8u_C3R
HLSToRGB_8u_C3R

Prototype

```
IppStatus ippiRGBToHLS_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiHLSToRGB_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from HLS color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image pointed to by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

RGBToYCbCr_8u_C3R
YCbCrToRGB_8u_C3R
YCbCrToRGB565_8u16u_C3R
YCbCrToRGB555_8u16u_C3R
YCbCrToRGB444_8u16u_C3R
YCbCrToBGR565_8u16u_C3R
YCbCrToBGR555_8u16u_C3R
YCbCrToBGR444_8u16u_C3R

Prototype

```
IppStatus ippiRGBToYCbCr_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCrToRGB_8u_C3R(const Ipp8u *pSrc, int srcStep, Ipp8u
   *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCrToRGB565_8u16u_C3R(const Ipp8u *pSrc, int srcStep,
   Ipp16u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCrToRGB555_8u16u_C3R(const Ipp8u *pSrc, int srcStep,
   Ipp16u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCrToRGB444_8u16u_C3R(const Ipp8u *pSrc, int srcStep,
   Ipp16u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCrToBGR565_8u16u_C3R(const Ipp8u *pSrc, int srcStep,
   Ipp16u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCrToBGR555_8u16u_C3R(const Ipp8u *pSrc, int srcStep,
   Ipp16u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCrToBGR444_8u16u_C3R(const Ipp8u *pSrc, int srcStep,
   Ipp16u *pDst, int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from YCbCr color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image pointed to by pDst.

- RGB565 will be 5 bits of red (bit 0-4), 6 bits of green (bit 5-10) and 5 bits of blue (bit 11-15) components which are packed into 16 bits data in the order of LSB to MSB.

- RGB555 will be 5 bits of red (bit 0-4), 5 bits of green (bit 5-9), 5 bits of blue (bit 10-14)components, and 1 bits zeros (bit 15)which are packed into 16 bits data in the order of LSB to MSB.
- RGB444 will be 4 bits of red (bit 0-3), 4 bits of green (bit 4-7), 4 bits of blue (bit 8-11) components, and 4 bits zeros (bit 12-15) which are packed into 16 bits data in the order of LSB to MSB.
- BGR565 will be 5 bits of blue (bit 0-4), 6 bits of green (bit 5-10) and 5 bits of red (bit 11-15) components which are packed into 16 bits data in the order of LSB to MSB.
- BGR555 will be 5 bits of blue (bit 0-4), 5 bits of green (bit 5-9), 5 bits of red (bit 10-14) components, and 1 bits zeros (bit 15) which are packed into 16 bits data in the order of LSB to MSB.
- BGR444 will be 4 bits of blue (bit 0-3), 4 bits of green (bit 4-7), 4 bits of red components (bit 8-11), and 4 bits zeros (bit 12-15) which are packed into 16 bits data in the order of LSB to MSB.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

RGBToYCbCr422_8u_C3C2R
YCbCr422ToRGB_8u_C2C3R
YCbCr422ToRGB565_8u16u_C2C3R
YCbCr422ToRGB555_8u16u_C2C3R

YCbCr422ToRGB444_8u16u_C2C3R
YCbCr422ToBGR565_8u16u_C2C3R
YCbCr422ToBGR555_8u16u_C2C3R
YCbCr422ToBGR444_8u16u_C2C3R

Prototype

```
IppStatus ippiRGBToYCbCr422_8u_C3C2R(const Ipp8u *pSrc, int srcStep,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCr422ToRGB_8u_C2C3R (const Ipp8u *pSrc, int srcStep,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCr422ToRGB565_8u_C2C3R (const Ipp8u *pSrc, int srcStep,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCr422ToRGB555_8u_C2C3R (const Ipp8u *pSrc, int srcStep,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCr422ToRGB444_8u_C2C3R (const Ipp8u *pSrc, int srcStep,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCr422ToBGR565_8u_C2C3R (const Ipp8u *pSrc, int srcStep,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCr422ToBGR555_8u_C2C3R (const Ipp8u *pSrc, int srcStep,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYCbCr422ToBGR444_8u_C2C3R (const Ipp8u *pSrc, int srcStep,
   Ipp8u *pDst, int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from YCbCr422 color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image indicated by pDst. YCbCr data are in two channel format as YCbYCrYCbYCr.....

- RGB565 will be 5 bits of red (bit 0-4), 6 bits of green (bit 5-10) and 5 bits of blue (bit 11-15) components which are packed into 16 bits data in the order of LSB to MSB.
- RGB555 will be 5 bits of red (bit 0-4), 5 bits of green (bit 5-9), 5 bits of blue (bit 10-14)components, and 1 bits zeros (bit 15)which are packed into 16 bits data in the order of LSB to MSB.
- RGB444 will be 4 bits of red (bit 0-3), 4 bits of green (bit 4-7), 4 bits of blue (bit 8-11) components, and 4 bits zeros (bit 12-15) which are packed into 16 bits data in the order of LSB to MSB.

- BGR565 will be 5 bits of blue (bit 0-4), 6 bits of green (bit 5-10) and 5 bits of red (bit 11-15) components which are packed into 16 bits data in the order of LSB to MSB.
- BGR555 will be 5 bits of blue (bit 0-4), 5 bits of green (bit 5-9), 5 bits of red (bit 10-14) components, and 1 bits zeros (bit 15) which are packed into 16 bits data in the order of LSB to MSB.
- BGR444 will be 4 bits of blue (bit 0-3), 4 bits of green (bit 4-7), 4 bits of red components (bit 8-11), and 4 bits zeros (bit 12-15) which are packed into 16 bits data in the order of LSB to MSB.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

RGBToYUV_8u_C3R
YUVToRGB_8u_C3R
RGBToYUV_8u_C3P3R
YUVToRGB_8u_P3C3R

Prototype

```
IppStatus ippiRGBToYUV_8u_C3R(const Ipp8u* pSrc, int srcStep , Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
```

```
IppStatus ippiYUVToRGB_8u_C3R(const Ipp8u* pSrc, int srcStep , Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
IppStatus ippiRGBToYUV_8u_C3P3R(const Ipp8u* pSrc, int srcStep , Ipp8u*
   pDst[3], int dstStep, IppiSize roiSize);
IppStatus ippiYUVToRGB_8u_P3C3R(const Ipp8u* const pSrc[3], int srcStep,
   Ipp8u* pDst,int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from YUV color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image indicated by pDst.

Input Arguments

- pSrc – pointer to the source ROI
- srcStep – step in bytes through the source image
- dstStep – step in bytes through the destination image
- roiSize – size of the source and destination ROI in pixels

Output Arguments

pDst – pointer to the destination ROI

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointer error
- ippStsStepErr – step value is less or equal zero
- ippStsSizeErr – illegal data size values

RGBToYUV422_8u_C3R
RGBToYUV422_8u_C3P3R
RGBToYUV422_8u_C3P3

YUV422ToRGB_8u_C3R
YUV422ToRGB_8u_P3C3R
YUV422ToRGB_8u_P3C3

Prototype

```
IppStatus ippiRGBToYUV422_8u_C3R(const Ipp8u* pSrc, int srcStep , Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
IppStatus ippiRGBToYUV422_8u_C3P3R(const Ipp8u* pSrc, int srcStep,
   Ipp8u* Dst[3],intdstStep[3], IppiSize roiSize);
IppStatus ippiRGBToYUV422_8u_C3P3(const Ipp8u* pSrc, Ipp8u* pDst[3],
   IppiSize imgSize);
IppStatus ippiYUV422ToRGB_8u_C3R(const Ipp8u* pSrc,int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
IppStatus ippiYUV422ToRGB_8u_P3C3R(const Ipp8u* const pSrc[3],int
   srcStep[3], Ipp8u* pDst,int dstStep, IppiSize roiSize);
IppStatus ippiYUV422ToRGB_8u_P3C3(const Ipp8u* const pSrc[3], Ipp8u*
   pDst, IppiSize imgSize);
```

Description

Convert RGB color space to and from YUV422 color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image indicated by pDst. The data YUV422 in C3 format are formatted as YUYVYUYV.....

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `imgSize` – size of the source and destination image in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- ippStsNoErr – no error

- ippStsNullPtrErr – NULL pointer error
- ippStsStepErr – step value is less or equal zero
- ippStsSizeErr – illegal data size values

RGBToYUV420_8u_C3P3R
RGBToYUV420_8u_C3P3
YUV420ToRGB_8u_P3C3R
YUV420ToRGB_8u_P3C3
YUV420ToRGB565_8u16u_P3C3R
YUV420ToRGB555_8u16u_P3C3R
YUV420ToRGB444_8u16u_P3C3R
YUV420ToBGR565_8u16u_P3C3R
YUV420ToBGR555_8u16u_P3C3R
YUV420ToBGR444_8u16u_P3C3R

```
IppStatus ippiRGBToYUV420_8u_C3P3R(const Ipp8u* pSrc, int srcStep,
   Ipp8u* pDst[3],int dstStep[3], IppiSize roiSize);
IppStatus ippiRGBToYUV420_8u_C3P3(const Ipp8u* pSrc, Ipp8u* pDst[3],
   IppiSize imgSize);
IppStatus ippiYUV420ToRGB_8u_P3C3R(const Ipp8u* const pSrc[3],int
   srcStep[3], Ipp8u* pDst,int dstStep, IppiSize roiSize);
IppStatus ippiYUV420ToRGB_8u_P3C3(const Ipp8u* const pSrc[3], Ipp8u*
   pDst, IppiSize imgSize);
IppStatus ippiYUV420ToRGB565_8u16u_P3C3R(const Ipp8u* const pSrc[3],int
   srcStep[3], Ipp16u* pDst,int dstStep, IppiSize roiSize);
IppStatus ippiYUV420ToRGB555_8u16u_P3C3R(const Ipp8u* const pSrc[3],int
   srcStep[3], Ipp16u* pDst,int dstStep, IppiSize roiSize);
IppStatus ippiYUV420ToRGB444_8u16u_P3C3R(const Ipp8u* const pSrc[3],int
   srcStep[3], Ipp16u* pDst,int dstStep, IppiSize roiSize);
IppStatus ippiYUV420ToBGR565_8u16u_P3C3R(const Ipp8u* const pSrc[3],int
   srcStep[3], Ipp16u* pDst,int dstStep, IppiSize roiSize);
IppStatus ippiYUV420ToBGR555_8u16u_P3C3R(const Ipp8u* const pSrc[3],int
   srcStep[3], Ipp16u* pDst,int dstStep, IppiSize roiSize);
```

```
IppStatus ippiYUV420ToBGR444_8u16u_P3C3R(const Ipp8u* const pSrc[3],int
   srcStep[3], Ipp16u* pDst,int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from YUV420 color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image indicated by pDst.

- RGB565 will be 5 bits of red (bit 0-4), 6 bits of green (bit 5-10) and 5 bits of blue (bit 11-15) components which are packed into 16 bits data in the order of LSB to MSB.
- RGB555 will be 5 bits of red (bit 0-4), 5 bits of green (bit 5-9), 5 bits of blue (bit 10-14)components, and 1 bits zeros (bit 15)which are packed into 16 bits data in the order of LSB to MSB.
- RGB444 will be 4 bits of red (bit 0-3), 4 bits of green (bit 4-7), 4 bits of blue (bit 8-11) components, and 4 bits zeros (bit 12-15) which are packed into 16 bits data in the order of LSB to MSB.
- BGR565 will be 5 bits of blue (bit 0-4), 6 bits of green (bit 5-10) and 5 bits of red (bit 11-15) components which are packed into 16 bits data in the order of LSB to MSB.
- BGR555 will be 5 bits of blue (bit 0-4), 5 bits of green (bit 5-9), 5 bits of red (bit 10-14) components, and 1 bits zeros (bit 15) which are packed into 16 bits data in the order of LSB to MSB.
- BGR444 will be 4 bits of blue (bit 0-3), 4 bits of green (bit 4-7), 4 bits of red components (bit 8-11), and 4 bits zeros (bit 12-15) which are packed into 16 bits data in the order of LSB to MSB.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `imgSize` – size of the source and destination image in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero

- ippStsSizeErr – illegal data size values

ColorTwistQ14_8u_C3R

Prototype

```
IppStatus ippiColorTwistQ14_8u_C3R(const Ipp8u* pSrc, int srcStep,
   Ipp8u* pDst, int dstStep, IppiSize roiSize, const Ipp32s
   twistQ14[3][4]);
```

Description

Apply color twist Q1.14 matrix twist_Q14 to the ROI of the source image pointed to by pSrc. Place the results into the ROI of the destination image pointed to by pDst. Ordinarily, the matrix for the color twist conversion has float–point format but the fix point matrix implementation is used here. Q14 modifier implies that the matrix values were obtained by multiplying every original float–point matrix element by (1<<14).

Input Arguments

- pSrc – pointer to the source ROI
- srcStep – step in bytes through the source image
- dstStep – step in bytes through the destination image
- roiSize – size of the source and destination ROI in pixels
- twistQ14 – twist matrix

Output Arguments

pDst – pointer to the destination ROI

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointer error
- ippStsStepErr – step value is less or equal zero
- ippStsSizeErr – illegal data size values

GammaFwd_8u_C3R
GammaInv_8u_C3R

Prototype

```
IppStatus ippiGammaFwd_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
IppStatus ippiGammaInv_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
```

Description

Convert RGB color space to and from R'G'B' color space image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image indicated by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Example 16-8 Use of Color Space Function

```
#include <stdlib.h>
#include "extools.h" /* The tool for print/display and listed in the appendix*/
#include "ipp.h"

int MAIN()
{
   // Three-channel images
   const int NC = 3;
   const int SIDE = 8;
   Ipp8u src[SIDE * NC*SIDE]; // source RGB image
   Ipp8u dst[SIDE * NC*SIDE]; // destination image should be XYZ

   // Fill source image by any values
   for(int n=0; n<NC*SIDE*SIDE; n++) src[n] = (Ipp8u)(rand()&0x3F);

   // Specify ROI location and size
   IppiPoint srcLocation = {NC,1}; // ROI Location at {1,1}
   IppiPoint dstLocation = {NC,1}; // but three-channel images in use
   IppiSize roiSize = {6,6};
   // Convert ROI
   ippiRGBToXYZ_8u_C3R(AddressOf(src,NC*SIDE,srcLocation), NC*SIDE,
                       AddressOf(dst,NC*SIDE,dstLocation), NC*SIDE, roiSize);

   // Print result
   IppiSize showSize = {NC*SIDE,SIDE};
   PrintROI_C1(_T("Example: Color Space"), dst, NC*SIDE, showSize);
```

Example 16-8 Use of Color Space Function (continued)

```
    return 0;
}
//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
    return pImg +sLine*aLocation.y +aLocation.x;
}
```

RGBToGray_8u_C3C1R
ColorToGray_8u_C3C1R

Prototype

```
IppStatus ippiRGBToGray_8u_C3C1R(const Ipp8u* pSrc,int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
IppStatus ippiColorToGrayQ14_8u_C3C1R(const Ipp8u* pSrc,int srcStep,
   Ipp8u* pDst, int dstStep, IppiSize roiSize, const Ipp32s coefQ14[3]);
```

Description

Convert Color space to Gray image. The ROI of the source image is pointed to by pSrc, the result is placed into the ROI of the destination image indicated by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `coefQ14` – Color to Gray coefficients

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Image Morphological Primitives

This section describes IPP image processing Primitives that performs basic morphological operations on an image. IPP image processing supports basic morphological operations like erosion and dilation. Both are operates with solid 3x3 structuring element.

Table 16-11 Morphological Definitions

Operation	Description
erode	Operation erode defines as the follows: $y_{n,m} = \min\{x_{i,j}\}$, where: $x_{i,j}$ and $y_{n,m}$ are input and output signals correspondingly $i = m-1, m, m+1$ and $j = n-1, n, n+1$
dilate	Operation dilate defines as the follows: $y_{n,m} = \max\{x_{i,j}\}$, where: $x_{i,j}$ and $y_{n,m}$ are input and output signals correspondents $i = m-1, m, m+1$ and $j = n-1, n, n+1$

Erode3x3_8u_C1R
Erode3x3_8u_C3R

Prototype

```
IppStatus ippiErode3x3_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
IppStatus ippiErode3x3_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
```

Description

Perform erosion of the ROI of the source image pointed to by pSrc. Place the result into the ROI of the destination image pointed to by pDst.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

pDst – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Dilate3x3_8u_C1R
Dilate3x3_8u_C3R

Prototype

```
IppStatus ippiDilate3x3_8u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
IppStatus ippiDilate3x3_8u_C3R(const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize);
```

Description

Dilates the image ROI.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Example 16-9 Use of Morphological Primitives

```
#include <stdlib.h>
#include "extools.h" /* The tool for print/display and listed in the appendix*/
#include "ipp.h"

int MAIN()
{
   // One-channel source and destination images
   const int SIDE = 16;
   Ipp8u src[SIDE*SIDE];
   Ipp8u tmp[SIDE*SIDE];
   Ipp8u dst[SIDE*SIDE];
   IppiSize imgSize = {SIDE,SIDE};

   // Fill source image by any values
   for(int i=0; i<SIDE*SIDE; i++) src[i] = (Ipp8u)(rand()&0xF);

   // Specify ROI location and size
   IppiPoint srcLocation = {1,1};
   IppiPoint dstLocation = {1,1};
   IppiSize roiSize = {7,7};

   //
   // Perform "opening" operation. Open = dilate(erode(img))
   //
   ippiCopy_8u_C1R(src,SIDE, tmp,SIDE, imgSize);

   // First step: perform Eerode operation
   ippiErode3x3_8u_C1R(AddressOf(src,SIDE,srcLocation),SIDE,
                       AddressOf(tmp,SIDE,srcLocation),SIDE, roiSize);
```

Example 16-9 Use of Morphological Primitives (continued)

```
   // Next step: perform Dilate operation
   ippiDilate3x3_8u_C1R(AddressOf(tmp,SIDE,srcLocation),SIDE,
                        AddressOf(dst,SIDE,dstLocation),SIDE, roiSize);

   // Print result
   IppiSize showSize = {9,9};
   PrintROI_C1(_T("Example: Morphological Primitives (source)"), src, SIDE,
showSize);
   PrintROI_C1(_T("Example: ippiErode (temporary)"), tmp, SIDE, showSize);
   PrintROI_C1(_T("Example: ippiDilate (opening)"),  dst, SIDE, showSize);

   return 0;
}
//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
   return pImg +sLine*aLocation.y +aLocation.x;
}
```

Image Thresholding Primitives

This section describes IPP image processing primitives that perform image thresholding operations.

Table 16-12 Thresholding Definitions

Operation	Description
Threshold	Performs thresholding of input values using specified comparison operation. Output values are set as follows: for comparison operation "greater than" , $$y_{n,m} = \begin{cases} T, \text{if } x_{n,m} > T \\ x_{n,m}, \text{if } x_{n,m} \le T \end{cases}$$ for comparison for operation "less than", $$y_{n,m} = \begin{cases} T, \text{if} x_{n,m} < T \\ x_{n,m}, \text{if} x_{n,m} \ge T \end{cases}$$ where $x_{n,m}$, $y_{n,m}$ are input and output values respectively, T is threshold value.
ThresholdGT	Performs thresholding of input values using "greater than" comparison operation. Output values are set as follows: $$y_{n,m} = \begin{cases} T, \text{if } x_{n,m} > T \\ x_{n,m}, \text{if } x_{n,m} \le T \end{cases}$$ where $x_{n,m}$, $y_{n,m}$ are input and output values respectively, T is threshold value.
ThresholdLT	Performs thresholding of input values using "less than" comparison operation. Output values are set as follows: $$y_{n,m} = \begin{cases} T, \text{if } x_{n,m} < T \\ x_{n,m}, \text{if } x_{n,m} \ge T \end{cases}$$ where $x_{n,m}$, $y_{n,m}$ are input and output values respectively, T is threshold value.

Table 16-12 Thresholding Definitions (continued)

Threshold_Value	Performs thresholding of input values using specified comparison operation. Output values are set as follows $y_{n,m} = \begin{cases} v, \text{if } x_{n,m} < T \\ x_{n,m}, \text{if } x_{n,m} \geq T \end{cases}$ for comparison operation “greater than” , for comparison operation “less than”, where $x_{n,m}$, $y_{n,m}$ are input and output values respectively, T is thresholding value, v is set value.
Threshold_GTVvalue	Performs thresholding of input values using “greater than” comparison operation. Output values are set as follows: $y_{n,m} = \begin{cases} v, \text{if } x_{n,m} > T \\ x_{n,m}, \text{if } x_{n,m} \leq \end{cases}$ where $x_{n,m}$, $y_{n,m}$ are input and output values respectively, T is threshold value, v is set value.
Threshold_LTVvValue	Performs thresholding of input values using “less than” comparison operation. Output values are set as follows: $y_{n,m} = \begin{cases} v, \text{if } x_{n,m} < T \\ x_{n,m}, \text{if } x_{n,m} \geq T \end{cases}$ where $x_{n,m}$, $y_{n,m}$ are input and output values respectively, T is thresholding value, v is set value.

Table 16-12 Thresholding Definitions (continued)

Threshold_GTLValue	Performs two–level thresholding of input values using “greater than” and “less than” comparison operations. Output values are set as follows: $$y_{n,m} = \begin{cases} v_{GT}, \text{if } x_{n,m} > T_{GT} \\ x_{n,m}, \text{if } T_{LT} \le x_{n,m} \le T_{GT} \\ v_{LT}, \text{if } x_{n,m} < T_{LT} \end{cases}$$ where $x_{n,m}$, $y_{n,m}$ are input and output values respectively, T_{GT} is threshold value for “greater than” comparison operation, T_{LT} is threshold value for “less than” comparison operation, v_{GT} is set value for “greater than” comparison operation, v_{LT} is set value for “less than” comparison operation.

Threshold_GT_8u_C1R
Threshold_GT_8u_C3R

Prototype

```
IppStatus ippiThreshold_GT_8u_C1R (const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, Ipp8u threshold);
IppStatus ippiThreshold_GT_8u_C3R (const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, const Ipp8u threshold[3]);
```

Description

Performs thresholding of pixel values in the ROI of source image using “greater than” comparison operation and place the result into the ROI of destination image `pDst`.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image

- roiSize – size of the source and destination ROI in pixels
- threshold – threshold value (or vector of values for multi–channel image)

Output Arguments

pDst – pointer to the destination ROI

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – pSrc, pDst or threshold[3] pointer is NULL
- ippStsStepErr – step value is less or equal zero
- ippStsSizeErr – illegal data size values

Threshold_LT_8u_C1R
Threshold_LT_8u_C3R

Prototype

```
IppStatus ippiThreshold_LT_8u_C1R (const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, Ipp8u threshold);
IppStatus ippiThreshold_LT_8u_C3R (const Ipp8u* pSrc, int srcStep, Ipp8u*
   pDst, int dstStep, IppiSize roiSize, const Ipp8u threshold[3]);
```

Description

Perform thresholding of pixel values in the ROI of source image pSrc using “less than” comparison operation and place the result into the ROI of destination image pDst.

Input Arguments

- pSrc – pointer to the source ROI
- srcStep – step in bytes through the source image
- dstStep – step in bytes through the destination image
- roiSize – size of the source and destination ROI in pixels
- threshold – threshold value (or vector of values for multi–channel image)

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pDst` or `threshold[3]` pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Threshold_GTVal_8u_C1R
Threshold_GTVal_8u_C3R

Prototype

```
IppStatus ippiThreshold_GTVal_8u_C1R (const Ipp8u* pSrc, int srcStep,
   Ipp8u* pDst, int dstStep, IppiSize roiSize, Ipp8u threshold, Ipp8u
   value);
IppStatus ippiThreshold_GTVal_8u_C3R (const Ipp8u* pSrc, int srcStep,
   Ipp8u* pDst, int dstStep, IppiSize roiSize, const Ipp8u threshold[3],
   const Ipp8u value[3]);
```

Description

Perform thresholding of pixel values in the ROI of source image `pSrc` using the “greater than” comparison operations and place the result into the ROI of destination image `pDst` setting pixel values to specified value (value[3]).

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `threshold` – threshold value (or vector of values for multi–channel image)
- `value` – Output value to be set for each pixel that satisfies the compare condition (or vector of values for multi–channel image)

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pDst`, `threshold[3]` or `value[3]` pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Threshold_LTVal_8u_C1R
Threshold_LTVal_8u_C3R

Prototype

```
IppStatus ippiThreshold_LTVal_8u_C1R (const Ipp8u* pSrc, int srcStep,
   Ipp8u* pDst, int dstStep, IppiSize roiSize, Ipp8u threshold, Ipp8u
   value);
IppStatus ippiThreshold_LTVal_8u_C3R (const Ipp8u* pSrc, int srcStep,
   Ipp8u* pDst, int dstStep, IppiSize roiSize, const Ipp8u threshold[3],
   const Ipp8u value[3]);
```

Description

Perform thresholding of pixel values in the ROI of source image `pSrc` using the “less than” comparison operation and places the result into the ROI of destination image `pDst` setting pixel values to specified value (value[3]).

Input Arguments

- `pSrc` –Pointer to the source ROI
- `srcStep` –Step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `threshold` – threshold value (or vector of values for multi–channel image)
- `value` – Output value to be set for each pixel that satisfies the compare condition (or vector of values for multi–channel image)

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pDst`, `threshold[3]` or `value[3]` pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Threshold_LTValGTVal_8u_C1R
Threshold_LTValGTVal_8u_C3R

Prototype

```
IppStatus ippiThreshold_LTValGTVal_8u_C1R (const Ipp8u* pSrc, int
   srcStep, Ipp8u* pDst, int dstStep, IppiSize roiSize, Ipp8u
   thresholdLT, Ipp8u valueLT, Ipp8u thresholdGT, Ipp8u valueGT);
IppStatus ippiThreshold_LTValGTVal_8u_C3R (const Ipp8u* pSrc, int
   srcStep, Ipp8u* pDst, int dstStep, IppiSize roiSize, const Ipp8u
   thresholdLT[3], const Ipp8u valueLT[3], const Ipp8u thresholdGT[3],
   const Ipp8u valueGT[3]);
```

Description

Perform thresholding of pixel values in the ROI of source image pSrc using the “greater than” and “less than” comparison operations and places the result into the ROI of destination image setting pixel values to specified values (value[3]).

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `dstStep` – step in bytes through the destination image
- `roiSize` – size of the source and destination ROI in pixels
- `thresholdGT` – the upper threshold value (or vector of values for multi–channel image)

- `valueGT` – Output value to be set for each pixel that is greater than `thresholdGT` (or vector of values for multi–channel image)
- `thresholdLT` – the lower threshold value (or vector of values for multi–channel image)
- `valueLT` –Output value to be set for each pixel that is less than thresholdLT (or vector of values for multi–channel image)

Output Arguments

`pDst` – pointer to the destination ROI

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pDst`, `thresholdGT[3]`, `thresholdLT[3]`, `valueGT[3]` or `valueLT[3]` pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `ippStsThresholdErr` – value of thresholdGT is less than value of thresholdLT

Example 16-10 Use of Thresholding Primitives

```
#include <stdlib.h>
#include "extools.h" /* The tool for print/display and listed in the appendix*/
#include "ipp.h"

int MAIN()
{
   // One-channel source and destination images
   const int SIDE = 16;
   Ipp8u src[SIDE*SIDE];
   Ipp8u dst[SIDE*SIDE];

   // Fill source image by any values
   int i;
   for(i=0; i<SIDE*SIDE; i++) src[i] = (Ipp8u)(rand()&0x3F);

   // Determine min & max values
   Ipp8u minVal=src[0], maxVal= src[0];
   for(i=1; i<SIDE*SIDE; i++) {
      minVal = IPP_MIN(minVal, src[i]);
      maxVal = IPP_MAX(minVal, src[i]);
   }

   // Specify ROI location and size
   IppiPoint srcLocation = {1,1};
   IppiPoint dstLocation = {1,1};
   IppiSize roiSize = {7,7};
   // Perform two-level thresholding.
   ippiThreshold_LTValGTVal_8u_C1R(AddressOf(src,SIDE,srcLocation), SIDE,
                                   AddressOf(dst,SIDE,dstLocation), SIDE, roiSize,
                                      (Ipp8u)(minVal+1),0,
(Ipp8u)(maxVal-1),IPP_MAX_8U);
```

Example 16-10 Use of Thresholding Primitives (continued)

```
   // Print result
   IppiSize showSize = {9,9};
   PrintROI_C1(_T("Example: ippiThreshold"),  dst, SIDE, showSize);

   return 0;
}
//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
   return pImg +sLine*aLocation.y +aLocation.x;
}
```

Image Statistical Primitives

This section describes IPP image processing Primitives that performs statistical-moments and norm operations on an image. The mathematical definition of these operation are given in Table 16-13 and Table 16-14.

Table 16-13 Statistical Moments Definitions

Operation	Description
spatial moment	Basic equation of spatial moment of (p,q) order is the following: $m_{pq} = \sum_{-\infty}^{\infty}\sum_{-\infty}^{\infty} x^p y^q f(x,y)$, where: $f(x,y)$ input image pixel at the (x,y) location
central moment	Basic equation of central moment of (p,q) order is the following: $\mu_{pq} = \sum_{-\infty}^{\infty}\sum_{-\infty}^{\infty} (x-\hat{x})^p (y-\hat{y})^q f(x,y)$ $\hat{x} = \frac{m_{10}}{m_{00}}$ and $\hat{y} = \frac{m_{01}}{m_{00}}$ are coordinates of the center of mass

Table 16-13 Statistical Moments Definitions (continued)

normalized spatial moment	Basic equation of normalized spatial moment of (p,q) order is the following $\gamma_{pq} = \frac{m_{pq}}{m_{00}^{[(p+q)/2]+1}}$
normalized central moment	Basic equation of normalized central moment of (p,q) order is the following: $\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{[(p+q)/2]+1}}$
Hu moments	The seven moment–based features, proposed by Hu, that are primitives of normalized moments up to the third order are: $\phi_1 = \eta_{20} + \eta_{02}$ $\phi_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$ $\phi_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$ $\phi_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$ $\phi_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})\ [(\eta_{03} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] +$ $(3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$ $\phi_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$ $\phi_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] -$ $(\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$

Table 16-14 Norm Values

Operation	Description
Infinity norm	Infinity norm of image x of size $W \times H$ is defined as follows: $\|x\|_\infty = \max\{|x_{n,m}|\}$

Table 16-14 Norm Values (continued)

1-norm	1–norm of image x of size $W \times H$ is defined as follows: $\|x\|_1 = \sum_{n=0}^{H-1}\sum_{m=0}^{W-1}\|x_{n,m}\|$
2-norm	2–norm of image x of size $W \times H$ is defined as follows: $\|x\|_2 = \sqrt{\sum_{n=0}^{H-1}\sum_{m=0}^{W-1}\|x_{n,m}\|^2}$
Absolute error (infinity norm)	Absolute error of image x and estimate image $\hat{x}$ of size $W \times H$ in case of infinity norm is defined as follows: $\varepsilon_{abs,\infty} = \|\hat{x} - x\|_\infty \equiv \max\{\|\hat{x}_{n,m} - x_{n,m}\|\}$
Absolute error (1-norm)	Absolute error of image x and estimate image $\hat{x}$ of size $W \times H$ in case of 1–norm is defined as follows: $\varepsilon_{abs,1} = \|\hat{x} - x\|_1 \equiv \sum_{n=0}^{H-1}\sum_{m=0}^{W-1}\|\hat{x}_{n,m} - x_{n,m}\|$
Absolute error (2-norm)	Absolute error of image x and estimate image $\hat{x}$ of size $W \times H$ in case of 2–norm is defined as follows: $\varepsilon_{abs,2} = \|\hat{x} - x\|_2 \equiv \sqrt{\sum_{n=0}^{H-1}\sum_{m=0}^{W-1}\|\hat{x}_{n,m} - x_{n,m}\|}$
Relative error (infinity norm)	Relative error of image x and estimate image $\hat{x}$ of size $W \times H$ in case of infinity norm is defined as follows: $\varepsilon_{rel,\infty} = \frac{\|\hat{x} - x\|_\infty}{\|x\|_\infty}$
Relative error (1-norm)	Relative error of image x and estimate image $\hat{x}$ of size $W \times H$ in case of 1–norm is defined as follows: $\varepsilon_{rel,1} = \frac{\|\hat{x} - x\|_1}{\|x\|_1}$
Relative error (2-norm)	Relative error of image x and estimate image $\hat{x}$ of size $W \times H$ in case of 2–norm is defined as follows: $\varepsilon_{rel,2} = \frac{\|\hat{x} - x\|_2}{\|x\|_2}$

MomentGetStateSize_64s

Prototype

```
IppStatus ippiMomentGetStateSize_64s(IppHintAlgorithm hint, int* pSize);
```

Description

Get size of state structure in bytes. This returns the structure size in pSize.

Input Arguments

`hint` – Recommends to use a specific code for the transform

Output Arguments

`pSize` – pointer to the size of structure

Returns

- `ippStsNoErr` – no error

MomentInit_64s

Prototype

```
IppStatus ippiMomentInit_64s(IppiMomentState_64s* pState,
   IppHintAlgorithm hint);
```

Description

Allocates memory and initialize state structure.

Input Arguments

`hint` – recommends to use a specific code for the transform

Output Arguments

`pState` – pointer to the initialized state structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error

MomentInitAlloc_64s

Prototype

```
IppStatus ippiMomentInitAlloc_64s(IppiMomentState_64s** pState,
   IppHintAlgorithm hint);
```

Description

Allocates memory and initialize state structure.

Input Arguments

`hint` – recommends to use a specific code for the transform

Output Arguments

`pState` – address of pointer to the created state structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsMemAllocErr` – not enough memory for the operation

MomentFree_64s

Prototype

```
IppStatus ippiMomentFree_64s(IppiMomentState_64s* pState);
```

Description

Release state structure created by ippiMomentInitAlloc_64s.

Input Arguments

`pState` – pointer to the state structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsContextMatchErr` – the context parameter doesn't match to the operation

Moments64s_8u_C1R
Moments64s_8u_C3R

Prototype

```
IppStatus ippiMoments64s_8u_C1R(const Ipp8u* pSrc, int srcStep, IppiSize
   roiSize, IppiMomentState_64s* pState);
IppStatus ippiMoments64s_8u_C3R(const Ipp8u* pSrc, int srcStep, IppiSize
   roiSize, IppiMomentState_64s* pState);
```

Description

Calculate statistical spatial moments of order 0 to 3 for the ROI of the image pointed to by pSrc.

Input Arguments

- `pSrc` – pointer to the source ROI
- `srcStep` – step in bytes through the source image
- `roiSize` – size of the ROI in pixels

Output Arguments

`pState` – pointer to the state structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `IppStsContextMatchErr` – the context parameter does not match to the operation

GetSpatialMoment_64s

Prototype

```
IppStatus ippiGetSpatialMoment_64s(const IppiMomentState_64s* pState,
   int mOrd, int nOrd, int nChannel, IppiPoint roiOffset, Ipp64s* pValue,
   int scaleFactor);
```

Description

Returns spatial by nOrd and mOrd spatial moment calculated by ippiMoments64s ( ) function. Place the scaled result into the memory pointed to by pValue.

Input Arguments

- `pState` – pointer to the state structure
- `nChannel` – specify the image channel number
- `roiOffset` – ROI location
- `pValue` – pointer to the variable required moment value to be stored. Function returns $m_{pq} \cdot 2^{-scaleFactor}$ value
- `scaleFactor` – value of the scale factor

Output Arguments

`mOrd,nOrd` – Specify the required spatial moment

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `IppStsContextMatchErr` – the context parameter doesn't match to the operation
- `ippStsSizeErr` – illegal data size values
- `ippStsChannelErr` – illegal channel number

GetCentralMoment_64s

Prototype

```
IppStatus ippiGetCentralMoment_64s(const IppiMomentState_64s* pState,
   int mOrd, int nOrd, int nChannel, Ipp64s* pValue, int scaleFactor);
```

Description

Returns specified by nOrd and mOrd central moment calculated by ippiMoments 64s ( ) function. Place the scaled result into the memory pointed to by pValue.

Input Arguments

- `pState` – pointer to the state structure
- `mOrd,nOrd` – specify the required spatial moment
- `nChannel` – specify the image channel number
- `scaleFactor` – value of the scale factor

Output Arguments

`pValue` – pointer to the variable required moment value to be stored. Function returns $\mu_{pq} \cdot 2^{-scaleFactor}$ value

Returns

- `ippStsNoErr` – no error

- ippStsNullPtrErr – NULL pointer error
- IppStsContextMatchErr – the context parameter doesn't match to the operation
- ippStsSizeErr – illegal data size values
- ippStsChannelErr – illegal channel number

GetNormalizedSpatialMoment_64s

Prototype

```
IppStatus ippiGetNormalizedSpatialMoment_64s(const IppiMomentState_64s*
   pState, int mOrd, int nOrd, int nChannel, IppiPoint roiOffset, Ipp64s*
   pValue, int scaleFactor);
```

Description

Returns specified by nOrd and mOrd normalized spatial moment calculated by ippiMoments64s ( ) function. Place the scaled result into the memory pointed to by pValue.

Input Arguments

- pState – pointer to the state structure
- mOrd,nOrd – specify the required spatial moment
- nChannel – specify the image channel number
- roiOffset – ROI location
- scaleFactor – value of the scale factor

Output Arguments

pValue – pointer to the variable required moment value to be stored. Function returns $\gamma_{pq} \cdot 2^{-scaleFactor}$ value

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointer error
- IppStsContextMatchErr – the context parameter doesn't match to the operation
- ippStsSizeErr – illegal data size values
- ippStsChannelErr – illegal channel number

- ippStsMoment00ZeroErr – moment value M(0,0) is too small to continue calculation

GetNormalizedCentralMoment_64s

Prototype

```
IppStatus ippiGetNormalizedCentralMoment_64s(const IppiMomentState_64s*
   pState, int mOrd, int nOrd, int nChannel, Ipp64s* pValue, int
   scaleFactor);
```

Description

Returns specified by nOrd and mOrd normalized central moment calculated by ippiMoments64s ( ) function. Place the scaled result into the memory pointed to by pValue.

Input Arguments

- pState – pointer to the state structure
- mOrd, nOrd – specify the required spatial moment
- nChannel – specify the image channel number
- scaleFactor – value of the scale factor

Output Arguments

pValue – pointer to the variable required moment value to be stored. Function returns value

$\eta_{pq} \cdot 2^{-scaleFactor}$

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointer error
- IppStsContextMatchErr – the context parameter doesn't match to the operation
- ippStsSizeErr – illegal data size values
- ippStsChannelErr – illegal channel number
- ippStsMoment00ZeroErr – moment value M(0,0) is too small to continue calculation

GetHuMoments_64s

Prototype

```
IppStatus ippiGetHuMoments_64s(const IppiMomentState_64s* pState, int
   nChannel, IppiHuMoment_64s pHm, int scaleFactor);
```

Description

Computes Hu invariant moments.

Input Arguments

- `pState` – pointer to the state structure
- `nChannel` – specify the image channel number
- `scaleFactor` – value of the scale factor

Output Arguments

`pHm` – pointer to the array of Hu–invariant moments buffer. Function calculates values:

$\phi_{1-7} \cdot 2^{-scaleFactor}$

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointer error
- `IppStsContextMatchErr` – the context parameter doesn't match to the operation
- `ippStsChannelErr` – illegal channel number
- `ippStsMoment00ZeroErr` – moment value M(0,0) is too small to continue calculation

Example 16-11 Use of Moment Primitives

```
#include <stdlib.h>
#include "extools.h" /* The tool for print/display and listed in the appendix*/
#include "ipp.h"

int MAIN()
{
   // Source one-channel images
   const int SIDE = 16;
   Ipp8u src[SIDE*SIDE];

   // Fill source image by any values
   for(int i=0; i<SIDE*SIDE; i++) src[i] = (Ipp8u)(rand()&0x1F);

   // Allocate and initialize State
   IppiMomentState_64s* pState;
   ippiMomentInitAlloc_64s(&pState, ippAlgHintNone);

   // Specify ROI location and size
   IppiPoint roiLocation = {1,1};
   IppiSize roiSize = {5,5};
   // Pre-calculates spatial moments of specified ROI and check status
   ippiMoments64s_8u_C1R(AddressOf(src,SIDE,roiLocation), SIDE, roiSize,
pState);

   // Get spatial moments M(0,0), M(1,0), M(0,1)
   Ipp64s m00, m10, m01;
   IppiPoint offset = {0,0};
   ippiGetSpatialMoment_64s(pState, 0,0,0, offset, &m00, 0);
   ippiGetSpatialMoment_64s(pState, 1,0,0, offset, &m10, 0);
   ippiGetSpatialMoment_64s(pState, 0,1,0, offset, &m01, 0);
```

Example 16-11 Use of Moment Primitives (continued)

```
   // Remove State
   ippiMomentFree_64s(pState);

   // Print ROI content
   PrintROI_C1(_T("Example: Moment Primitives"),
AddressOf(src,SIDE,roiLocation), SIDE, roiSize);
   // Print ROI centroid location
   _tprintf(_T("ROI centroid at: (%f:%f) location\n"),
(double)m10/m00,(double)m01/m00);

   return 0;
}
//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
   return pImg +sLine*aLocation.y +aLocation.x;}
```

Norm_Inf_8u_C1RSfs
Norm_Inf_8u_C3RSfs

Prototype

```
IppStatus ippiNorm_Inf_8u_C1RSfs (const Ipp8u* pSrc, int srcStep,
   IppiSize roiSize, Ipp32s* pNorm, int scaleFactor);
IppStatus ippiNorm_Inf_8u_C3RSfs (const Ipp8u* pSrc, int srcStep,
   IppiSize roiSize, Ipp32s norm[3], int scaleFactor);
```

Description

Compute the infinity norm of pixel values in the ROI.

Input Arguments

- `pSrc` – pointer to the ROI
- `srcStep` – step in bytes through the image
- `scaleFactor` – scale factor value

Output Arguments

- `roiSize` – size of the ROI in pixels
- `pNorm` – pointer to the variable where to store the result (one–channel image)
- `norm[3]` – address of vector where to store the result (three–channel image)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – pSrc, pNorm or norm[3] pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Norm_L1_8u_C1RSfs
Norm_L1_8u_C3RSfs

Prototype

```
IppStatus ippiNorm_L1_8u_C1RSfs (const Ipp8u* pSrc, int srcStep, IppiSize
   roiSize, Ipp32s* pNorm, int scaleFactor);
IppStatus ippiNorm_L1_8u_C3RSfs (const Ipp8u* pSrc, int srcStep, IppiSize
   roiSize, Ipp32s norm[3], int scaleFactor);
```

Description

Compute 1–norm of pixel values in the ROI `pSrc`.

Input Arguments

- `pSrc` – pointer to the ROI
- `srcStep` – step in bytes through the image
- `roiSize` – size of the ROI in pixels

- `scaleFactor` – scale factor value

Output Arguments

- `pNorm` – pointer to the variable where to store the result (one–channel image)
- `norm[3]` – address of vector where to store the result (three–channel image)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pNorm` or `norm[3]` pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Norm_L2_8u_C1RSfs
Norm_L2_8u_C3RSfs

Prototype

```
IppStatus ippiNormL2_8u_C1RSfs (const Ipp8u* pSrc, int srcStep, IppiSize
   roiSize, Ipp32s* pNorm, int scaleFactor);
IppStatus ippiNormL2_8u_C3RSfs (const Ipp8u* pSrc, int srcStep, IppiSize
   roiSize, Ipp32s norm[3], int scaleFactor);
```

Description

Compute 2–norm of pixel values in the ROI.

Input Arguments

- `pSrc` – pointer to the ROI
- `srcStep` – step in bytes through the image
- `roiSize` – size of the ROI in pixels
- `scaleFactor` – scale factor value

Output Arguments

- `pNorm` – pointer to the variable where to store the result (one–channel image)
- `norm[3]` – address of vector where to store the result (three–channel image)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc`, `pNorm` or `norm[3]` pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `ippStsSqrtNegArg` – sqrt() function became a negative argument through calculations

NormDiff_Inf_8u_C1RSfs
NormDiff_Inf_8u_C3RSfs

Prototype

```
IppStatus ippiNormDiff_Inf_8u_C1RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s* pNorm, int
   scaleFactor);
IppStatus ippiNormDiff_Inf_8u_C3RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s norm[3],
   int scaleFactor);
```

Description

Compute the absolute error (infinity norm case) of two ROIs pSrc1 and pSrc2.

Input Arguments

- `pSrc1` – pointer to the ROI in the first image
- `src1Step` – step in bytes through the first image
- `pSrc2` – pointer to the ROI in the second image
- `src2Step` – step in bytes through the second image
- `roiSize` – size of the ROIs in pixels
- `scaleFactor` – scale factor value

Output Arguments

- `pNorm` – pointer to the variable where to store the result (one–channel image)
- `norm[3]` – address of vector where to store the result (three–channel image)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc1`, `pSrc2`, `pNorm` or `norm[3]` pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

NormDiff_L1_8u_C1RSfs
NormDiff_L1_8u_C3RSfs

Prototype

```
IppStatus ippiNormDiff_L1_8u_C1RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s* pNorm, int
   scaleFactor);
IppStatus ippiNormDiff_L1_8u_C3RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s norm[3],
   int scaleFactor);
```

Description

Compute the absolute error (1–norm case) of two ROIs pSrc1 and pSrc2.

Input Arguments

- `pSrc1` – pointer to the ROI in the first image
- `src1Step` – step in bytes through the first image
- `pSrc2` – pointer to the ROI in the second image
- `src2Step` – step in bytes through the second image
- `roiSize` – size of the ROIs in pixels
- `scaleFactor` – scale factor value

Output Arguments

- `pNorm` – pointer to the variable where to store the result (one–channel image)
- `norm[3]` – address of vector where to store the result (three–channel image)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – pSrc1, pSrc2, pNorm or norm[3] pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

NormDiff_L2_8u_C1RSfs
NormDiff_L2_8u_C3RSfs

Prototype

```
IppStatus ippiNormDiff_L2_8u_C1RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s* pNorm, int
   scaleFactor);
IppStatus ippiNormDiff_L2_8u_C3RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s norm[3],
   int scaleFactor);
```

Description

Compute the absolute error (2–norm case) of two ROIs pSrc1 and pSrc2.

Input Arguments

- `pSrc1` – pointer to the ROI in the first image
- `src1Step` – step in bytes through the first image
- `pSrc2` – pointer to the ROI in the second image
- `src2Step` – step in bytes through the second image
- `roiSize` – size of the ROIs in pixels
- `scaleFactor` – scale factor value

Output Arguments

- `pNorm` – pointer to the variable where to store the result (one–channel image)
- `norm[3]` – address of vector where to store the result (three–channel image)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – pSrc1, pSrc2, pNorm or norm[3] pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values
- `ippStsSqrtNegArg` – negative input argument to sqrt() function detected

NormRel_Inf_8u_C1RSfs
NormRel_Inf_8u_C3RSfs

Prototype

```
IppStatus ippiNormRel_Inf_8u_C1RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s* pNorm, int
   scaleFactor);
IppStatus ippiNormRel_Inf_8u_C3RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s norm[3],
   int scaleFactor);
```

Description

Compute the relative error (infinity norm case) of two ROIs pSrc1 and pSrc2.

Input Arguments

- `pSrc1` – pointer to the ROI in the first image
- `src1Step` – step in bytes through the first image
- `pSrc2` – pointer to the ROI in the second image
- `src2Step` – step in bytes through the second image
- `roiSize` – size of the ROIs in pixels
- `scaleFactor` – scale factor value

Output Arguments

- `pNorm` – pointer to the variable where to store the result (one–channel image)
- `norm[3]` – address of vector where to store the result (three–channel image)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – pSrc1, pSrc2, pNorm or norm[3] pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

NormRel_L1_8u_C1RSfs
NormRel_L1_8u_C3RSfs
NormRel_L2_8u_C1RSfs
NormRel_L2_8u_C3RSfs

Prototype

```
IppStatus ippiNormRel_L1_8u_C1RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s* pNorm, int
   scaleFactor);
IppStatus ippiNormRel_L1_8u_C3RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s norm[3],
   int scaleFactor);
IppStatus ippiNormRel_L2_8u_C1RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s* pNorm, int
   scaleFactor);
IppStatus ippiNormRel_L2_8u_C3RSfs (const Ipp8u* pSrc1, int src1Step,
   const Ipp8u* pSrc2, int src2Step, IppiSize roiSize, Ipp32s norm[3],
   int scaleFactor);
```

Description

Compute the relative error:

- 1–norm case of two ROIs.
- 2-norm case of two ROIs.

Input Arguments

- `pSrc1` – pointer to the ROI in the first image
- `src1Step` – step in bytes through the first image

- `pSrc2` – pointer to the ROI in the second image
- `src2Step` – step in bytes through the second image
- `roiSize` – size of the ROIs in pixels
- `scaleFactor` – scale factor value

Output Arguments

- `pNorm` – pointer to the variable where to store the result (one–channel image)
- `norm[3]` – address of vector where to store the result (three–channel image)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – pSrc1, pSrc2, pNorm or norm[3] pointer is NULL
- `ippStsStepErr` – step value is less or equal zero
- `ippStsSizeErr` – illegal data size values

Example 16-12 Use of Norm Primitives

```
#include <stdlib.h>
#include "extools.h" /* The tool for print/display and listed in the appendix*/
#include "ipp.h"

int MAIN()
{
   // Source one-channel images
   const int SIDE = 16;
   Ipp8u src1[SIDE*SIDE];
   Ipp8u src2[SIDE*SIDE];

   // Fill images by values
   for(int i=0; i<SIDE*SIDE; i++) {
      src1[i] = (Ipp8u)(rand()&0x1F);
      src2[i] = (Ipp8u)(rand()&0x1F);
   }

   // Specify ROI location and size
   IppiPoint roiLocation = {1,1};
   IppiSize roiSize = {5,5};
   // Get max of absolute error
   Ipp32s NormDiffInf;
   ippiNormDiff_Inf_8u_C1RSfs(AddressOf(src1,SIDE,roiLocation), SIDE,
                              AddressOf(src2,SIDE,roiLocation), SIDE,
                              roiSize, &NormDiffInf, 0);
   // Get relative error
   Ipp32s NormRelInf;
   int scaleFactor = 16; // value (-16) means that calculated norm values are
                         // fixed-point with fractional part of 16 bits
   ippiNormRel_Inf_8u_C1RSfs(AddressOf(src1,SIDE,roiLocation), SIDE,
                             AddressOf(src2,SIDE,roiLocation), SIDE,
                             roiSize, &NormRelInf, -scaleFactor);
```

Example 16-12 Use of Norm Primitives (continued)

```
   // Print error values
   _tprintf(_T("Absolute error (infinity): %d\n"), NormDiffInf);
   _tprintf(_T("Relative error (infinity): %f\n"),
(double)NormRelInf/(1<<scaleFactor));
return 0;
}
//
// Returns address of ROI
//
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation)
{
   return pImg +sLine*aLocation.y +aLocation.x;
}
```

Camera Image Processing

This section describes the set of special-purpose image and video processing building blocks for camera applications that are provided as part of the Intel® Integrated Performance Primitives (Intel® IPP, function domain ippIP) for the Intel XScale® microarchitecture. The section fully defines the function calls, function behaviors, function arguments, function usage conventions, status return codes, and data structures that comprise the Intel® IPP camera interface Application Programming Interface (API).

Product Objectives

The design of the Intel® IPP camera API for the Intel XScale® microarchitecture satisfies the following three objectives:

Performance

In order to facilitate reduced power consumption, longer battery life, and maximum possible CPU headroom for concurrent applications, media, or system-level processes, the camera API has been optimized for the Intel XScale® microarchitecture such that an image or video processing chain constructed from its building blocks consumes a minimum number of MIPS with a small memory (ROM/RAM) footprint. Functions have been partitioned such that bus bandwidth is minimized, to the extent possible, for typical still and video camera preview and capture applications.

Quality

For a given power-performance operating point, the Intel® IPP camera API offers Intel XScale® microarchitecture-based products and devices the same high quality image or video output as is typically achieved by many camera solutions with dedicated image processing hardware.

OS Independence

The API, comprised of three core functions, implements all of the operating system-independent (OS-independent) computationally intensive portions of the image and video processing chain required to support image or video preview and capture applications with a variety of available high-level camera sensors over the Intel® PXA27x processor camera capture interface. At the same time, the API allows the user application to manage all of the OS-dependent and application-related high-level administrative functions such as memory allocation, buffer control, and media stream synchronization.

Target processors for this product incorporate the Intel XScale® microarchitecture. Moreover, the Intel® IPP camera API for the Intel XScale® microarchitecture has been designed to comply with the platform-independent Intel® Integrated Performance Primitives Interface Specification.

Model

Product Description

The constituent functions of the Intel® IPP camera API for the Intel XScale® microarchitecture can be combined to support a wide range of camera applications including the following:

- Image preview
- Image capture
- Video preview
- Video capture
- Full-duplex video conferencing

Figure 16-3 Still Image Capture over the Camera Interface with Intel® IPP

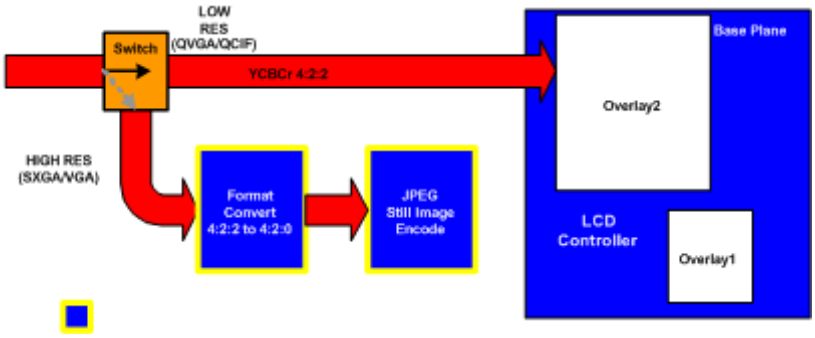


Figure 16-4 Video Capture over the Camera Interface with Intel® Integrated Performance Primitives

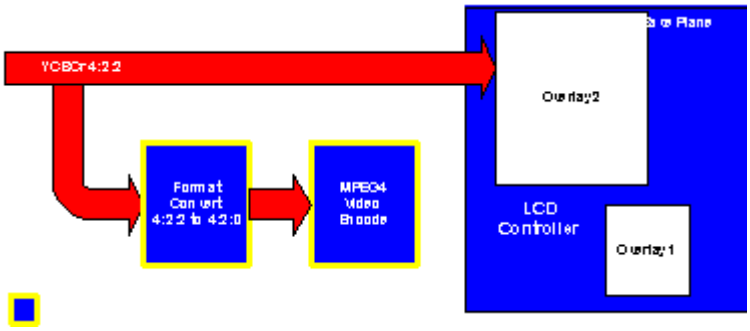
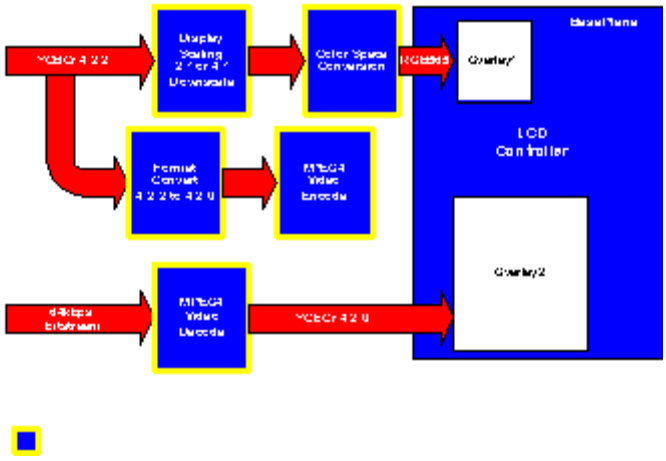


Figure 16-5 Video Conference over the Camera Interface with Intel® Integrated Performance Primitives



Product Capabilities

The primitives that comprise the Intel® IPP Camera API include the following:

Image Resizing with Color Space Conversion and Rotation

ResizeCscRotate_8u_C2R

Prototype

```
IppStatus ippiResizeCscRotate_8u_C2R(const Ipp8u *pSrc, int srcStep,
   Ipp16u *pDst, int dstStep, ippiSize roiSize, int scaleFactor,
   IppCameraInterpolation interpolation, IppCameraCsc colorConversion,
   IppCameraRotation rotation);
```

Description

This function synthesizes a low-resolution preview image for high-resolution video or still capture applications. In order to maximize bus bandwidth efficiency, this function combines several operations into a single function call, including scale reduction (2:1, 4:1 or 8:1), color space conversion, and rotation.

Color Space Conversion with Rotation

YCbCr422ToYCbCr420Rotate_8u_C2P3R

Prototype

```
IppStatus ippiYCbCr422ToYCbCr420Rotate_8u_C2P3R(const Ipp8u *pSrc, int
   srcStep, Ipp8u *pDst[3], int dstStep[3], ippiSize roiSize, int
   rotation);
```

Description

This function decimates and interpolates the color space of the input image from YCbCr 422 to YCbCr 420 planar data, applies an optional rotation of -90, +90, or 180 degrees, and then rearranges the data from the pixel-oriented input format to a planar output format.

YCbCr422ToYCbCr420Rotate_8u_P3R

Prototype

```
IppStatus ippiYCbCr422ToYCbCr420Rotate_8u_P3R(const Ipp8u *pSrc[3], int
   srcStep[3], Ipp8u *pDst[3], int dstStep[3], ippiSize roiSize, int
   rotation);
```

Description

This function decimates and interpolates the color space of the input image from YCbCr 422 planar data to YCbCr 420 planar data, and then applies an optional rotation of -90, +90, or 180 degrees.

Environment

Hardware and Operating Systems

The standard Intel® IPP development environment includes an x86 CPU workstation (Pentium III/4) configured with Microsoft Windows* 2000.

Software

The Intel® IPP code base has been developed in ANSI C under the Visual C++ 6.0 environment. Performance critical functions have been optimized for the Intel XScale® microarchitecture using assembly code. Performance simulations are performed on an Intel XScale® microarchitecture hardware platform.

Interfaces

Files

The Intel® Integrated Performance Primitives for the Intel XScale® microarchitecture comprise a set of static libraries against which an application may be linked. Function prototypes, as well as assorted useful macros and constants are provided in a collection of C header files. Function object code is contained in a set of library binaries.

Header Files

The Intel® IPP image processing interface header files <ippIP.h>, must be included at the beginning of any application source module that references the camera image and video processing primitives, as shown in the following example:

```
#include "ippIP.h"
int main()
{.
 .
 .
    /* call camera image and video IPP functions */
ippiResizeCscRotate_8u_C2R(. . .);
```

```
        .
        .
        .
}
```

Binary Libraries

The Intel® IPP camera binary library must be referenced by the linker when building an application containing source modules that reference any of the Intel® IPP camera primitives. For any particular supported operating system, two different library binaries are provided in the installation package. One is the debug version, which enables enhanced argument bounds checking for function entry points. The other is a release version, which maximizes performance by minimizing argument bounds checking and error reporting to a compact essential set required for a released product. For example, the names of release-mode image processing Intel® IPP binary libraries for the Windows* CE and Linux* operating systems, respectively, are "ippIP_WMMX41BPPC_r.lib" and "ippIP_WMMX41BLNX_r.lib".

Macros

A macro, in a language-independent context, is a defined symbolic name to be substituted with particular replacement text at compilation or assembly time. Macros might include numeric or string constants as well as functional units.

Common Macros

The Intel® IPPgeneral header file <ippdefs.h> defines the set of constants shown in Table 16-15.

Table 16-15 Constant Macros

Macros	Defined as	Description
TRUE	1	Logical true
FALSE	0	Logical false
NULL	((void *)0)	NULL pointer
IPP_MAX_16S	32767	16-bit signed integer maximum
IPP_MIN_16S	-32768	16-bit signed integer minimum
IPP_MAX_32S	2147483647	32-bit signed integer maximum
IPP_MIN_32S	-2147483648	32-bit signed integer maximum

Flags

Flags are pre-defined arguments or return values employed by the constituent functions that comprise the API. The file <ippdefs.h> defines the set of flags shown in Table 16-16.

Table 16-16 Flag Macros

Macros	Used in	Description
IppStsNoErr	Status codes for all functions	No error
IppStsBadArgErr		Bad argument(s)
IppStsErr		Error(s) has (have) occurred
IppStsNoMemErr		Out of memory

Data Types

General Data Types

Intel® IPP for the Intel XScale® microarchitecture supports only integer data types, which means that all function parameters and return values are of integer types. The most frequently used Intel® IPP integer data types are shown below in Table 16-17.

Table 16-17 Common Data Types Used

Data Type	Corresponding Data Type in C	Corresponding Data Type in ARM Assembly
8-bit unsigned integer	Ipp8u	Byte
16-bit integer	Ipp16s	Signed Halfword
32-bit integer	Ipp32s	Signed Word
64-bit integer	Ipp64s	Signed Double Word
32-bit IPP status code	typedef Ipp32s IppStatus;	Signed Word
Pointer	Any_type *	Unsigned Word

NOTE. *Denote by "Any_type" any of the other Intel® IPP data types specified in* Table 16-17.

Camera Enumerated Types

The Intel® IPP camera APIs define a collection of enumerated data types that facilitate efficient synchronization and data transfer between the various camera components. As shown in Table 16-18, the camera API includes several enumerated types that have been defined to provide semantic interpretations for frequently used constants.

Table 16-18 IPP Camera Enumerated Types

Enumerated Type Name	Symbolic Values	Constant Value
IppCameraInterpolation	ippCameraInterpNearest	0
	ippCameraInterpBilinear	1
IppCameraRotatation	ippCameraRotateDisable	0
	ippCameraRotate90L	1
	ippCameraRotate90R	2
	ippCameraRotate180	3
	ippCameraFlipHorizontal	4
	ippCameraFlipVertical	5
IppCameraCsc	ippCameraCscYCbCr422ToRGB565	0
	ippCameraCscYCbCr422ToRGB555	1
	ippCameraCscYCbCr422ToRGB444	2
	ippCameraCscYCbCrToRGB888	3

Camera Image Processing Functions

ResizeCscRotate

Prototype

```
IppStatus ippiResizeCscRotate_8u_C2R(const Ipp8u *pSrc, int srcStep,
   Ipp16u *pDst, int dstStep, ippiSize roiSize, int scaleFactor,
   IppCameraInterpolation interpolation, IppCameraCsc colorConversion,
   IppCameraRotation rotation);
```

Description

This function synthesizes a low-resolution preview image for high-resolution video or still capture applications. In order to maximize bus bandwidth efficiency, several atomic image processing kernels have been combined into a single function. In particular, the following sequence of operations is applied to the raw input image:

1. Scale reduction by an integer scalefactor. First, the input image scale is reduced by an integer scalefactor of either 2, 4, or 8 using the interpolation methodology specified by the control parameter interpolation. The following interpolation schemes are supported: nearest neighbor, bilinear. For each of these, respectively, computational complexity and preview image quality rank from low to high.
2. Color space conversion. Following scale reduction, color space conversion is applied according to the control parameter colorConversion.
3. Rotation. After color space conversion, the preview output image is rotated according to the control parameter rotation.

Input Arguments

- `pSrc` – pointer to the start of the buffer containing the pixel-oriented input image.
- `srcStep` – distance, in bytes, between the start of lines in the source image.
- `dstStep` – distance, in bytes, between the start of lines in the destination image.
- `roiSize` – dimensions, in pixels, of the source and destination regions of interest.
- `scaleFactor` – reduction scalefactor; values other than 2, 4, or 8 are invalid.
- `interpolation` – interpolation methodology control parameter; must take one of the following values: `ippCameraInterpNearest` or `ippCameraInterpBilinear` for nearest neighbor or bilinear interpolation, respectively.
- `colorConversion` – color conversion control parameter; must be set to one of the following pre-defined values: `ippCameraCscYCbCr422ToRGB565` or `ippCameraCscYCbCr422ToRGB555`
- `rotation` – rotation control parameter; must be set to one of the following pre-defined values: . `ippCameraRotateDisable`, `ippCameraRotate90L`, `ippCameraRotate90R`, or `ippCameraRotate180.`

Output Arguments

- `pDst` – pointer to the start of the buffer containing the resized, color-converted, and rotated output image.

Returns

If the function runs without error, it returns ippStsNoErr.

If one of the following cases occurs, the function returns ippStsBadArgErr:

- `pSrc` or `pDst` is NULL.
- `pSrc` or `pDst` is not aligned at 8 bytes boundary.
- `srcStep` or `dstStep` is less than 1, or `srcStep`, or `dstStep` is not multiple of 8.
- roiSize.width is larger than half of `srcStep`.
- roiSize.width or roiSize.height is less than `scaleFactor`.
- Invalid values of one or more of the following control parameters:
  - `scaleFactor`, `interpolation`, `colorConversion` or `rotation`.
  - Half of the `dstStep` is less than width of downscaled, color-converted and rotated image.

NOTE. *Input Byte order:*
*Input (*pSrc) byte ordering is Cb Y Cr Y, which is supported by Intel PXA27x processor. The memory organization for the packed format is illustrated in* Table 16-19.

Table 16-19 Memory Organization for 4:2:2 YCbCr Packed Format

Bit	31 24	25 16	15 8	7 0
Base+0x0	Y1	Cr0	Y0	Cb0
Base+0x4	Y3	Cr1	Y2	Cb1

Alignment Requirement

The start address of *pSrc*, *pDst*, must be aligned at 8-byte boundary; *srcStep* and *dstStep* must be multiple of 8.

Size of Output Image

If roiSize.width or roiSize.height cannot be divided by *scaleFactor* exactly, it will be cut to be the multiple of *scaleFactor*. For example, if the rotation control parameter is ippCameraRotateDisable or ippCameraRotate180, the output image's roiSize.width is equal to round ((input image's roiSize.width)/scaleFactor).

Accuracy Criteria

Compared with the floating-point implementation in double precision, the ABS error < 1 for R/G/B respectively.

Example

Here is an example for how to call the ippiResizeCscRotate_8u_C2R.

```
#include <stdlib.h>
#include "ippIP.h"
#define  _ALIGN8(adr)    ((((Ipp32u)(adr))+7)&(~7))

int main()
{
      int W, H, BufSize;

      /*   Variables in preview function   */
      Ipp8u        *pSrc;
      Ipp16u       *pDst;
      int          scaleFactor;
      int          srcStep, dstStep;
      IppCameraInterpolation interpolation;
      IppCameraRotation rotation;
      IppCameraCsc colorConversion;
      IppiSize roiSize;

      /*   Initialize Width and Height of the input image */
      W=320;
      H=240;

      /*   Initialize control parameters    */
      interpolation    = ippCameraInterpBilinear;
```

```
        rotation         = ippCameraRotate90R;
        colorConversion  = ippCameraCscYCbCr422ToRGB565;

        /*   Initialize source parameter   */
        roiSize.width   = W;
        roiSize.height   = H;
        srcStep     = 640;
        dstStep    = 240;
        scaleFactor      = 2;

        /*   Allocate buffer */
        pSrc = (Ipp8u*) malloc(srcStep*H+7);
        pDst = (Ipp16u*) malloc(dstStep*W+7);

        /*   Align Source and Output Buffer at 8-byte   */
        pDst  = (Ipp16u*)_ALIGN8(pDst);
        pSrc = (Ipp8u*)_ALIGN8(pSrc);

        /*   Call preview Function  */
ippiResizeCscRotate_8u_C2R(pSrc, srcStep, pDst, dstStep, roiSize,
   scaleFactor,
interpolation, colorConversion, rotation);

        /*   Free buffer   */
        free(pSrc);
        free(pDst);

        return (0);
}
```

Arbitrary ResizeRotate

YCbCr420RszRot_8u_P3R
YCbCr422RszRot_8u_P3R

Prototype

```
IppStatus ippiYCbCr420RszRot_8u_P3R( Ipp8u *pSrc[3], int srcStep[3],
   IppiSize srcSize, Ipp8u *pDst[3], int dstStep[3], IppiSize dstSize,
   IppCameraInterpolation interpolation, IppCameraRotation rotation, int
   rcpRatiox, int rcpRatioy);
IppStatus ippiYCbCr422RszRot_8u_P3R( Ipp8u *pSrc[3], int srcStep[3],
   IppiSize srcSize, Ipp8u *pDst[3], int dstStep[3], IppiSize dstSize,
   IppCameraInterpolation interpolation, IppCameraRotation rotation, int
   rcpRatiox, int rcpRatioy);
```

Description

This function combines several atomic image processing kernels into a single function in order to maximize bus bandwidth efficiency. In particular, the following sequence of operations is applied to the raw input image:

1. Spatial resizing. First, the input image of srcSize is scaled to dstSize image using the interpolation methodology specified by the control parameter interpolation. The following interpolation schemes are supported: nearest neighbor, bilinear. For each of these, respectively, computational complexity and preview image quality rank from low to high.
2. Rotation. After color space conversion, the preview output image is rotated according to the control parameter rotation.

The input data should be YCbCr420 planar format for <ippiYCbCr420RszRot_8u_P3R> and YCbCr422 planar format for <ippiYCbCr422RszRot_8u_P3R>.

Input Arguments

- `pSrc` – a 3-element vector containing pointers to the start of each of the YCbCr420 input planes for <ippiYCbCr420RszRot_8u_P3R> and YCbCr422 input planes for <ippiYCbCr422RszRot_8u_P3R>.
- `srcStep` – a 3-element vector containing the distance, in bytes, between the start of lines in each of the input image planes.

- `dstStep` – a 3-element vector containing the distance, in bytes, between the start of lines in each of the output image planes.
- `srcSize` – dimensions, in pixels, of the source image.
- `dstSize` – dimensions, in pixels, of the destination image(before applying rotation to the resized image).
- `interpolation` – interpolation methodology control parameter; must take one of the following values: ippCameraInterpNearest, or ippCameraInterpBilinear for nearest neighbor or bilinear interpolation, respectively.
- `rotation` – rotation control parameter; must be set to one of the following pre-defined values: . ippCameraRotateDisable, ippCameraRotate90L, ippCameraRotate90R, ippCameraRotate180, ippCameraFlipHorizontal, or ippCameraFlipVertical
- `rcpRatiox` – reciprocal resizing ratio in X direction, which is in Q16 format. If rcpRatiox>65536, it means the input image will expand in x direction.
- `rcpRatioy` – reciprocal resizing ratio in Y direction, which is in Q16 format. If rcpRatioy>65536, it means the input image will expand in y direction.

Output Arguments

- `pDst` – a 3-element vector containing pointers to the start of each of the YCbCr420 output planes.

Returns

If the function runs without error, it returns ippStsNoErr.

If one of the following cases occurs, the function returns ippStsBadArgErr:

- Any pointer is NULL.
- Each of srcSize.width, srcSize.height, dstSize.width and dstSize.height <= 0.
- rcpRatiox <= 0 or rcpRatioy <= 0.
- pDst[0] is not aligned at 4 bytes boundary, pSrc[0] is not aligned at 4 bytes boundary.
- dstSize.width or dstSize.height is not even.
- srcStep[0] , srcStep[1] , srcStep[2]or dstStep[0] , dstStep[1] , dstStep[2]is less than 1.
- srcStep[0] is not multiple of 4, dstStep[0] is not multiple of 4.
- roiSize.width is larger than srcStep[0]; srcSize.width>>1 is larger than srcStep[1] or srcStep[2].
- Invalid values of one or more of the following control parameters:
  - interpolation or rotation.

- dstSize.height of output image is larger than dstStep[0], dstSize.height>>1 of output image is larger than dstStep[1] or dstStep[2] when rotation is ippCameraRotate90L or ippCameraRotate90R; dstSize.width of output image is larger than dstStep[0], dstSize.width>>1 of output image is larger than dstStep[1] or dstStep[2] when other valid rotation options.

NOTE. *Performance notes:*
To get better performance, the dstSize.width must be multiples of 4 and dstSize.height must be multiples of 4.

NOTE. *Size notes:*
The dstSize.width and dstSize.height must be even.

Alignment Requirement

The start address of pDst[0] must be aligned at 4-byte boundary and dstStep[0] must be multiple of 4.

The start address of pSrc[0] must be aligned at 4-byte boundary, and srcStep[0] must be multiple of 4.

Reciprocal Resizing Ratio Setting

rcpRatiox should be larger than 0 and should not be larger than ((((srcSize.width&~1)-1) / ((dstSize.width&~1)-1)) << 16).

rcpRatioy should be larger than 0 and should not be larger than ((((srcSize.height&~1)-1) / ((dstSize.height&~1)-1)) << 16).

It's recommended to set rcpRatiox to be ((((srcSize.width&~1)-1) / ((dstSize.width&~1)-1)) << 16) and set rcpRatioy to be ((((srcSize.height&~1)-1) / ((dstSize.height&~1)-1)) << 16).

Size of Output Image

dstSize is the size of destination image before rotation, that is, dstSize is the size of the image after resizing.

Example

Here is an example for how to call the ippiYCbCr420RszRot_8u_P3R.

```
#include <stdio.h>
#include <stdlib.h>
#include "ippIP.h"
#define  _ALIGN4(adr)    ((((Ipp32u)(adr))+3)&(~3))

IppStatus resize_rotate();

int main()
{
    resize_rotate();
}
IppStatus resize_rotate()
{
Ipp8u    *pSrc1, *pSrc2, *pSrc3, *pDst1, *pDst2, *pDst3;
Ipp8u    *pSrc[3], *pDst[3];
IppiSize srcSize, dstSize;
IppiRotationrotation;
IppCameraInterpolationinterpolation;
int      rcpRatioX, rcpRatioY;
    int      srcStep[3], dstStep, size;
int      status;

    /* Initialize parameters */
    srcSize.width  = 176;
    srcSize.height = 144;
    dstSize.width  = 320;
    dstSize.height = 240;
    srcStep[0]     = 176;
    srcStep[1]     = 88;
    srcStep[2]     = 88;
dstStep[0]     = 320;
dstStep[1]     = 160;
dstStep[2]     = 160;
interpolation  = ippCameraInterpBilinear;
```

```
rotation    = ippCameraRotateDisable;
    rcpRatioX =
   (int)(((double)((srcSize.width-1)<<16))/(dstSize.width-1));
rcpRatioY =
   (int)(((double)((srcSize.height-1)<<16))/(dstSize.height-1));

    pSrc1 = (Ipp8u*)malloc(srcSize.height*srcStep[0] + 8);
    pSrc2 = (Ipp8u*)malloc((srcSize.height>>1)*srcStep[1] + 8);
pSrc3 = (Ipp8u*)malloc((srcSize.height>>1)*srcStep[2] + 8);
pSrc[0] = (Ipp8u*)_ALIGN4(pSrc1);
pSrc[1] = pSrc2;
pSrc[2] = pSrc3;

size  = dstSize.height * dstStep[0] + 8;
pDst = (Ipp8u*)malloc(size);
size  = (dstSize.height>>1) * dstStep[1] + 8;
pDst2 = (Ipp8u*)malloc(size);
pDst3 = (Ipp8u*)malloc(size);

pDst[0] = (Ipp8u*)_ ALIGN4(pDst1);
pDst[1] = pDst2;
pDst[2] = pDst3;

/* here to initialize the content of pSrc[0], pSrc[1] and pSrc[2] */
pSrc[0] = ...;
pSrc[1] = ...;
pSrc[2] = ...;
/* here to initialize the content of pSrc[0], pSrc[1] and pSrc[2] */

    /* resize & rotate */
status = ippiYCbCr420RszRot_8u_P3R(pSrc, srcStep, srcSize, pDst,
 dstStep, dstSize, interpolate, rotation,
 rcpRatioX, rcpRatioY);
    return status;
}
```

Arbitrary ResizeRotateCsc

YCbCr420RszCscRotRGB_8u_P3C3R

Prototype

```
IppStatus ippiYCbCr420RszCscRotRGB_8u_P3C3R( Ipp8u *pSrc[3], int
   srcStep[3], IppiSize srcSize, void *pDst, int dstStep, IppiSize
   dstSize, IppCameraCsc colorConversion, IppCameraInterpolation
   interpolation, IppCameraRotation rotation, int rcpRatiox, int
   rcpRatioy);
```

Description

This function provides an image for previewing of video or still-capture applications or playback of video. In order to maximize bus bandwidth efficiency, several atomic image processing kernels have been combined into a single function. In particular, the following sequence of operations is applied to the raw input image:

1. Spatial resizing. First, the input image of srcSize is scaled to dstSize image using the interpolation methodology specified by the control parameter interpolation. The following interpolation schemes are supported: nearest neighbor, bilinear. For each of these, respectively, computational complexity and preview image quality rank from low to high.
2. Color space conversion. Following scaling, color space conversion is applied according to the control parameter colorConversion.
3. Rotation. After color space conversion, the preview output image is rotated according to the control parameter rotation.

The input data should be YCbCr420 planar format.

Input Arguments

- `pSrc` – a 3-element vector containing pointers to the start of each of the YCbCr420 input planes.
- `srcStep` – a 3-element vector containing the distance, in bytes, between the start of lines in each of the input image planes.
- `dstStep` – distance, in bytes, between the start of lines in the destination image.
- `srcSize` – dimensions, in pixels, of the source image.

- `dstSize` – dimensions, in pixels, of the destination image (before applying rotation to the resizing image).
- `interpolation` – interpolation methodology control parameter; must take one of the following values: ippCameraInterpNearest, or ippCameraInterpBilinear for nearest neighbor or bilinear interpolation, respectively.
- `colorConversion` – color conversion control parameter; must be set to one of the following pre-defined values: ippCameraCscYCbCr422ToRGB565, ippCameraCscYCbCr422ToRGB555, ippCameraCscYCbCr422ToRGB444, or ippCameraCscYCbCrToRGB888
- `rotation` – rotation control parameter; must be set to one of the following pre-defined values: . ippCameraRotateDisable, ippCameraRotate90L, ippCameraRotate90R, ippCameraRotate180, ippCameraFlipHorizontal, or ippCameraFlipVertical
- `rcpRatiox` – reciprocal resizing ratio in X direction, which is in Q16 format. If rcpRatiox>65536, it means the input image will expand in x direction.
- `rcpRatioy` – reciprocal resizing ratio in Y direction, which is in Q16 format. If rcpRatioy>65536, it means the input image will expand in y direction.

Output Arguments

- `pDst` – pointer to the start of the buffer containing the, resized, color-converted, and rotated output image.

Returns

If the function runs without error, it returns ippStsNoErr.

If one of the following cases occurs, the function returns ippStsBadArgErr:

- Any pointer is NULL.
- rcpRatiox <= 0 or rcpRatioy <= 0.
- Each of srcSize.width, srcSize.height, dstSize.width and dstSize.height <= 0.
- pDst is not aligned at 8 bytes boundary, pSrc[0] is not aligned at 4 bytes boundary, pSrc[1] is not aligned at 2 bytes boundary, or pSrc[2] is not aligned at 2 bytes boundary.
- srcStep[0] , srcStep[1] , srcStep[2]or dstStep is less than 1.
- srcStep[0] is not multiple of 4, srcStep[1] is not multiple of 2, or srcStep[2] is not multiple of 2.
- dstStep is not multiple 8 when colorConversion is ippCameraCscYCbCr422ToRGB565, ippCameraCscYCbCr422ToRGB555, or ippCameraCscYCbCr422ToRGB444; dstStep is not multiple of 2 when colorConversion is ippCameraCscYCbCrToRGB888.
- roiSize.width is larger than srcStep[0]; roiSize.width>>1 is larger than half of srcStep[1] or srcStep[2].

- Invalid values of one or more of the following control parameters:
  — interpolation, colorConversion or rotation.
- dstSize.height * bytes/pixel of output image is larger than dstStep when rotation is ippCameraRotate90L or ippCameraRotate90R; dstSize.width * bytes/pixel of output image is larger than dstStep when other valid rotation options.

NOTE. *Performance notes:*
To get better performance, the dstSize.width must be multiples of 4 and dstSize.height must be multiples of 2.

Alignment Requirement

The start address of pDst must be aligned at 8-byte boundary; and dstStep must be multiple of 8 when output format is RGB444/555/565, and dstStep must be multiple of 2 when output format is RGB888.

The start address of pSrc[0] must be aligned at 4-byte boundary, pSrc[1] must be aligned at 2-boundary and pSrc[2] must be aligned at 2-boundary; and srcStep[0] must be multiple of 4, srcStep[1] must be multiple of 2 and srcStep[2] must be multiple of 2.

Reciprocal Resizing Ratio Setting

rcpRatiox should be larger than 0 and should not be larger than ((((srcSize.width&~1)-1) / ((dstSize.width&~1)-1)) << 16).

rcpRatioy should be larger than 0 and should not be larger than ((((srcSize.height&~1)-1) / ((dstSize.height&~1)-1)) << 16).

It's recommended to set rcpRatiox to be ((((srcSize.width&~1)-1) / ((dstSize.width&~1)-1)) << 16) and set rcpRatioy to be ((((srcSize.height&~1)-1) / ((dstSize.height&~1)-1)) << 16).

Size of Output Image

dstSize is the size of destination image before rotation, that is, dstSize is the size of the image after resizing.

Example

Here is an example for how to call the ippiYCbCr420RszCscRotRGB_8u_P3C3R.

```
#include <stdio.h>
#include <stdlib.h>
#include "ippIP.h"
#define  _ALIGN8(adr)   ((((Ipp32u)(adr))+7)&(~7))
#define  _ALIGN4(adr)   ((((Ipp32u)(adr))+3)&(~3))

IppStatus resize_rotate();

int main()
{
    resize_rotate();
}
IppStatus resize_rotate()
{
Ipp8u    *pSrc1, *pSrc2, *pSrc3;
Ipp8u    *pSrc[3];
    void     *pDst;
IppiSize srcSize, dstSize;
IppCameraCsccolorConversion;
IppiRotationrotation;
IppCameraInterpolationinterpolation;
int      rcpRatioX, rcpRatioY;
    int      srcStep[3], dstStep, size;
int      status;

    /* Initialize parameters */
    srcSize.width  = 176;
    srcSize.height = 144;
    dstSize.width  = 320;
    dstSize.height = 240;
    srcStep[0]     = 176;
    srcStep[1]     = 88;
    srcStep[2]     = 88;
dstStep        = 640;
colorConversion= ippCameraCscYCbCr422ToRGB565;
interpolation  = ippCameraInterpBilinear;
rotation    = ippCameraRotateDisable;
```

```
    rcpRatioX =
   (int)(((double)((srcSize.width-1)<<16))/(dstSize.width-1));
rcpRatioY =
   (int)(((double)((srcSize.height-1)<<16))/(dstSize.height-1));
size           = dstSize.height * dstStep + 8;

    pSrc1 = (Ipp8u*)malloc(srcSize.height*srcStep[0] + 8);
    pSrc2 = (Ipp8u*)malloc((srcSize.height>>1)*srcStep[1] + 8);
pSrc3 = (Ipp8u*)malloc((srcSize.height>>1)*srcStep[2] + 8);
pSrc[0] = (Ipp8u*)_ALIGN4(pSrc1);
pSrc[1] = (Ipp8u*)_ALIGN4(pSrc2);
pSrc[2] = (Ipp8u*)_ALIGN4(pSrc3);

pDst = malloc(size);
pDst = (void*)_ALIGN8(pDst);

/* here to initialize the content of pSrc[0], pSrc[1] and pSrc[2] */
pSrc[0] = ...;
pSrc[1] = ...;
pSrc[2] = ...;
/* here to initialize the content of pSrc[0], pSrc[1] and pSrc[2] */

    /* resize & rotate */
status = ippiYCbCr420RszCscRotRGB_8u_P3C3R(pSrc, srcStep, srcSize, pDst,
 dstStep, dstSize, colorConversion, interpolate, rotation,
 rcpRatioX, rcpRatioY);

    return status;
}
```

YCbCr422toYCbCr420Rotate

YCbCr422ToYCbCr420Rotate_8u_C2P3R

Prototype

```
IppStatus ippiYCbCr422ToYCbCr420Rotate_8u_C2P3R(const Ipp8u *pSrc, int
   srcStep, Ipp8u *pDst[3], int dstStep[3], ippiSize roiSize, int
   rotation);
```

Description

This function decimates and interpolates the color space of the input image from YCbCr 422 to YCbCr 420, applies an optional rotation of -90, +90, or 180 degrees, and then rearranges the data from the pixel-oriented input format to a planar output format.

Input Arguments

- `pSrc` – pointer to the start of the buffer containing the pixel-oriented YCbCr422 input.
- `srcStep` – distance, in bytes, between the start of lines in the source image.
- `dstStep` – a 3-element vector containing the distance, in bytes, between the start of lines in each of the output image planes.
- `roiSize` – dimensions, in pixels, of the source and destination regions of interest.
- `rotation` – rotation control parameter; must be set to one of the following pre-defined values: ippCameraRotateDisable, ippCameraRotate90L, ippCameraRotate90R, or ippCameraRotate180.

OutputArguments

- `pDst` – a 3-element vector containing pointers to the start of each of the YCbCr420 output planes.

Returns

If the function runs without error, it returns ippStsNoErr

If one of the following cases occurs, the function returns ippStsBadArgErr:

- pSrc, or pDst[0] or pDst[1] or pDst[2] is NULL.

- pSrc or pDst[0] is not aligned at 8 bytes boundary; pDst[1] or pDst[2] is not aligned at 4-byte boundary.
- srcStep or dstStep[1], or dstStep[2], or dstStep[3] is less than 1; srcStep or dstStep[0] is not multiple of 8; dstStep[1] or dstStep[2] is not multiple of 4.
- roiSize.width is larger than half of srcStep.
- Invalid values of rotation control parameters.
- dstStep[0] is less than roiSize.width of downscaled, color-converted and rotated image; Half of the dstStep[1] or half of the dstStep[2] is less than roiSize.width of downscaled, color-converted and rotated image.
- RoiSize.width or roiSize.height is less than 8.

NOTE. *Input Bytes Order:*
*Input (*pSrc) byte ordering is Cb Y Cr Y, which is supported by Intel® PXA27x processor. The memory organization for the packed format is illustrated in* Table 16-19.

Alignment Requirement

pSrc, pDst[0], srcStep and dstStep[0] should be 8 bytes aligned; pDst[1], pDst[2], dstStep[1] and dstStep[2] should be 4 bytes aligned.

Size of Output Image

If the roiSize.width or roiSize.height cannot be divided by 8 exactly, it will be cut to be multiple of 8.

YCbCr422ToYCbCr420Rotate_8u_P3R

Prototype

```
IppStatus ippiYCbCr422ToYCbCr420Rotate_8u_P3R(const Ipp8u *pSrc[3], int
   srcStep[3], Ipp8u *pDst[3], int dstStep[3], ippiSize roiSize, int
   rotation);
```

Description

This function decimates and interpolates the color space of the input image from YCbCr 422 planar data to YCbCr 420 planar data, and then applies an optional rotation of -90, +90, or 180 degrees. Difference between this function and ippiYCbCr422ToYCbCr420Rotate_8u_C2P3R is that this function supports the input YCbCr422 format in planar order.

Input Arguments

- `pSrc` – a 3-element vector containing pointers to the start of each of the YCbCr420 input planes.
- `srcStep` – a 3-element vector containing the distance, in bytes, between the start of lines in each of the input image planes.
- `dstStep` – a 3-element vector containing the distance, in bytes, between the start of lines in each of the output image planes.
- `roiSize` – dimensions, in pixels, of the source and destination regions of interest.
- `rotation` – rotation control parameter; must be set to one of the following pre-defined values: ippCameraRotateDisable, ippCameraRotate90L, ippCameraRotate90R, or ippCameraRotate180.

OutputArguments

- `pDst` – a 3-element vector containing pointers to the start of each of the YCbCr420 output planes.

Returns

If the function runs without error, it returns ippStsNoErr.

If one of the following cases occurs, the function returns ippStsBadArgErr:

- Any pointer is NULL.
- pSrc[0] or pDst[0] is not aligned at 8 bytes boundary; pSrc[1], pSrc[2], pDst[1] or pDst[2] is not aligned at 4-byte boundary.
- Any of the steps is less than 1; srcStep[0] or dstStep[0] is not multiple of 8; srcStep[1], srcStep[2], dstStep[1] or dstStep[2] is not multiple of 4.
- roiSize.width is larger than srcStep[0]; roiSize.width is larger than half of srcStep[1] or half of srcStep[2].
- Invalid values of rotation control parameters.
- dstStep[0] is less than roiSize.width of downscaled, color-converted and rotated image; Half of the dstStep[1] or half of the dstStep[2] is less than roiSize.width of downscaled, color-converted and rotated image.
- roiSize.width or roiSize.height is less than 8.

NOTE. *Input Image Order:*
Input image is of YCbCr422 format in planar order. That is, pSrc[0] is buffer for Y plane, pSrc[1] is buffer for Cb plane and pSrc[2] is buffer for Cr plane

Alignment Requirement

pSrc[0], pDst[0], srcStep[0] and dstStep[0] should be 8 bytes aligned; pSrc[1], pSrc[2], pDst[1], pDst[2], srcStep[1], srcStep[2], dstStep[1] and dstStep[2] should be 4 bytes aligned.

Size of Output Image

If the roiSize.width or roiSize.height cannot be divided by 8 exactly, it will be cut to be multiple of 8.

Example

The following is an example of how to call the ippiYCbCr422ToYCbCr420_Rotate_C2P3R:

```
#include <stdlib.h>
#include "ippIP.h"

#define  _ALIGN8(adr)    ((((Ipp32u)(adr))+7)&(~7))
#define  _ALIGN4(adr)    ((((Ipp32u)(adr))+3)&(~3))

int main()
{
int W, H;
Ipp8u  *pDst1, *pDst2, *pDst3;

/*   Variables used in Function   */
Ipp8u              *pSrc, *pDst[3];
int                      srcStep, dstStep[3];
IppiSize           roiSize;
IppCameraInterpolation rotation;

       /*   Initialize the width and height of input image   */
W = 320;
```

```
H = 240;

/*   Initialize source parameters   */
rotation  = ippCameraRotate180;
roiSize.width = W;
roiSize.height = H;
srcStep = 640;
dstStep[0] = 320;
dstStep[1] = 160;
dstStep[2] = 160;

/*   Allocate Buffer   */
pSrc = (Ipp8u*) malloc(W*H*2+7);
pDst1 = (Ipp8u*) malloc(W*H+7);
pDst2 = (Ipp8u*) malloc(W*H/2+7);
pDst3 = (Ipp8u*) malloc(W*H/2+7);

/*   Align buffer*/
pSrc = (Ipp8u*)_ALIGN8(pSrc);
pDst1 = (Ipp8u*)_ALIGN8(pDst1);
pDst2 = (Ipp8u*)_ALIGN4(pDst2);
pDst3 = (Ipp8u*)_ALIGN4(pDst3);

pDst[0] = pDst1;
pDst[1] = pDst2;
pDst[2] = pDst3;

/*   Call ippiYCbCr422ToYCbCr420Rotate_8u_C2P3R  */
ippiYCbCr422ToYCbCr420Rotate_8u_C2P3R(pSrc, srcStep, pDst, dstStep,
   roiSize, rotation);

/*   Free Buffer*/
free(pSrc);
free(pDst1);
free(pDst2);
free(pDst3);

return (0);
```

```
}
```

JPEG Image CODEC 17

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) on Intel® PCA Processors with Intel® Wireless MMX™ (PCA processors with MMX™).This product is designed to support image CODECs that are compliant with the ISO/IEC International Standard 10918 or so called JPEG CODEC. Like all the other primitives, the JPEG primitives focus on well-defined and the computational expensive operations in JPEG CODEC.

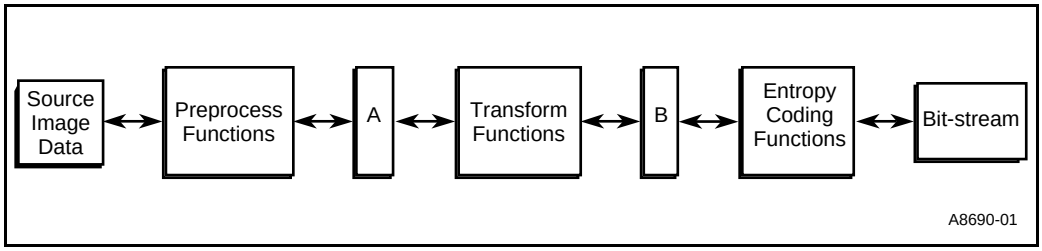
IPP JPEG primitives support the following features:

- interleave and non-interleave
- DCT (Discrete Cosine Transform)-based sequential
- DCT-based progressive
- entropy coding based on Huffman coding algorithm
- processing of a Rectangle of Interest (ROI).

High Level Description

The major building blocks of the JPEG CODEC system and the steps to convert images into JPEG bit-stream are shown in Figure 17-1:

Figure 17-1 Major Blocks in JPEG CODEC System



Each of the function set shown in Figure 17-1 can be described in detail as follows,

- **Preprocess functions** include copy function set and color conversion function set. These function sets are considered as helper functions, which convert the source image:
  a. Color space to the special color space for CODEC (YCbCr is general used, but it is not necessary, user can use its own color space).
  b. Convert input image data format to internal coefficient storage format: RGB <> YCbCr, RGB <> YCbCr422, RGB <> YCbCr411 are supported in this version. IPP JPEG CODEC can process other color space with complex sub-sampling, but you should write your own preprocess functions.
- **Transform functions** include DCT and Quantization function set.
- **Entropy coding functions** include Huffman CODEC functions that support both of sequential mode and progressive mode.
- **A, B** are internal coefficient buffers (CBs). See "IPP JPEG Coefficient Buffer (CB)" for a detailed explanation.

IPP JPEG Coefficient Buffer (CB)

The coefficient buffer defined for IPP JPEG CODEC is in signed 16-bit format. All coefficients are stored as 8x8 blocks in the coefficient buffer. Figure 17-2 shows the data formats of the coefficient buffer for interleave and non-interleave with different sub-sample rates.

Figure 17-2 Interleaved and Non-Interleaved Image Data Formats

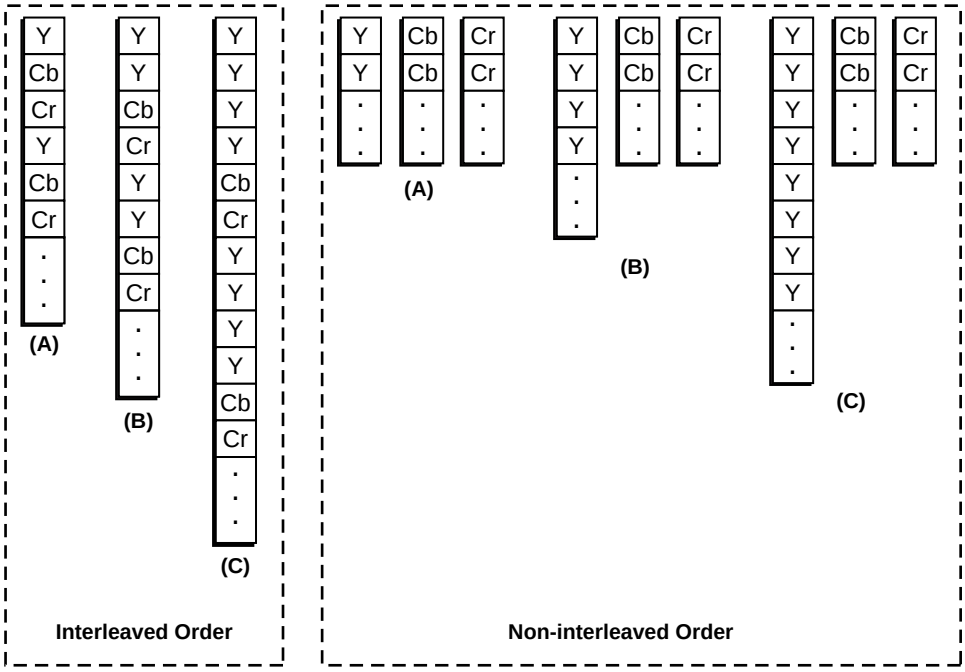


Figure 17-2 shows the data arrangement order for both interleaved and non-interleaved images, which can be described in detail as follows:

- **(A)** is normal YCbCr data with non-subsampling. Each MCU contains 3 blocks (1 Y, 1Cb, 1Cr) for non-interleaved order and one block (1Y or 1Cb or 1Cr) for interleave order.
- **(B)** is YCbCr data with 4:2:2 sub-sampling. Each MCU contains 4 blocks (2 Y, 1 Cb, 1 Cr) for non-interleaved order and one block (1Y or 1Cb or 1Cr) for interleave order (2N blocks in Y buffer, N blocks in Cb or Cr buffer).
- **(C)** is YCbCr data with 4:1:1 sub-sampling. Each MCU contains 6 blocks (4 Y, 1 Cb, 1 Cr) for non-interleaved order and one block (1Y or 1Cb or 1Cr) for interleave order (4N blocks in Y buffer, N blocks in Cb or Cr buffer).

The width of each coefficient buffer is equal to the block width (8 in Ipp16s). The height of each coefficient buffer is equal to the block height (8 lines) times the number of blocks. This has been specially designed to optimize D-Cache. The start address of each coefficient buffer must be 8-byte aligned. It is better to align at a 32-byte boundary (recommended on Intel® Integrated Performance Primitives on Intel® PCA Processors with Intel® Wireless MMX™ Technology).

IPP JPEG Primitive Usage Mode

There are five groups of IPP JPEG primitives based on functionality. See Table 17-1.

- Helper function set groups:
  - Copy and Expand function set: It performs block boundary processing for the CODEC. It also performs the processing of ROI.
  - Color Conversion function set: It performs the conversion between the special color space and YCbCr. The level-shift and sub-sampling with color conversion were also merged to gain performance.
- CODEC function set groups:
  - DCT and Quantization function set (DCTQ): It performs transform and quantization generating coefficients for entropy coding.
  - Huffman Coding function set (HUFFC): It performs entropy coding for the coefficients generated by DCTQ.

Although YCbCr is recommended as a default color space for JPEG CODEC, users have the freedom to use other color spaces with IPP JPEG functions.

Some typical JPEG encode systems supported by JPEG primitives (assuming the image data is RGB) are the following:

- DCT-based sequential coding with Low memory required shown in Figure 17-3.
- DCT-based sequential coding shown in Figure 17-4,
- DCT-based progressive coding shown in Figure 17-5,

Figure 17-3 shows a system that requires one CB of one MCU size.

Figure 17-3 DCT-based Sequential System A (Low memory required)

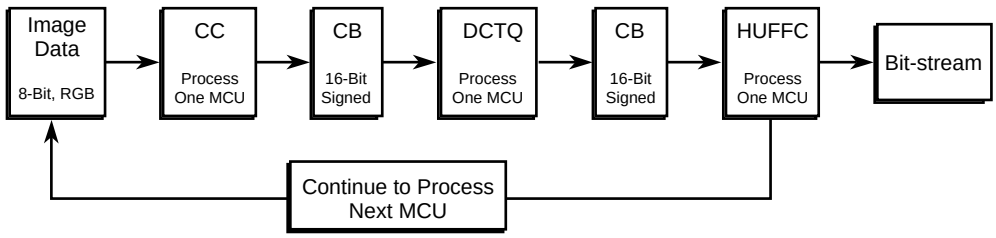


Figure 17-4 DCT-based Sequential System B

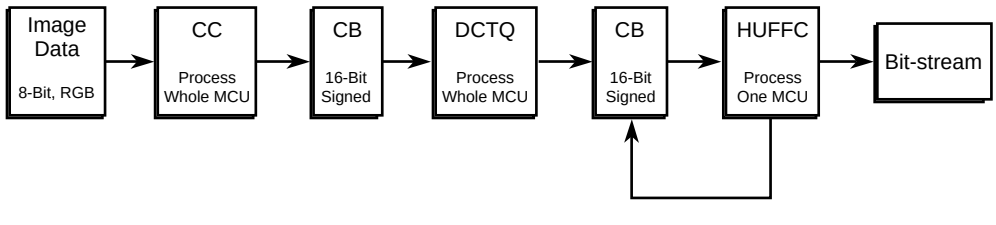
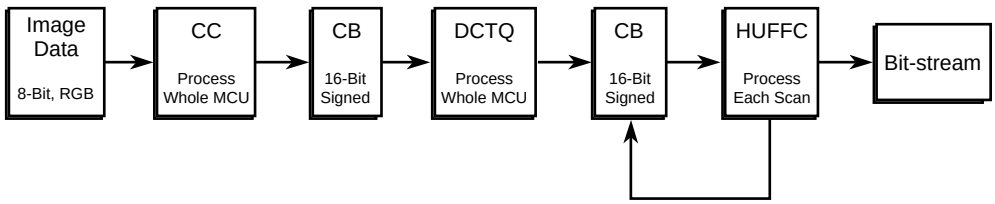


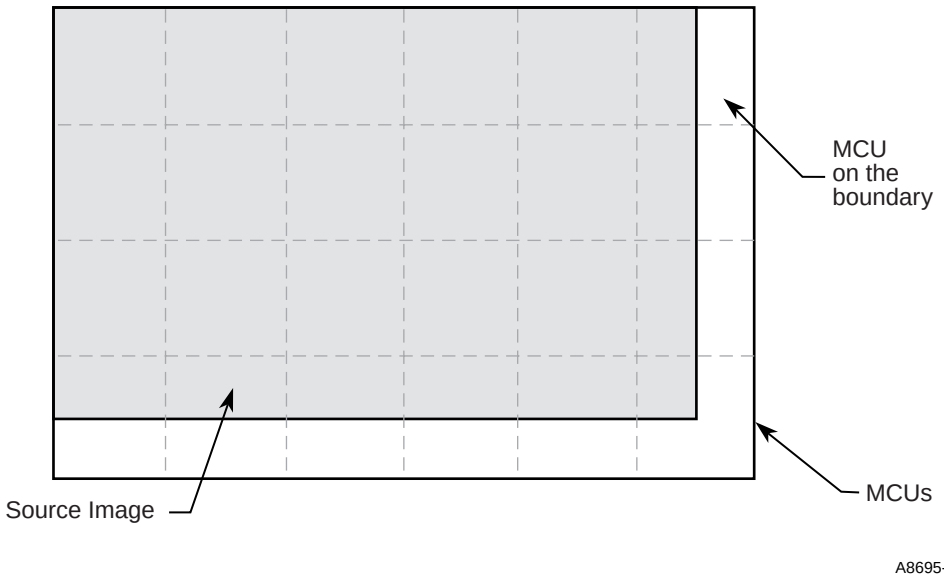
Figure 17-5 DCT-based Progressive System



How to Process MCU on the Boundary

Figure 17-6 shows an image with 6*4 MCU blocks.

Figure 17-6 MCU On the Boundary



We suggest the following:

- Encoding:
  - — If MCU is on the boundary

    ```
    Image Data Buffer (RGB input, for example) ->Expand function
    (Expand it to MCU size) -> Full MCU Data Color Conversion ->
    Encoder
    ```

  - — If MCU is within the boundary,

    ```
    Image Data Buffer (RGB input, for example) -> Full MCU Data
    Color Conversion -> Encoder
    ```

- Decoding:
  - — If MCU is on the boundary

    ```
    Decoded Image Data (MCU, RGB input) -> Copy ROI from MCU -> Image
    Data Buffer (RGB output).
    ```

  - — If MCU is within the boundary,

    ```
    Decoded Image Data (MCU, RGB input) -> Image Data Buffer (RGB
    output).
    ```

Figure 17-7 Rectangle Of Interest (ROI) for Encoding Procedure

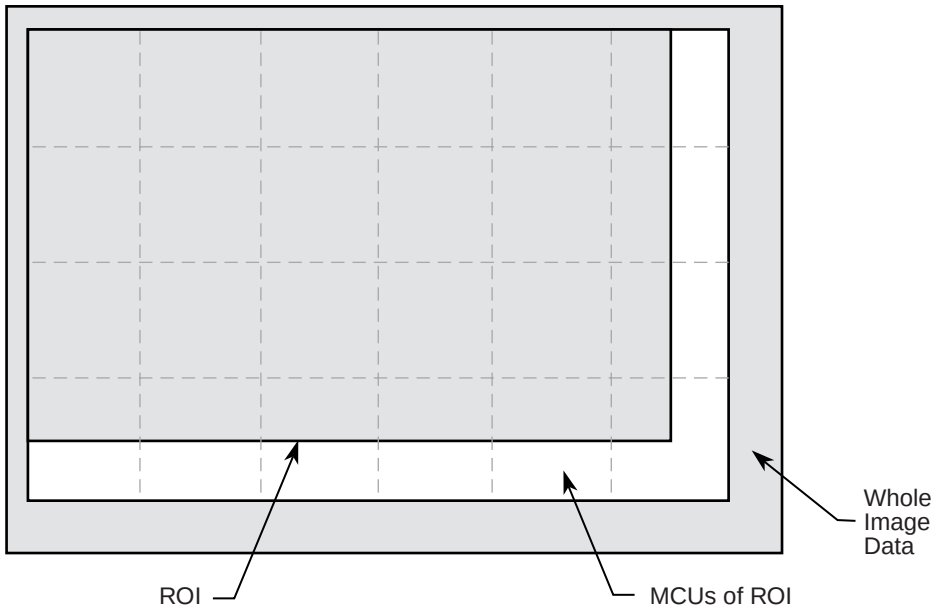


Figure 17-7 illustrates the relationship between ROI, MCU of ROI, and the whole image data:

These parameters are used to indicate ROI:

1. Start Pointer which points to the start address of ROI in the image data buffer;
2. Width is less than or equal to the width of the image.
3. Height is less than or equal to the height of the image.
4. Line Step of image data buffer.

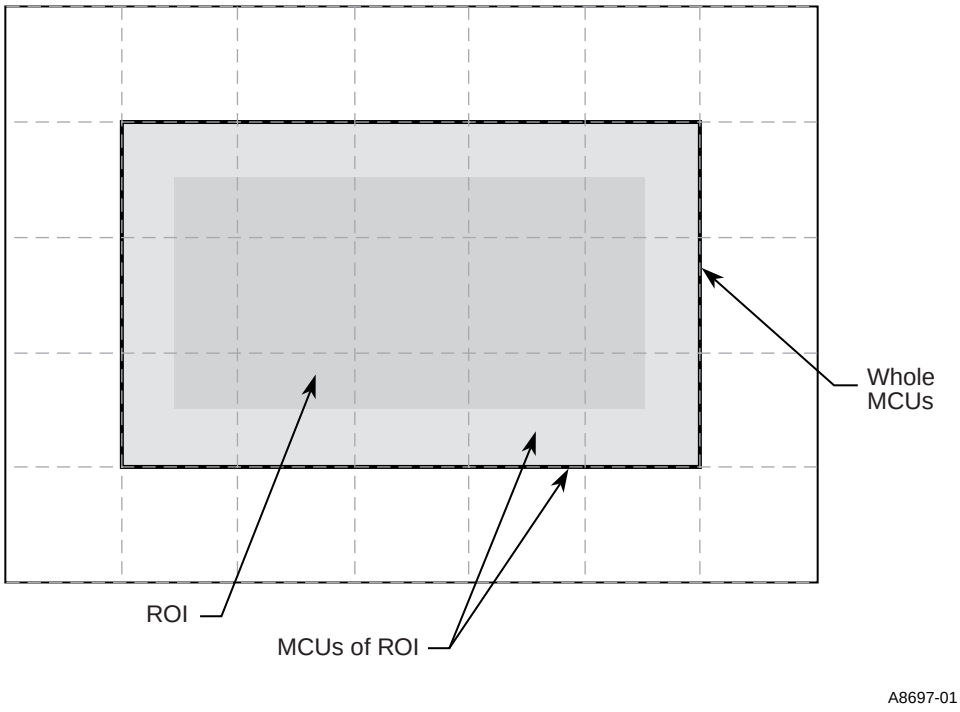
The procedure is:

1. Move the Start Pointer to indicate the start address of ROI.
2. Process MCUs on the boundary with padding function (Start Point, Width, Height, Line Step).
3. Encode each MCU

Suggested Usage Mode in Processing ROI

Figure 17-8 shows a ROI in an image to be decoded.

Figure 17-8 An Image with ROI to be Decoded



To decode a ROI, the following procedure is suggested:

1. Decode MCUs covered by ROI with Huffman decoder.
2. Decode image data from each MCU.
3. Copy the data of full MCU or ROI of the MCU (in the boundary) data to image data buffer.

Definitions and Structure Type Declarations

IppiEncodeHuffmanSpec

```
typedef struct _IPPIENCODEHUFFMANSPEC
{
     Ipp32u huffSizeCode[256];
}IppiEncodeHuffmanSpec;
```

IppiDecodeHuffmanSpec

```
typedef struct _IPPIDECODEHUFFMANSPEC
{
     Ipp32u lookBitsVal[256];
     Ipp16u huffVal[256];
     Ipp32u mincodeptr[18];
     Ipp16u maxcode[18];
}IppiDecodeHuffmanSpec;
```

Function Sets

Table 17-1 describes the five IPP set groups discussed in the "IPP JPEG Primitive Usage Mode" section.

Table 17-1 JPEG CODEC APIs for IPP

Helper and CODEC Set Group		Function	Page	Description
Copy & Expand		ippiCopyExpand_8u_C3	17-13	Copy, Expand pixels for MCU on the boundary
Color Conversion	BGR -> YCbCr	ippiBGRToYCbCr444LS_MCU_8u16s_C3P3R ippiBGRToYCbCr422LS_MCU_8u16s_C3P3R ippiBGRToYCbCr411LS_MCU_8u16s_C3P3R	17-16	Convert one MCU image data
		ippiBGR555ToYCbCr444LS_MCU_16u16s_C3P3R; ippiBGR555ToYCbCr422LS_MCU_16u16s_C3P3R; ippiBGR555ToYCbCr411LS_MCU_16u16s_C3P3R;	17-18	
		ippiBGR565ToYCbCr444LS_MCU_16u16s_C3P3R; ippiBGR565ToYCbCr422LS_MCU_16u16s_C3P3R; ippiBGR565ToYCbCr411LS_MCU_16u16s_C3P3R;	17-20	
	YCbCr-> BGR	ippiYCbCr444ToBGRLS_MCU_16s8u_P3C3R ippiYCbCr422ToBGRLS_MCU_16s8u_P3C3R ippiYCbCr411ToBGRLS_MCU_16s8u_P3C3R	17-22	Process one MCU data
		ippiYCbCr444ToBGR555LS_MCU_16s16u_P3C3R; ippiYCbCr422ToBGR555LS_MCU_16s16u_P3C3R; ippiYCbCr411ToBGR555LS_MCU_16s16u_P3C3R;	17-24	
		ippiYCbCr444ToBGR565LS_MCU_16s16u_P3C3R; ippiYCbCr422ToBGR565LS_MCU_16s16u_P3C3R; ippiYCbCr411ToBGR565LS_MCU_16s16u_P3C3R;	17-26	
DCT_Quant	Encode	ippiDCTQuantFwdTableInit_JPEG_8u16u	17-28	Init quantization table to the optimized DCT
		ippiDCTQuantFwd_JPEG_16s ippiDCTQuantFwd_JPEG_16s_I	17-30	Process one block
	Decode	ippiDCTQuantInvTableInit_JPEG_16u32s	17-31	Init quantization table to the optimized IDCT
		ippiDCTQuantInv_JPEG_16s ippiDCTQuantInv_JPEG_16s_I	17-33	Process one block
Huffman	Encode	ippiEncodeHuffmanSpecGetBufSize_JPEG_8u	17-34	Generate size of Huffman encoding table
		ippiEncodeHuffmanStateGetBufSize_JPEG_8u	17-35	Generate size of Huffman decoding state

Table 17-1 JPEG CODEC APIs for IPP (continued)

		ippiEncodeHuffmanStateInit_JPEG_8u	17-36	Init Huffman encoding state
		ippiEncodeHuffmanSpecInit_JPEG_8u	17-36	Initialize huffman table for encoding
		ippiEncodeHuffman8x8_Direct_JPEG_16s1u_C1	17-38	For sequential processing
		ippiEncodeHuffman8x8_DCFirst_JPEG_16s1u_C1	17-40	Implement Huffman DC encoding of first scan for progressive mode.
		ippiEncodeHuffman8x8_DCRefine_JPEG_16s1u_C1	17-42	Implement Huffman DC encoding of refined scans
		ippiEncodeHuffman8x8_ACFirst_JPEG_16s1u_C1	17-43	Implement Huffman AC encoding of first scans
		ippiEncodeHuffman8x8_ACRefine_JPEG_16s1u_C1	17-44	Implement Huffman AC encoding of refined scans
	Decode	ippiDecodeHuffmanSpecGetBufSize_JPEG_8u	17-46	Generate size of Huffman decoding table
		ippiDecodeHuffmanStateGetBufSize_JPEG_8u	17-46	Generate size of Huffman decoding table
		ippiDecodeHuffmanStateInit_JPEG_8u	17-47	Init Huffman decoding state
		ippiDecodeHuffmanSpecInit_JPEG_8u	17-48	Initialize Huffman table for decoding
		ippiDecodeHuffman8x8_Direct_JPEG_1u16_C1	17-49	
		ippiDecodeHuffman8x8_DCFirst_JPEG_1u16s_C1	17-51	Implement Huffman DC decoding of first scan
		ippiDecodeHuffman8x8_DCRefine_JPEG_1u16s_C1	17-53	Implement Huffman DC decoding of refined scans
		ippiDecodeHuffman8x8_ACFirst_JPEG_1u16s_C1	17-54	Implement Huffman AC decoding of first scans
		ippiDecodeHuffman8x8_ACRefine_JPEG_1u16s_C1	17-55	Implement Huffman AC decoding of first scans

Helper Function Set Group

Copy Function Set (COPAD)

CopyExpand_8u_C3

Prototype

```
IppStatus ippiCopyExpand_8u_C3(const Ipp8u *pSrc, int srcStep, IppiSize
   srcSize, Ipp8u *pDst, int dstStep, IppiSize dstSize);
```

Description

This function processes the image data block on the boundary. The values, which are situated outside of the image boundary, are replicated from the sample values at the boundary to provide missing edge values. This function just processes 3 channels image data in interleaved order.

NOTE. Both of source image data buffer and destination image data buffer support down-top storage format. In that case, srcStep and dstStep can be less than 0.

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.
- Each size is larger than zero.
- The destination size is larger than or equal to source size.
- The absolute values of srcStep and dstStep are larger than or equal to 3.

Input Arguments

- `pSrc` – identifies source image data buffer
- `srcStep` – specifies the number of bytes in a line of the image data buffer
- `srcSize` – identifies `IppiSize` data structure to indicate the size of source rectangle
- `dstStep` – specifies the number of bytes in a line of output MCU buffer
- `dstSize` – identifies `IppiSize` data structure to indicate the size of destination rectangle

Output Arguments

`pDst` – identifies the output MCU buffer

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - one or more vectors was 0 in length.
  - the source size was smaller than the destination size
  - absolute values of `srcStep` and `dstStep` were smaller than 3

Color Conversion Function Set (CC)

Convert RGB to YCbCr in the CCIR 601 color space. The following equations specify the mathematical definition of forward and inverse.

(Y, Cb and Cr belong to [0, 1].)

- RGB -> YCbCr:
  - Y = 0.29900 * R + 0.58700 * G + 0.11400 * B
  - Cb = -0.16874 * R - 0.33126 * G + 0.50000 * B + 0.5
  - Cr = 0.50000 * R - 0.41869 * G - 0.08131 * B + 0.5
- YCbCr -> RGB:
  - R = Y + 1.402(Cr-0.5)
  - G = Y – 0.34414(Cb-0.5) –0.71414(Cr-0.5)
  - B = Y + 1.772(Cb-0.5)

Level shift was integrated into IPP JPEG color conversion function set to gain higher performance. So the equations were modified as follows:

(Y, Cb and Cr belong to [-0.5, 0.5].)

- RGB -> YCbCr:
  - Y = 0.29900 * R + 0.58700 * G + 0.11400 * B – 0.5
  - Cb = -0.16874 * R - 0.33126 * G + 0.50000 * B
  - Cr = 0.50000 * R - 0.41869 * G - 0.08131 * B
- YCbCr -> RGB:
  - R = (Y+0.5) + 1.402*Cr
  - G = (Y+0.5) – 0.34414*Cb –0.71414*Cr
  - B = (Y+0.5) + 1.772*Cb

BGRToYCbCr444LS_MCU_8u16s_C3P3R
BGRToYCbCr422LS_ MCU_8u16s_C3P3R
BGRToYCbCr411LS_MCU_8u16s_C3P3R

Prototype

```
IppStatus ippiBGRToYCbCr444LS_MCU_8u16s_C3P3R (const Ipp8u * pSrcBGR,
   int srcStep, Ipp16s * pDstMCU[3]);
IppStatus ippiBGRToYCbCr422LS_MCU_8u16s_C3P3R (const Ipp8u * pSrcBGR,
   int srcStep,  Ipp16s * pDstMCU[3]);
IppStatus ippiBGRToYCbCr411LS_MCU_8u16s_C3P3R (const Ipp8u * pSrcBGR,
   int srcStep,  Ipp16s * pDstMCU[3]);
```

Description

These functions convert BGR data to YCbCr data with level-shift and sampling (in 4:1:1 and 4:2:2). They process image data in MCU size, 8x8 for 444; 16x8 for 422; 16x16 for 411.

Input Arguments

- `pSrcBGR` – identifies source image data buffer. The source image data are stored in interleaved order as BGRBGRBGR… The image data buffer `pSrcBGR` can support down-top storage formats. In that case, `srcStep` can be less than 0.
- `srcStep` – specifies the number of bytes in a line of the image data buffer. This buffer contains the current block, which is identified by *pSrcBGR*. This value can be less than 0 to support down to top storage format.

Output Arguments

`pDstMCU[3]` – identifies a three-pointer array in sequence. The start address of each pointer in `pDstMCU[]` must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize the D-Cache access. The destination buffers identified by `pDstMCU[]` support top-down storage format only.

- `pDstMCU[0]` points to Y block
- `pDstMCU[1]` points to Cb block
- `pDstMCU[2]` points to Cr block

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - `The absolute value of srcStep was smaller than 24 (for 444) or 48 (for 422 and 411)`
  - the start address of each pointer in `pDstMCU[]` was not 8-byte aligned

NOTE. The start address of each pointer in `pDstMCU[]` must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. The image data buffer pSrcBGR can support down-top storage formats. In this case, srcStep can be less than 0. The destination buffers identified by pDstMCU[] just support top-down storage format only.

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer and each pointer in pointer array cannot be NULL.
- The absolute value of srcStep must be larger than or equal to 24 (for 444) or 48 (for 422 and 411).
- The start address of each pointer in pDstMCU[] must be 8-byte aligned.

BGR555ToYCbCr444LS_MCU_16u16s_C3P3R
BGR555ToYCbCr422LS_MCU_16u16s_C3P3R
BGR555ToYCbCr411LS_MCU_16u16s_C3P3R

Prototype

```
IppStatus ippiBGR555ToYCbCr444LS_MCU_16u16s_C3P3R (const Ipp16u * pSrcBGR,
   int srcStep,Ipp16s * pDstMCU[3]);
IppStatus ippiBGR555ToYCbCr422LS_MCU_16u16s_C3P3R (const Ipp16u * pSrcBGR,
   int srcStep, Ipp16s * pDstMCU[3]);
IppStatus ippiBGR555ToYCbCr411LS_MCU_16u16s_C3P3R (const Ipp16u * pSrcBGR,
   int srcStep, Ipp16s * pDstMCU[3]);
```

Description

These functions convert packed BGR555 data to YCbCr data with level-shift and sampling (in 4:1:1 and 4:2:2). They process image data in MCU size, 8x8 for 444; 16x8 for 422; 16x16 for 411.

Input Arguments

- `pSrcBGR` – identifies source image data buffer. The source image data are stored in interleaved order as [x B G R x B G R …], where x represents a vacant bit, and B, G, R are expressed in 5 bits, respectively. The image data buffer `pSrcBGR` can support down-top storage formats. In that case, `srcStep` can be less than 0.
- `srcStep` – specifies the number of bytes in a line of the image data buffer. This buffer contains the current block, which is identified by `pSrcBGR`. This value can be less than 0 to support down to top storage format.

Output Arguments

`pDstMCU[3]` – identifies a three-pointer array in sequence. The start address of each pointer in `pDstMCU[]` must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access. The destination buffers identified by `pDstMCU[]` support top-down storage format only.

- `pDstMCU[0]`– points to Y block
- `pDstMCU[1]`– points to Cb block
- `pDstMCU[2]` – points to Cr block

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - srcStep was an odd number or its absolute value was less than 16 (for 444) or 32 (for 422 and 411).
  - a pointer in `pDstMCU[]` was not 8-byte aligned.

NOTE. The start address of each pointer in pDstMCU[] must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. The image data buffer pSrcBGR can support down-top storage formats. In this case, srcStep can be less than 0. The destination buffers which identified by pDstMCU[] just supports top-down storage format.

NOTE. If one or more of the following conditions is true, this function returns ippStsBadArgErr.

- Any pointer is NULL.
- srcStep is odd or its absolute value is less than 16 (for 444) or 32 (for 422 and 411).
- Any pointer in pDstMCU[] is not 8-byte aligned.

BGR565ToYCbCr444LS_MCU_16u16s_C3P3R
BGR565ToYCbCr422LS_MCU_16u16s_C3P3R
BGR565ToYCbCr411LS_MCU_16u16s_C3P3R

Prototype

```
IppStatus ippiBGR565ToYCbCr444LS_MCU_16u16s_C3P3R (const Ipp16u * pSrcBGR,
   int srcStep,Ipp16s * pDstMCU[3]);
IppStatus ippiBGR565ToYCbCr422LS_MCU_16u16s_C3P3R (const Ipp16u * pSrcBGR,
   int srcStep, Ipp16s * pDstMCU[3]);
IppStatus ippiBGR565ToYCbCr411LS_MCU_16u16s_C3P3R (const Ipp16u * pSrcBGR,
   int srcStep,Ipp16s * pDstMCU[3]);
```

Description

These functions convert packed BGR565 data to YCbCr data with level-shift and sampling (in 4:1:1 and 4:2:2). They process image data in MCU size, 8x8 for 444; 16x8 for 422; 16x16 for 411.

Input Arguments

- `pSrcBGR` – identifies source image data buffer. The source image data are stored in interleaved order as [B G R B G R …], where G is expressed by 6 bits, and B and R are expressed in 5 bits, respectively. The image data buffer `pSrcBGR` can support down-top storage formats. In that case, `srcStep` can be less than 0.
- `srcStep` – specifies the number of bytes in a line of the image data buffer. This buffer contains the current block, which is identified by `pSrcBGR`. This value can be less than 0 to support down to top storage format.

Output Arguments

`pDstMCU[3]` – identifies a three-pointer array in sequence. The start address of each pointer in `pDstMCU[]` must be 8-byte aligned. The destination buffers identified by `pDstMCU[]` support top-down storage format only.

- `pDstMCU[0]` points to Y block.
- `pDstMCU[1]` points to Cb block.
- `pDstMCU[2]` points to Cr block.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - srcStep was an odd number or its absolute value was less than 16 (for 444) or 32 (for 422 and 411).
  - a pointer in `pDstMCU[]` was not 8-byte aligned.

NOTE. The start address of each pointer in pDstMCU[] must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. The image data buffer pSrcBGR can support down-top storage formats. In this case, srcStep can be less than 0. The destination buffers which identified by pDstMCU[] just supports top-down storage format.

NOTE. If one or more of the following conditions is true, this function returns ippStsBadArgErr.

- Any pointer is NULL.
- srcStep is odd or its absolute value is less than 16 (for 444) or 32 (for 422 and 411).
- Any pointer in pDstMCU[] is not 8-byte aligned.

YCbCr444ToBGRLS_MCU_16s8u_P3C3R
YCbCr422ToBGRLS_MCU_16s8u_P3C3R
YCbCr411ToBGRLS_MCU_16s8u_P3C3R

Prototype

```
IppStatus ippiYCbCr444ToBGRLS_MCU_16s8u_P3C3R(const Ipp16s
   *pSrcMCU[3],Ipp8u * pDstBGR, int dstStep);
IppStatus ippiYCbCr422ToBGRLS_MCU_16s8u_P3C3R(const Ipp16s *pSrcMCU[3],
   Ipp8u * pDstBGR, int dstStep);
IppStatus ippiYCbCr411ToBGRLS_MCU_16s8u_P3C3R(const Ipp16s *pSrcMCU[3],
   Ipp8u * pDstBGR, int dstStep);
```

Description

These functions convert YCbCr data to BGR data with level-shift and sampling (in 4:1:1 and 4:2:2). They process image data in MCU size, 8x8 for 444; 16x8 for 422; 16x16 for 411.

Input Arguments

- `pSrcMCU[3]` – identifies a three-pointer array in sequence. The start address of each pointer in `pSrcMCU[]` must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access. The source data buffers identified by `pSrcMCU[]` support top-down format only.
  - — `pSrcMCU[0]` points to Y block
  - — `pSrcMCU[1]` points to Cb block
  - — `pSrcMCU[2]` points to Cr block
- `dstStep` – specifies the number of bytes in a line of the image data buffer. This buffer contains the current block, which is identified by `pDstBGR`. This value can be less than 0 to support down to top storage format.

Output Arguments

`pDstBGR` – identifies output image data buffer. The source image data are stored in interleaved order as BGRBGRBGR… Each output value in `pDstBGR` is saturated at [0, 255]. The destination image data buffer `pDstBGR` can support down-top storage formats. In that case, `srcStep` can be less than 0.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - The absolute value of dstStep was smaller than 24 (for 444) or 48 (for 422 and 411).
  - the start address of a pointer in `pDstMCU[]` was not 8-byte aligned.

NOTE. Each output value in pDstBGR is saturated at [0, 255].

NOTE. The start address of each pointer in pSrcMCU[] must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. The destination image data buffer pDstBGR can support down-top storage formats. In this case, dstStep can be less than 0. The source data buffers, which identified by pSrcMCU[] just support top-down format.

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer and each pointer in pointer array cannot be NULL.
- The absolute value of srcStep must be larger than or equal to 24 (for 444) or 48 (for 422 and 411).
- The start address of each pointer in pSrcMCU[] must be 8-byte aligned.

YCbCr444ToBGR555LS_MCU_16s16u_P3C3R
YCbCr422ToBGR555LS_MCU_16s16u_P3C3R
YCbCr411ToBGR555LS_MCU_16s16u_P3C3R

Prototype

```
IppStatus ippiYCbCr444ToBGR555LS_MCU_16s16u_P3C3R (const Ipp16s *pSrcMCU[3],
    Ipp16u * pDstBGR, int dstStep);
IppStatus ippiYCbCr422ToBGR555LS_MCU_16s16u_P3C3R (const Ipp16s *pSrcMCU[3],
    Ipp16u * pDstBGR, int dstStep);
IppStatus ippiYCbCr411ToBGR555LS_MCU_16s16u_P3C3R (const Ipp16s *pSrcMCU[3],
    Ipp16u * pDstBGR, int dstStep);
```

Description

These functions convert YCbCr data to BGR555 data to with level-shift and sampling (in 4:1:1 and 4:2:2). They process image data in MCU size, 8x8 for 444; 16x8 for 422; 16x16 for 411.

Input Arguments

- pSrcMCU[3] – identifies a three-pointer array in sequence. The start address of each pointer in `pSrcMCU[]` must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access. The source data buffers identified by `pSrcMCU[]` supports top-down format. only.
  - `pSrcMCU[0]` points to Y block.
  - `pSrcMCU[1]` points to Cb block.
  - `pSrcMCU[2]` points to Cr block.
- `dstStep` – specifies the number of bytes in a line of the image data buffer. This buffer contains the current block, which is identified by `pDstBGR`. This value can be less than 0 to support down to top storage format.

Output Arguments

`pDstBGR` – identifies output image data buffer. The image data are stored in interleaved order as [x B G R x B G R …], where x represents a vacant bit, and B, G, R are expressed in 5 bits, respectively. Each output value in `pDstBGR` is saturated. The destination image data buffer `pDstBGR` can support down-top storage formats. In this case, `dstStep` can be less than 0.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - dstStep was an odd number or its absolute value was less than 16 (for 444) or 32 (for 422 and 411).
  - a pointer in `pDstMCU[]` was not 8-byte aligned.

NOTE. Each output value in pDstBGR is saturated.

NOTE. The start address of each pointer in pSrcMCU[] must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. The destination image data buffer pDstBGR can support down-top storage formats. In this case, dstStep can be less than 0. The source data buffers, which identified by pSrcMCU[] just support top-down format.

NOTE. If one or more of the following conditions is true, this function returns ippStsBadArgErr.

- Any pointer is NULL.
- dstStep is odd or its absolute value is less than 16 (for 444) or 32 (for 422 and 411).
- Any pointer in pSrcMCU[] is not 8-byte aligned.

YCbCr444ToBGR565LS_MCU_16s16u_P3C3R
YCbCr422ToBGR565LS_MCU_16s16u_P3C3R
YCbCr411ToBGR565LS_MCU_16s16u_P3C3R

Prototype

```
IppStatus ippiYCbCr444ToBGR565LS_MCU_16s16u_P3C3R (const Ipp16s
   *pSrcMCU[3],
   Ipp16u * pDstBGR, int dstStep);
IppStatus ippiYCbCr422ToBGR565LS_MCU_16s16u_P3C3R (const Ipp16s
   *pSrcMCU[3],
   Ipp16u * pDstBGR, int dstStep);
IppStatus ippiYCbCr411ToBGR565LS_MCU_16s16u_P3C3R (const Ipp16s
   *pSrcMCU[3],
   Ipp16u * pDstBGR, int dstStep);
```

Description

These functions convert YCbCr data to BGR565 data to with level-shift and sampling (in 4:1:1 and 4:2:2). They process image data in MCU size, 8x8 for 444; 16x8 for 422; 16x16 for 411.

Input Arguments

- `pSrcMCU[3]` – identifies a three-pointer array in sequence. The start address of each pointer in `pSrcMCU[]` must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access. The source data buffers identified by `pSrcMCU[]` supports top-down format. only.
  - — `pSrcMCU[0]` points to Y block
  - — `pSrcMCU[1]` points to Cb block
  - — `pSrcMCU[2]` points to Cr block
- `dstStep` – specifies the number of bytes in a line of the image data buffer. This buffer contains the current block, which is identified by `pDstBGR`. This value can be less than 0 to support down to top storage format.

Output Arguments

`pDstBGR` – identifies output image data buffer. The image data are stored in interleaved order as [B G R B G R …], where G is expressed by 6 bits, and B and R are expressed in 5 bits, respectively. Each output value in pDstBGR is saturated. The destination image data buffer `pDstBGR` can support down-top storage formats. In this case, `dstStep` can be less than 0.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - dstStep was an odd number or its absolute value was less than 16 (for 444) or 32 (for 422 and 411).
  - a pointer in `pSrcMCU[]` was not 8-byte aligned.

NOTE. Each output value in pDstBGR is saturated.

NOTE. The start address of each pointer in pSrcMCU[] must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. The destination image data buffer pDstBGR can support down-top storage formats. In this case, dstStep can be less than 0. The source data buffers, which identified by pSrcMCU[] just support top-down format.

NOTE. If one or more of the following conditions is true, this function returns ippStsBadArgErr.

- Any pointer is NULL.
- dstStep is odd or its absolute value is less than 16 (for 444) or 32 (for 422 and 411).
- Any pointer in pSrcMCU[] is not 8-byte aligned.

DCT and Quantization Function Set (DCTQ)

The following equations specify the ideal functional definition of forward DCT (FDCT) and inverse DCT (IDCT).

FDCT: $$S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

IDCT: $$s_{yx} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{vu} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

where

$C_u, C_v = 1/\sqrt{2}$ for $u, v = 0$

otherwise

$C_u, C_v = 1$

The following equations specify the ideal functional definition of Quantization and Dequantization operations.

Quantization:

$$Sq_{v,u} = round\_up\left(\frac{R_{v,u}}{Q_{v,u}}\right)$$

Dequantization:

$$R_{v,u} = Sq_{v,u} \times Q_{v,u}$$

Forward DCT

DCTQuantFwdTableInit_JPEG_8u16u

Prototype

```
IppStatus ippiDCTQuantFwdTableInit_JPEG_8u16u (const Ipp8u
   *pQuantRawTable, Ipp16u *pQuantFwdTable);
```

Description

Generate the quantization table for optimized FDCT.

Input Arguments

`pQuantRawTable` – identifies original quantization table. The table length is 64. The start address of `pQuantRawTable` must be 8-byte aligned.

Output Arguments

`pQuantFwdTable` – identifies destination quantization table. The table length is 64. The start address of `pQuantFwdTable` must be 8-byte aligned.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - the start address of a pointer was not 8-byte aligned.

NOTE. The start address of pQuantRawTable and pQuantFwdTable must be 8-byte aligned

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.
- The start address of each pointer must be 8-byte aligned.

DCTQuantFwd_JPEG_16s DCTQuantFwd_JPEG_16s_I

Prototype

```
IppStatus ippiDCTQuantFwd_JPEG_16s (const Ipp16s* pSrc,Ipp16s *pDst,
   const Ipp16u *pQuantFwdTable);
IppStatus ippiDCTQuantFwd_JPEG_16s_I (Ipp16s* pSrcDst, const Ipp16u
   *pQuantFwdTable);
```

Description

This function implements forward DCT with quantization for the 8-bit image data. It processes one block (8x8).

Input Arguments

- `pSrc` – identifies input coefficient block(8x8) buffer. This start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.
- `pQuantFwdTable` – identifies the quantization table which was generated from "DCTQuantFwdTableInit_JPEG_8u16u". The table length is 64. This start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

Output Arguments

`pDst` – identifies output coefficient block(8x8) buffer. This start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

To achieve better performance, the output 8x8 matrix is the transpose of the explicit result. This transpose will be handled in Huffman encoding.

In-Out Arguments

`pSrcDst` – identifies coefficient block(8x8) buffer for in-place processing. This start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL

— the start address of a pointer was not 8-byte aligned.

NOTE. The start address of pSrcDst, pSrc, pDst and pQuantFwdTable must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. In the IPP for PCA processors with MMX™, to achieve better performance, the output 8x8 matrix is the transpose of the explicit result. This transpose is handled in Huffman encoding.

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.
- The start address of each pointer must be 8-byte aligned.

Inverse DCT

DCTQuantInvTableInit_JPEG_8u16u

Prototype

```
IppStatus ippiDCTQuantInvTableInit_JPEG_8u16u (const Ipp8u
   *pQuantRawTable, Ipp16u *pQuantInvTable);
```

Description

Generate the dequantization table for optimized IDCT for 8-bit image data.

Input Arguments

`pQuantRawTable` – identifies original quantization table. The start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

Output Arguments

`pQuantInvTable` – identifies destination quantization table. The table length is 128. The start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - the start address of a pointer was not 8-byte aligned.

NOTE. The start address of pQuantRawTable and pQuantInvTable must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-cache access.

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.
- The start address of each pointer must be 8-byte aligned.

DCTQuantInv_JPEG_16s DCTQuantInv_JPEG_16s_I

Prototype

```
IppStatus ippiDCTQuantInv_JPEG_16s(const Ipp16s *pSrc, Ipp16s *pDst,
   const Ipp16u *pQuantInvTable);
IppStatus ippiDCTQuantInv_JPEG_16s_I (Ipp16s* pSrcDst,
   const Ipp16u *pQuantInvTable);
```

Description

This function implements inverse DCT with dequantization for 8-bit image data. It processes one block (8x8).

Input Arguments

- `pSrc` – identifies input coefficient block(8x8) buffer. The start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.
- `pQuantInvTable` – identifies the quantization table which was generated from "DCTQuantInvTableInit_JPEG_8u16u". The table length is 128. The start address must be 8-byte aligned.

Output Arguments

`pDst` – identifies output coefficient block(8x8) buffer. The start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

In-Out Arguments

`pSrcDst` – identifies coefficient block(8x8) buffer for in-place processing. This should be aligned on an 8-byte boundary. The start address must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL
  - the start address of a pointer was not 8-byte aligned.

NOTE. The start address of pSrc, pDst and pSrcDst must be 8-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.
- The start address of each pointer must be 8-byte aligned.

Huffman Coding Function Set (HUFFC)

NOTE. Please refer to the ISO/IEC 10918-1 for the details of Huffman coding. Sequential (baseline) Huffman coding is supported by IPP JPEG primitives.

EncodeHuffmanSpecGetBufSize_JPEG_8u

Prototype

```
IppStatus ippiEncodeHuffmanSpecGetBufSize_JPEG_8u (int* pSize);
```

Description

This function generates the size of the Huffman encoding table.

Input Arguments

None.

Output Arguments

- `pSize` - pointer to the size

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
  - a pointer was NULL

EncodeHuffmanStateGetBufSize_JPEG_8u

Prototype

```
IppStatus ippiEncodeHuffmanStateGetBufSize_JPEG_8u (int* pSize);
```

Description

This API generates the size of Huffman encoding state.

Input Arguments

None.

Output Arguments

- `pSize` - pointer to the size

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns `IPP_STATUS_BAD_ARG`:
  - pointer cannot be NULL.

EncodeHuffmanStateInit_JPEG_8u

Prototype

```
IppStatus ippiEncodeHuffmanStateInit_JPEG_8u (IppiEncodeHuffmanState*
   pHuffState);
```

Description

This function initializes the Huffman encoding state.

Input Arguments

None.

Output Arguments

- `pHuffState` - pointer to a IppiEncodeHuffmanState structure

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
  - a pointer was NULL

EncodeHuffmanSpecInit_JPEG_8u

Prototype

```
IppStatus ippiEncodeHuffmanSpecInit_JPEG_8u (const Ipp8u *pHuffBits,
   const Ipp8u *pHuffValue, IppiEncodeHuffmanSpec *pHuffTable);
```

Description

Generate Huffman table for encoding from Huffman table specification.

Input Arguments

- `pHuffBits` - Pointer to the array of HUFFBITS, which contains the number of Huffman codes for size 1-16.
- `pHuffValue` - Pointer to the array of HUFFVAL, which contains the symbol values to be associated with the Huffman codes ordering by size.

Output Arguments

`pHuffTable` – identifies a `IppiEncodeHuffmanSpec` data structure. The structure member is 4-byte aligned.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - a pointer was NULL

NOTE. This function generates DC (or AC) Huffman encoding table with Huffman table specification (pHuffBits/pHuffValue).

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.

EncodeHuffman8x8_Direct_JPEG_16s1u_C1

Prototype

```
IppStatus ippiEncodeHuffman8x8_Direct_JPEG_16s1u_C1 (const Ipp16s *pSrc, Ipp8u
    *pDst,int *pDstBitsLen, Ipp16s *pDCPred, const IppiEncodeHuffmanSpec
    *pDCHuffTable, const IppiEncodeHuffmanSpec *pACHuffTable);
```

NOTE. The structure was named IppJPEGEnHuffTable in previous versions of the API.

Description

This function implements Huffman encoding for baseline mode.

NOTE. In the IPP for PCA processors with MMX™, to match the use of forward DCTQ primitives, an implicit transpose of the source data block is integrated into Huffman encoding.

Input Arguments

- `pSrc` – identifies source data block (8x8). The start address of `pSrc` must be 4-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.
- `pDCHuffTable` – identifies `IppiEncodeHuffmanSpec` data structure, each of which indicates a DC Huffman encoding table. The structure member is 4-byte aligned.
- `pACHuffTable` – identifies `IppiEncodeHuffmanSpec` data structure. The structure member is 4-byte aligned.

Output Arguments

`pDst` – identifies destination bit stream buffer.

In-Out Arguments

- `pDstBitsLen`

— As an Input it identifies the length of valid bits in `pDst`.
— As an Output it identifies new length of valid bits in `pDst`.

- `pDCPred`
  — As an Input it identifies quantized DC coefficient from the most recently coded block of the component.
  — As an Output it identifies DC coefficient from the current block of the component.

NOTE. *pDCPred is set to 0 initially, or after each restart interval.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  — a pointer was NULL
  — the start address of a pointer was not 4-byte aligned.
  — `*pDstBitsLen` was less than 0.

NOTE. The start address of pSrc must be 4-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. *pDCPred is 0 in initial or after each restart interval.

NOTE. In the IPP for PCA processors with MMX™, to match the use of forward DCTQ primitives in the same library, an implicit transpose of the source data block is integrated into Huffman encoding. If the forward DCT primitives are not used in the same library, the user must transpose the DCT coefficients before calling the Huffman encoding function.

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.
- The start address of each pointer must be 4-byte aligned.
- *pDstBitsLen must be greater than or equal to 0.

EncodeHuffman8x8_DCFirst_JPEG_16s1u_C1

Prototype

```
IppStatus ippiEncodeHuffman8x8_DCFirst_JPEG_16s1u_C1 (
  const Ipp16s*                pSrc,
        Ipp8u*                 pDst,
        int                    dstBytesLen,
        int*                   pDstCurrPos,
        Ipp16s*                pDCPred,
        int                    al,
  const IppiEncodeHuffmanSpec* pDCHuffTable,
        IppiEncodeHuffmanState* pHuffState,
```

```
        int                     bFlushState);
```

Description

This function implements Huffman DC encoding of the first scan for progressive mode. The encoding procedure conforms to G.1.2 of T.81.

Input Arguments

- `pSrc` - pointer to the source coefficient data in the 8x8 block
- `dstBytesLen` - length of bit stream buffer
- `pDstCurrPos` - pointer to the current byte position in the bit stream buffer
- `pDCPred` - pointer to the DC value of the previous 8x8 block
- `al` - low bit position of successive approximation
- `pDCHuffTable` - pointer to the structure of Huffman encoding table for DC. The table must be retrieved by calling `ippiEncodeHuffmanSpecInit_JPEG_8u`
- `pHuffState` - pointer to the structure of Huffman encoding state
- `bFlushState` - set to 1 for the last 8x8 block to flush any pending bits in the state

Output Arguments

- `pDst` - pointer to encoded Huffman stream data
- `pDstCurrPos` - pointer to the updated byte position in the bit stream buffer
- `pDCPred` - pointer to the DC value for next prediction
- `pHuffState` - pointer to the updated structure of Huffman encoding state

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns `IPP_STATUS_BAD_ARG`:
  - pointer cannot be NULL
  - start address of `pSrc` must be 4-byte aligned
  - value pointed by `pDstCurrPos` must be greater than or equal to 0.
  - `al` should be larger than or equal to 0.

EncodeHuffman8x8_DCRefine_JPEG_16s1u_C1

Prototype

```
IppStatus ippiEncodeHuffman8x8_DCRefine_JPEG_16s1u_C1 (
  const Ipp16s*                 pSrc,
        Ipp8u*                  pDst,
        int                     dstBytesLen,
        int*                    pDstCurrPos,
        int                     al,
        IppiEncodeHuffmanState* pHuffState,
        int                     bFlushState);
```

Description

This function implements Huffman DC encoding of the refined scans for progressive mode. The encoding procedure conforms to G.1.2 of T.81.

Input Arguments

- `pSrc` - pointer to the source coefficient data in the 8x8 block
- `dstBytesLen` - length of bit stream buffer
- `pDstCurrPos` - pointer to the current byte position in the bit stream buffer
- `al` - low bit position of successive approximation
- `pHuffState` - pointer to the structure of Huffman encoding state
- `bFlushState` - set to 1 for the last 8x8 block to flush any pending bits in the state

Output Arguments

- `pDst` - pointer to encoded huffman stream data
- `pDstCurrPos` - pointer to the updated byte position in the bit stream buffer
- `pHuffState` - pointer to the updated structure of Huffman encoding state

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)

- In debug version, if following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG:
  - pointer cannot be NULL
  - start address of pSrc must be 4-byte aligned
  - value pointed by pDstCurrPos must be greater than or equal to 0
  - al should be larger than or equal to 0

EncodeHuffman8x8_ACFirst_JPEG_16s1u_C1

Prototype

```
IppStatus ippiEncodeHuffman8x8_ACFirst_JPEG_16s1u_C1 (
  const Ipp16s*                pSrc,
        Ipp8u*                 pDst,
        int                    dstBytesLen,
        int*                   pDstCurrPos,
        int                    ss,
        int                    se,
        int                    al,
  const IppiEncodeHuffmanSpec* pACHuffTable,
        IppiEncodeHuffmanState* pHuffState,
        int                    bFlushState);
```

Description

This function implements Huffman AC encoding of the first scans for progressive mode. The encoding procedure conforms to G.1.2 of T.81.

Input Arguments

- pSrc - pointer to the source coefficient data in the 8x8 block
- dstBytesLen - length of bit stream buffer
- pDstCurrPos - pointer to the current byte position in the bit stream buffer
- ss - start index of spectral selection
- se - end index of spectral selection
- al - low bit position of successive approximation

- pACHuffTable - pointer to the structure of Huffman encoding table for AC. The table must be retrieved by calling ippiEncodeHuffmanSpecInit_JPEG_8u.
- pHuffState - pointer to the structure of Huffman encoding state
- bFlushState - set to 1 for the last 8x8 block to flush any pending bits in the state

Output Arguments

- pDst - pointer to encoded huffman stream data
- pDstCurrPos - pointer to the updated byte position in the bit stream buffer
- pHuffState - pointer to the updated structure of Huffman encoding state

Returns

- IPP_STATUS_OK - no error
- IPP_STATUS_BAD_ARG - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG:
  - pointer cannot be NULL
  - start address of pSrc must be 4-byte aligned
  - value pointed by pDstCurrPos must be greater than or equal to 0
  - ss shall be larger than 0.
  - se shall be larger than or equal to ss and less than 64
  - al should be larger than or equal to 0

EncodeHuffman8x8_ACRefine_JPEG_16s1u_C1

Prototype

```
IppStatus ippiEncodeHuffman8x8_ACRefine_JPEG_16s1u_C1 (
  const Ipp16s*                pSrc,
        Ipp8u*                 pDst,
        int                    dstBytesLen,
        int*                   pDstCurrPos,
        int                    ss,
        int                    se,
```

```
        int                        al,
  const IppiEncodeHuffmanSpec*   pACHuffTable,
        IppiEncodeHuffmanState *pHuffState,
        int                        bFlushState);
```

Description

This function implements Huffman AC encoding of the refined scans for progressive mode. The encoding procedure conforms to G.1.2 of T.81.

Input Arguments

- `pSrc` - pointer to the source coefficient data in the 8x8 block
- `dstBytesLen` - length of bit stream buffer
- `pDstCurrPos` - pointer to the current byte position in the bit stream buffer
- `ss` - start index of spectral selection
- `se` - end index of spectral selection
- `al` - low bit position of successive approximation
- `pACHuffTable` - pointer to the structure of Huffman encoding table for AC. The table must be retrieved by calling `ippiEncodeHuffmanSpecInit_JPEG_8u`.
- `pHuffState` - pointer to the structure of Huffman encoding state
- `bFlushState` - set to 1 for the last 8x8 block to flush any pending bits in the state

Output Arguments

- `pDst` - pointer to encoded huffman stream data
- `pDstCurrPos` - pointer to the updated byte position in the bit stream buffer
- `pHuffState` - pointer to the updated structure of Huffman encoding state

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns `IPP_STATUS_BAD_ARG`:
  - pointer cannot be NULL
  - start address of `pSrc` must be 4-byte aligned
  - value pointed by `pDstCurrPos` must be greater than or equal to 0
  - `ss` shall be larger than 0
  - `se` shall be larger than or equal to `ss` and less than 64
  - `al` should be larger than or equal to 0

DecodeHuffmanSpecGetBufSize_JPEG_8u

Prototype

```
IppStatus ippiDecodeHuffmanSpecGetBufSize_JPEG_8u (int* pSize);
```

Description

This function generates the size of Huffman decoding table.

Input Arguments

None.

Output Arguments

- `pSize` - pointer to the size

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns `IPP_STATUS_BAD_ARG`:
  - pointer cannot be NULL

DecodeHuffmanStateGetBufSize_JPEG_8u

Prototype

```
IppStatus ippiDecodeHuffmanStateGetBufSize_JPEG_8u (int* pSize);
```

Description

This function generates the size of Huffman decoding state.

Input Arguments

None.

Output Arguments

- `pSize` - pointer to the size

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns `IPP_STATUS_BAD_ARG`:
  - pointer cannot be NULL.

DecodeHuffmanStateInit_JPEG_8u

Prototype

```
IppStatus ippiDecodeHuffmanStateInit_JPEG_8u (IppiDecodeHuffmanState*
   pHuffState);
```

Description

This function initializes the Huffman decoding state.

Input Arguments

None.

Output Arguments

- `pHuffState` - pointer to a `IppiDecodeHuffmanState` structure

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns `IPP_STATUS_BAD_ARG`:

— pointer cannot be NULL

DecodeHuffmanSpecInit_JPEG_8u

Prototype

```
IppStatus ippiDecodeHuffmanSpecInit_JPEG_8u (const Ipp8u *pHuffBits,
   const Ipp8u *pHuffValue, IppiDecodeHuffmanSpec *pHuffTable);
```

Description

This function generates Huffman table for decoding from Huffman table specification.

NOTE. This function generates DC (or AC) Huffman encoding table with Huffman table specification (pHuffBits/pHuffValue).

Input Arguments

- `pHuffBits` – Pointer to the array of HUFFBITS, which contains the number of Huffman codes for size 1-16
- `pHuffValue` – Pointer to the array of HUFFVAL, which contains the symbol values to be associated with the Huffman codes ordering by size

Output Arguments

`pHuffTable` – identifies a `IppiDecodeHuffmanSpec` data structure. The structure member is 4-byte aligned.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - — a pointer was NULL

NOTE. This function generates DC (or AC) Huffman encoding table with Huffman table specification (pHuffBits/pHuffValue).

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.

DecodeHuffman8x8_Direct_JPEG_1u16s_C1

Prototype

```
IppStatus ippiDecodeHuffman8x8_Direct_JPEG_1u16s_C1 (const Ipp8u *pSrc,
   int *pSrcBitsLen, Ipp16s *pDst, Ipp16s *pDCPred, int   *pMarker,
   Ipp32u *pPrefetchedBits, int *pNumValidPrefetchedBits, const
   IppiDecodeHuffmanSpec *pDCHuffTable, const IppiDecodeHuffmanSpec
   *pACHuffTable);
```

Description

This function implements Huffman decoding for baseline mode.

Input Arguments

- `pSrc` – identifies source bit stream buffer
- `pDCHuffTable` – identifies `IppiDecodeHuffmanSpec` structure, each of which indicates a DC Huffman decoding table. This should be aligned on an 4-byte boundary.
- `pACHuffTable` – identifies `IppiDecodeHuffmanSpec` structure, each of which indicates an AC Huffman decoding table. This is 4-byte aligned.

Output Arguments

- `pDst` – identifies source data block. The start addresses of `pDst` must be 4-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

- `pMarker` – identified next marker decoded by the function. `pMarker` is set to 0 initially or after it is processed.

In-Out Arguments

- `pDCPred`
  - — As an Input it identifies quantized DC coefficient from the most recently decoded block of the component. `pDCPred` is set to 0 initially or after each restart interval.
  - — As an Output it identifies DC coefficient from the current block of the component.
- `pSrcBitsLen` – identifies the number of bits that have been decoded
- `pPrefetchedBits` – identifies pre-fetch bits buffer. It is initially set to 0.
- `pNumValidPrefetchedBits` – identifies the number of valid bits in the pre-fetch buffer. This is 0 in following conditions:
  - — initially
  - — after each restart interval
  - — after each marker.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments. Returned for any of the following conditions:
  - — a pointer was NULL
  - — `*pSrcBitsLen` was less than 0.
  - — `*pNumValidPrefetchedBits` was less than 0.
  - — the start address of `pDst` was not 4-byte aligned.

NOTE. The start addresses of pDst must be 4-byte aligned. It is better to align at a 32-byte boundary to optimize D-Cache access.

NOTE. *pDCPred is 0 in initial or after each restart interval.

NOTE. *pMarker is 0 in initial or after it was processed.

NOTE. *pNumValidPrefetchedBits is 0 in following case:

- Initial
- After each restart interval
- After each marker

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.
- *pSrcBitsLen is greater than or equal to 0.
- *pNumValidPrefetchedBits is greater than or equal to 0.
- The start address of pDst must be 4-byte aligned.

DecodeHuffman8x8_DCFirst_JPEG_1u16s_C1

Prototype

```
IppStatus ippiDecodeHuffman8x8_DCFirst_JPEG_1u16s_C1 (
  const Ipp8u*                  pSrc,
        int                     srcBytesLen,
```

```
        int*                    pSrcCurrPos,
        Ipp16s*                 pDst,
        Ipp16s*                 pDCPred,
        int*                    pMarker,
        int                     al,
  const IppiDecodeHuffmanSpec*  pDCHuffTable,
        IppiDecodeHuffmanState* pHuffState);
```

Description

This function implements Huffman DC decoding of the first scan for progressive mode. The decoding procedure conforms to G.2 of T.81.

Input Arguments

- `pSrc` - pointer to the input bit stream buffer
- `srcBytesLen` - length of bit stream
- `pSrcCurrPos` - pointer to the current byte position in the bit stream buffer
- `pDCPred` - pointer to the DC value of the previous 8x8 block
- `pMarker` - pointer to the marker
- `al` - low bit position of successive approximation
- `pDCHuffTable` - pointer to the structure of Huffman decoding table for DC. The table must be retrieved by calling `ippiDecodeHuffmanSpecInit_JPEG_8u`.
- `pHuffState` - pointer to the structure of Huffman decoding state

Output Arguments

- `pSrcCurrPos` - pointer to the updated byte position in the bit stream buffer
- `pDst` - pointer to the output block of decoded coefficient data
- `pDCPred` - pointer to the DC value for next prediction
- `pMarker` - pointer to the marker
- `pHuffState` - pointer to the updated structure of Huffman decoding state

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns `IPP_STATUS_BAD_ARG`:
  - pointer cannot be NULL
  - start address of `pDst` must be 4-byte aligned

— value pointed by `pSrcCurrPos` must be greater than or equal to 0
— `al` should be larger than or equal to 0

DecodeHuffman8x8_DCRefine_JPEG_1u16s_C1

Prototype

```
IppStatus ippiDecodeHuffman8x8_DCRefine_JPEG_1u16s_C1 (
  const Ipp8u*                  pSrc,
        int                     srcBytesLen,
        int*                    pSrcCurrPos,
        Ipp16s*                 pDst,
        int*                    pMarker,
        int                     al,
        IppiDecodeHuffmanState* pHuffState);
```

Description

This function implements Huffman DC decoding of the refined scans for progressive mode. The decoding procedure conforms to G.2 of T.81.

Input Arguments

- `pSrc` - pointer to the input bit stream buffer
- `srcBytesLen` - length of bit stream
- `pSrcCurrPos` - pointer to the current byte position in the bit stream buffer
- `pMarker` - pointer to the marker
- `al` - low bit position of successive approximation
- - pointer to the structure of Huffman decoding state

Output Arguments

- `pHuffState` - pointer to the updated byte position in the bit stream buffer
- `pDst` - pointer to the output block of decoded coefficient data
- `pMarker` - pointer to the marker
- `pHuffState` - pointer to the updated structure of Huffman decoding state

Returns

- IPP_STATUS_OK - no error
- IPP_STATUS_BAD_ARG - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG.
  - pointer cannot be NULL
  - start address of pDst must be 4-byte aligned
  - value pointed by pSrcCurrPos must be greater than or equal to 0
  - al should be larger than or equal to 0

DecodeHuffman8x8_ACFirst_JPEG_1u16s_C1

Prototype

```
IppStatus ippiDecodeHuffman8x8_ACFirst_JPEG_1u16s_C1 (
  const Ipp8u*                  pSrc,
        int                     srcBytesLen,
        int*                    pSrcCurrPos,
        Ipp16s*                 pDst,
        int*                    pMarker,
        int                     ss,
        int                     se,
        int                     al,
  const IppiDecodeHuffmanSpec*  pACHuffTable,
        IppiDecodeHuffmanState* pHuffState);
```

Description

This function implements Huffman AC decoding of the first scans for progressive mode. The decoding procedure conforms to G.2 of T.81.

Input Arguments

- pSrc - pointer to the input bit stream buffer
- srcBytesLen - length of bit stream

- pSrcCurrPos - pointer to the current byte position in the bit stream buffer
- pMarker - pointer to the marker
- ss - start index of spectral selection
- se - end index of spectral selection
- al -low bit position of successive approximation
- pACHuffTable - pointer to the structure of Huffman decoding table for AC. The table must be gotten by calling ippiDecodeHuffmanSpecInit_JPEG_8u.
- pHuffState - pointer to the structure of Huffman decoding state

Output Arguments

- pSrcCurrPo - pointer to the updated byte position in the bit stream buffer
- pDst - pointer to the output block of decoded coefficient data
- pMarker - pointer to the marker
- pHuffState - pointer to the updated structure of Huffman decoding state

Returns

- IPP_STATUS_OK - no error
- IPP_STATUS_BAD_ARG - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns IPP_STATUS_BAD_ARG:
  - pointer cannot be NULL
  - start address of pDst must be 4-byte aligned
  - value pointed by pSrcCurrPos must be greater than or equal to 0
  - ss shall be larger than 0
  - se shall be larger than or equal to ss and less than 64
  - al should be larger than or equal to 0

DecodeHuffman8x8_ACRefine_JPEG_1u16s_C1

Prototype

```
IppStatus ippiDecodeHuffman8x8_ACRefine_JPEG_1u16s_C1 (
  const Ipp8u*                  pSrc,
```

```
        int                      srcBytesLen,
        int*                     pSrcCurrPos,
        Ipp16s*                  pDst,
        int*                     pMarker,
        int                      ss,
        int                      se,
        int                      al,
  const IppiDecodeHuffmanSpec*   pACHuffTable,
        IppiDecodeHuffmanState*  pHuffState);
```

Description

This function implements Huffman AC decoding of the first scans for progressive mode. The decoding procedure conforms to G.2 of T.81.

Input Arguments

- `pSrc` - pointer to the input bit stream buffer
- `srcBytesLen` - length of bit stream
- `pSrcCurrPos` - pointer to the current byte position in the bit stream buffer
- `pMarker` - pointer to the marker
- `ss` - start index of spectral selection
- `se` - end index of spectral selection
- `al` - low bit position of successive approximation
- `pACHuffTable` - pointer to the structure of Huffman decoding table for AC. The table must be gotten by calling `ippiDecodeHuffmanSpecInit_JPEG_8u`.
- `pHuffState` - pointer to the structure of Huffman decoding state

Output Arguments

- `pSrcCurrPos` - pointer to the updated byte position in the bit stream buffer
- `pDst` - pointer to the output block of decoded coefficient data
- `pMarker` - pointer to the marker
- `pHuffState` - pointer to the updated structure of Huffman decoding state

Returns

- `IPP_STATUS_OK` - no error
- `IPP_STATUS_BAD_ARG` - bad arguments (invalid in release version)
- In debug version, if following conditions are not satisfied, this function returns `IPP_STATUS_BAD_ARG`:

— pointer cannot be NULL
— start address of `pDst` must be 4-byte aligned
— value pointed by `pSrcCurrPos` must be greater than or equal to 0
— `ss` shall be larger than 0
— `se` shall be larger than or equal to `ss` and less than 64
— `al` should be larger than or equal to 0

JPEG 2000 Image CODEC

18

Description

This chapter describes the Intel® Integrated Performance Primitives (Intel® IPP) on Intel® PCA Processors with Intel® Wireless MMX™ Technology (PCA processors with MMX™). The product is designed to support image CODECs that are compliant with ISO/IEC 15444-1 or so called JPEG 2000 CODEC. In this version the following function set groups is provided:

- Discrete Wavelet Transformation (DWT)

Below is an overview of DWT required by JPEG 2000. Please refer to ISO/IEC 15444-1 for the details.

High Level Description

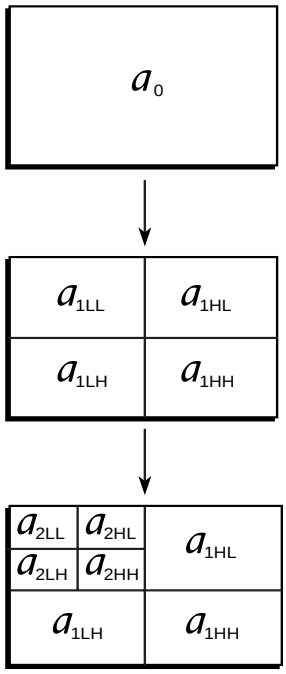
Discrete Wavelet Transformation (DWT)

For both of forward discrete wavelet transformation (FDWT) and inverse discrete wavelet transformation (IDWT), JPEG 2000 standard specifies two kinds of transformations: one uses a 5-3 reversible filter and the other applies a 9-7 irreversible filter.

Forward Discrete Wavelet Transformation (FDWT)

The forward discrete wavelet transformation decomposes DC-level shifted tile-component samples into a set of subbands. Figure 18-1 shows the case of 2-level decomposition.

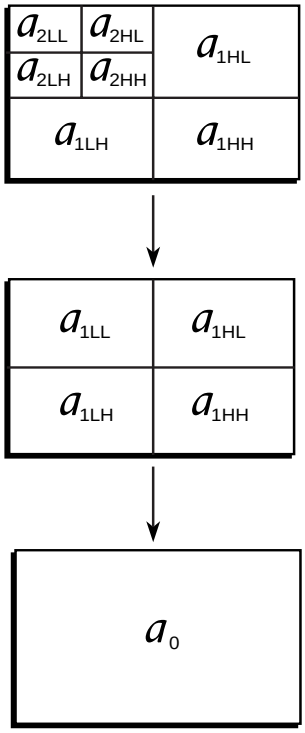
Figure 18-1 2-level FDWT



Inverse Discrete Wavelet Transformation (IDWT)

The inverse discrete wavelet transformation reconstructs a set of subbands into DC-level shifted tile-component samples. Figure 18-2 shows the case of 2-level reconstruction.

Figure 18-2 2-level IDWT



Function Sets

Table 18-1 summarizes the function sets and primitives for the JPEG 2000 CODEC:

Table 18-1 Function Sets and Descriptions of JPEG 2000 Primitives

Function Set groups	Functions Sets	APIs	Description
DWT	5-3 wavelet transformation	ippiWTGetBufSize_B53_JPEG2K_16s_C1IR	Getting the size (in bytes) of working buffer for 16-bit 5-3 DWT
		ippiWTGetBufSize_B53_JPEG2K_32s_C1IR	Getting the size (in bytes) of working buffer for 32-bit 5-3 DWT
		ippiWTFwd_B53_JPEG2K_16s_C1IR	16-bit forward 5-3 DWT
		ippiWTFwd_B53_JPEG2K_32s_C1IR	32-bit forward 5-3 DWT
		IppiWTInv_B53_JPEG2K_16s_C1IR	16-bit inverse 5-3 DWT
		ippiWTInv_B53_JPEG2K_32s_C1IR	32-bit inverse 5-3 DWT
	9-7 wavelet transformation	ippiWTGetBufSize_D97_JPEG2K_16s_C1IR	Getting the size (in bytes) of working buffer for 16-bit 9-7 DWT
		ippiWTGetBufSize_D97_JPEG2K_32s_C1IR	Getting the size (in bytes) of working buffer for 32-bit 9-7 DWT
		ippiWTFwd_D97_JPEG2K_16s_C1IR	16-bit forward 9-7 DWT
		ippiWTFwd_D97_JPEG2K_32s_C1IR	32-bit forward 9-7 DWT
		ippiWTInv_D97_JPEG2K_16s_C1IR	16-bit inverse 9-7 DWT
		ippiWTInv_D97_JPEG2K_32s_C1IR	32-bit inverse 9-7 DWT

DWT Group

5-3 DWT Function Set

WTGetBufSize_B53_JPEG2K_16s_C1IR

Prototype

```
IppStatus ippiWTGetBufSize_B53_JPEG2K_16s_C1IR(const IppiRect *pTileRect,
    int *pSize);
```

Description

Get the buffer size (in bytes) for both one-level 2D forward and inverse wavelet transformation on an image tile (of 16-bit data) using 5-3 reversible filter.

Input Arguments

`pTileRect` – pointer to an IppiRect data structure, which indicates the position and size of the image tile

Output Arguments

`pSize` – pointer to an integer of size of the internal buffer used by 2D 5-3 wavelet transformation

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

WTGetBufSize_B53_JPEG2K_32s_C1IR

Prototype

```
IppStatus ippiWTGetBufSize_B53_JPEG2K_32s_C1IR(const IppiRect *pTileRect,
    int *pSize);
```

Description

Get the buffer size (in bytes) for both one-level 2D forward and inverse wavelet transformation on an image tile (of 32-bit data) using 5-3 reversible filter.

Input Arguments

`pTileRect` – pointer to an IppiRect data structure, which indicates the position and size of the image tile

Output Arguments

`pSize` – pointer to an integer of size of the internal buffer used by 2D 5-3 wavelet transformation

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

WTFwd_B53_JPEG2K_16s_C1IR

Prototype

```
IppStatus ippiWTFwd_B53_JPEG2K_16s_C1IR(Ipp16s *pSrcDstTile,
    int step, const IppiRect *pTileRect,Ipp8u *pBuffer);
```

Description

This function makes a one-level forward 2D wavelet transformation on one image tile using the 5-3 reversible filter. The DWT coefficients are de-interleaved into LL, HL, LH and HH subbands and written back to the buffer of input data. The image tile data are in 16-bit data type.

NOTE. *The input image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The DWT results are de-interleaved into 4 subbands, which are written to the buffer pointed by pSrcDstTile as shown in* Figure 18-3.

Input Arguments

- `pSrcDstTile` – pointer to the buffer of input image tile. The start address of `pSrcDstTile` must be 8-byte aligned. It is better to make it 32-byte aligned.
- `step` – specifies the number of bytes in a line of the input data buffer. `step` is in bytes, and must be an integer multiple of 8.
- `pTileRect` – pointer to an IppiRect data structure, which indicates the position and size of the image tile
- `pBuffer` – pointer to the work buffer for transform.The start address of `pBuffer` must be 8-byte aligned. It is better to make it 32-byte aligned.

  The work buffer pointed by `pBuffer` must be allocated before the function is called, and its size can be gotten by calling "`WTGetBufSize_B53_JPEG2K_16s_C1IR`".

Output Arguments

`pSrcDstTile` – pointer to the buffer of output DWT coefficients

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

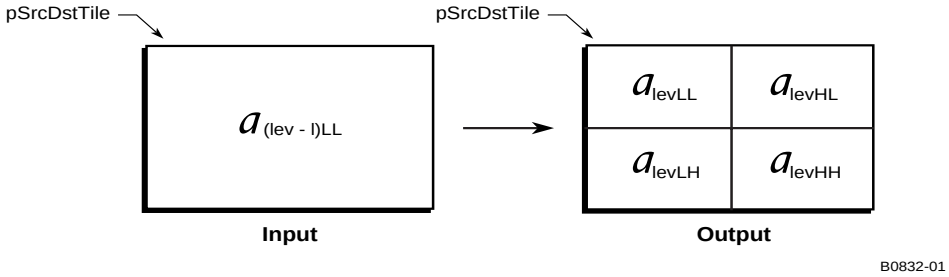
NOTE. The work buffer pointed by pBuffer must be allocated before the function is called, and its size can be gotten by calling ippiWTGetBufSize_B53_JPEG2K_16s_C1IR.

NOTE. The start address of pSrcDstTile and pBuffer must be 8-byte aligned. It is better to make them 32-byte aligned.

NOTE. `step` is in bytes, and must be an integer multiple of 8.

NOTE. The input image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The DWT results are de-interleaved into 4 subbands, which are written to the buffer pointed by pSrcDstTile as shown in Figure 18-3.

Figure 18-3 Input and Output Data in Buffer Pointed by pSrcDstTile (Forward DWT)



WTFwd_B53_JPEG2K_32s_C1IR

Prototype

```
IppStatus ippiWTFwd_B53_JPEG2K_32s_C1IR(Ipp32s *pSrcDstTile, int step,
   const IppiRect *pTileRect,Ipp8u *pBuffer);
```

Description

This function makes a one-level forward 2D wavelet transformation on one image tile using the 5-3 reversible filter. The DWT coefficients are de-interleaved into LL, HL, LH and HH subbands and written back to the buffer of input data. The image tile data are in 32-bit data type.

NOTE. *The input image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The DWT results are de-interleaved into 4 subbands, which are written to the buffer pointed by pSrcDstTile as shown in* Figure 18-3.

Input Arguments

- `pSrcDstTile` – pointer to the buffer of input image tile. The start address of `pSrcDstTile` must be 8-byte aligned. It is better to make it 32-byte aligned.
- `step` – specifies the number of bytes in a line of the input data buffer. `step` is in bytes, and must be an integer multiple of 8.
- `pTileRect` – pointer to an `IppiRect` data structure, which indicates the position and size of the image tile
- `pBuffer` – pointer to the work buffer for transform. The start address of `pBuffer` must be 8-byte aligned. It is better to make it 32-byte aligned.

  The work buffer pointed by `pBuffer` must be allocated before the function is called, and its size can be gotten by calling "`WTGetBufSize_B53_JPEG2K_32s_C1IR`".

Output Arguments

`pSrcDstTile` – pointer to the buffer of output DWT coefficients.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

NOTE. The work buffer pointed by pBuffer must be allocated before the function is called, and its size can be gotten by calling ippiWTGetBufSize_B53_JPEG2K_32s_C1IR.

NOTE. The start address of pSrcDstTile and pBuffer must be 8-byte aligned. It is better to make them 32-byte aligned.

NOTE. `step` is in bytes, and must be an integer multiple of 8.

NOTE. The input image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The DWT results are de-interleaved into 4 subbands, which are written to the buffer pointed by pSrcDstTile as shown in Figure 18-3.

WTInv_B53_JPEG2K_16s_C1IR

Prototype

```
IppStatus ippiWTInv_B53_JPEG2K_16s_C1IR(Ipp16s *pSrcDstTile,
    int step, const IppiRect *pTileRect,Ipp8u *pBuffer);
```

Description

This function interleaves the LL, HL, LH and HH subbands of DWT coefficients and then makes a one-level inverse 2D wavelet transformation on them using the 5-3 reversible filter. The results are written back to the buffer of input data. The image tile data are in 16-bit data type.

NOTE. *The output image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The input DWT coefficients from the buffer pointed by pSrcDstTile are de-interleaved into 4 subbands as shown in* Figure 18-4.

Input Arguments

- `pSrcDstTile` – pointer to the buffer of input LL, HL, LH and HH subbands of DWT coefficients. The start address of `pSrcDstTile` must be 8-byte aligned. It is better to make it 32-byte aligned.
- `step` – specifies the number of bytes in a line of the input data buffer. `step` is in bytes, and must be an integer multiple of 8.
- `pTileRect` – pointer to an IppiRect data structure which indicates the position and size of the image tile.
- `pBuffer` – pointer to the work buffer for transform. The start address of `pBuffer` must be 8-byte aligned. It is better to make it 32-byte aligned.

  The work buffer pointed by `pBuffer` must be allocated before the function is called, and its size can be gotten by calling "`WTGetBufSize_B53_JPEG2K_16s_C1IR`"

Output Arguments

`pSrcDstTile` – pointer to the buffer of output image tile

Returns

- `ippStsNoErr` – No error
- `ippStsBadArgErr` – Bad arguments

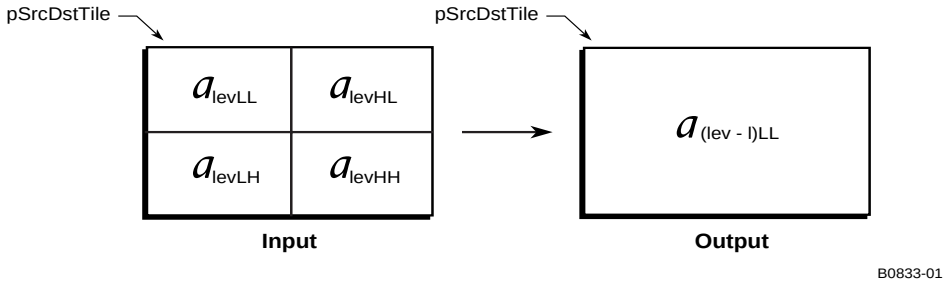
NOTE. The work buffer pointed by pBuffer must be allocated before the function is called, and its size can be gotten by calling ippiWTGetBufSize_B53_JPEG2K_16s_C1IR.

NOTE. The start address of pSrcDstTile and pBuffer must be 8-byte aligned. It is better to make them 32-byte aligned.

NOTE. `step` is in bytes, and must be an integer multiple of 8.

NOTE. The output image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The input DWT coefficients from the buffer pointed by pSrcDstTile are de-interleaved into 4 subbands as shown in Figure 18-4.

Figure 18-4 Input and Output Data in Buffer Pointed by pSrcDstTile (Inverse DWT)



WTInv_B53_JPEG2K_32s_C1IR

Prototype

```
IppStatus ippiWTInv_B53_JPEG2K_32s_C1IR(Ipp32s *pSrcDstTile, int step,
    const IppiRect *pTileRect, Ipp8u *pBuffer);
```

Description

This function interleaves the LL, HL, LH and HH subbands of DWT coefficients and then makes a one-level inverse 2D wavelet transformation on them using the 5-3 reversible filter. The results are written back to the buffer of input data. The image tile data are in 32-bit data type.

NOTE. *The output image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The input DWT coefficients from the buffer pointed by pSrcDstTile are de-interleaved into 4 subbands as shown in* Figure 18-4.

Input Arguments

- `pSrcDstTile` – pointer to the buffer of input LL, HL, LH and HH subbands of DWT coefficients. The start address of `pSrcDstTile` must be 8-byte aligned. It is better to make it 32-byte aligned.
- `step` – specifies the number of bytes in a line of the input data buffer. `step` is in bytes, and must be an integer multiple of 8.
- `pTileRect` – pointer to an IppiRect data structure which indicates the position and size of the image tile
- `pBuffer` – pointer to the work buffer for transform. The start address of `pBuffer` must be 8-byte aligned. It is better to make it 32-byte aligned.

  The work buffer pointed by `pBuffer` must be allocated before the function is called, and its size can be gotten by calling "WTGetBufSize_B53_JPEG2K_32s_C1IR".

Output Arguments

`pSrcDstTile` – pointer to the buffer of output image tile

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

NOTE. The work buffer pointed by pBuffer must be allocated before the function is called, and its size can be gotten by calling ippiWTGetBufSize_B53_JPEG2K_32s_C1IR.

NOTE. The start address of pSrcDstTile and pBuffer must be 8-byte aligned. It is better to make them 32-byte aligned.

NOTE. `step` is in bytes, and must be an integer multiple of 8.

NOTE. The output image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The input DWT coefficients from the buffer pointed by pSrcDstTile are de-interleaved into 4 subbands as shown in Figure 18-4.

9-7 DWT Function Set

WTGetBufSize_D97_JPEG2K_16s_C1IR

Prototype

```
IppStatus ippiWTGetBufSize_D97_JPEG2K_16s_C1IR(const IppiRect *pTileRect,
    int *pSize);
```

Description

Get the buffer size (in bytes) for both one-level 2D forward and inverse wavelet transformation on an image tile (of 16-bit data) using 9-7 reversible filter.

Input Arguments

`pTileRect` – pointer to an `IppiRect` data structure, which indicates the position and size of the image tile

Output Arguments

`pSize` – pointer to an integer of size of the internal buffer used by 2D 9-7 wavelet transformation.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

WTGetBufSize_D97_JPEG2K_32s_C1IR

Prototype

```
IppStatus ippiWTGetBufSize_D97_JPEG2K_32s_C1IR(const IppiRect *pTileRect,
    int *pSize);
```

Description

Get the buffer size (in bytes) for both one-level 2D forward and inverse wavelet transformation on an image tile (of 32-bit data) using 9-7 reversible filter.

Input Arguments

`pTileRect` – pointer to an IppiRect data structure, which indicates the position and size of the image tile

Output Arguments

`pSize` – pointer to an integer of size of the internal buffer used by 2D 9-7 wavelet transformation

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

WTFwd_D97_JPEG2K_16s_C1IR

Prototype

```
IppStatus ippiWTFwd_D97_JPEG2K_16s_C1IR (Ipp16s *pSrcDstTile, int step,
    const IppiRect *pTileRect, Ipp8u *pBuffer);
```

Description

This function makes a fix-point implementation of one-level forward 2D wavelet transformation on one image tile using the 9-7 irreversible filter. The DWT coefficients are de-interleaved into LL, HL, LH and HH subbands and written back to the buffer of input data. The image tile data are in 16-bit data type.

NOTE. *The input image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The DWT results are de-interleaved into 4 subbands, which are written to the buffer pointed by pSrcDstTile as shown in* Figure 18-3.

Input Arguments

- `pSrcDstTile` – pointer to the buffer of input image tile. The start address of `pSrcDstTile` must be 8-byte aligned. It is better to make it 32-byte aligned.
- `step` – specifies the number of bytes in a line of the input data buffer. `step` is in bytes, and must be an integer multiple of 8.
- `pTileRect` – pointer to an IppiRect data structure, which indicates the position and size of the image tile

- `pBuffer` – pointer to the work buffer for transform. The start address of `pBuffer` must be 8-byte aligned. It is better to make it 32-byte aligned.

  The work buffer pointed by `pBuffer` must be allocated before the function is called. Its size can be gotten by calling "WTGetBufSize_D97_JPEG2K_16s_C1IR"

Output Arguments

`pSrcDstTile` – pointer to the buffer of output DWT coefficients

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

NOTE. The work buffer pointed by pBuffer must be allocated before the function is called. Its size can be gotten by calling ippiWTGetBufSize_D97_JPEG2K_16s_C1IR.

NOTE. The start address of pSrcDstTile and pBuffer must be 8-byte aligned. It is better to make them 32-byte aligned.

NOTE. `step` is in bytes, and must be an integer multiple of 8.

NOTE. The input image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The DWT results are de-interleaved into 4 subbands, which are written to the buffer pointed by pSrcDstTile as shown in Figure 18-3.

WTFwd_D97_JPEG2K_32s_C1IR

Prototype

```
IppStatus ippiWTFwd_D97_JPEG2K_32s_C1IR (Ipp32s *pSrcDstTile,
   int step, const IppiRect *pTileRect, Ipp8u *pBuffer);
```

Description

This function makes a fix-point implementation of one-level forward 2D wavelet transformation on one image tile using the 9-7 irreversible filter. The DWT coefficients are de-interleaved into LL, HL, LH and HH subbands and written back to the buffer of input data. The image tile data are in 32-bit data type.

NOTE. *The input image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The DWT results are de-interleaved into 4 subbands, which are written to the buffer pointed by pSrcDstTile as shown in* Figure 18-3.

Input Arguments

- `pSrcDstTile` – pointer to the buffer of input image tile. The start address of `pSrcDstTile` must be 8-byte aligned. It is better to make it 32-byte aligned.
- `step` – specifies the number of bytes in a line of the input data buffer. `step` is in bytes, and must be an integer multiple of 8.
- `pTileRect` – pointer to an `IppiRect` data structure, which indicates the position and size of the image tile
- `pBuffer` – pointer to the work buffer for transform. The start address of `pBuffer` must be 8-byte aligned. It is better to make it 32-byte aligned.

  The work buffer pointed by `pBuffer` must be allocated before the function is called. Its size can be gotten by calling "`WTGetBufSize_D97_JPEG2K_32s_C1IR`"

Output Arguments

`pSrcDstTile` – pointer to the buffer of output DWT coefficients.

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

NOTE. The work buffer pointed by pBuffer must be allocated before the function is called. Its size can be gotten by calling ippiWTGetBufSize_D97_JPEG2K_32s_C1IR.

NOTE. The start address of pSrcDstTile and pBuffer must be 8-byte aligned. It is better to make them 32-byte aligned.

NOTE. `step` is in bytes, and must be an integer multiple of 8.

NOTE. The input image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The DWT results are de-interleaved into 4 subbands, which are written to the buffer pointed by pSrcDstTile as shown in Figure 18-3.

WTInv_D97_JPEG2K_16s_C1IR

Prototype

```
IppStatus ippiWTInv_D97_JPEG2K_16s_C1IR(Ipp16s *pSrcDstTile,
   int step, const IppiRect *pTileRect, Ipp8u *pBuffer);
```

Description

This function interleaves the LL, HL, LH and HH subbands of DWT coefficients and then makes a fix-point implementation of one-level inverse 2D wavelet transformation on them using the 9-7 irreversible filter. The results are written back to the buffer of input data. The image tile data are in 16-bit data type.

NOTE. *The output image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The input DWT coefficients from the buffer pointed by pSrcDstTile are de-interleaved into 4 subbands as shown in* Figure 18-4.

Input Arguments

- `pSrcDstTile` – pointer to the buffer of input LL, HL, LH and HH subbands of DWT coefficients. The start address of `pSrcDstTile` must be 8-byte aligned. It is better to make it 32-byte aligned.
- `step` – specifies the number of bytes in a line of the input data buffer. `step` is in bytes, and must be an integer multiple of 8.
- `pTileRect` – pointer to an IppiRect data structure which indicates the position and size of the image tile
- `pBuffer` – pointer to the work buffer for transform.The start address of `pBuffer` must be 8-byte aligned. It is better to make it 32-byte aligned.

  The work buffer pointed by `pBuffer` must be allocated before the function is called. Its size can be gotten by calling "`WTGetBufSize_D97_JPEG2K_16s_C1IR`".

Output Arguments

`pSrcDstTile` – pointer to the buffer of output image tile

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

NOTE. The work buffer pointed by pBuffer must be allocated before the function is called. Its size can be gotten by calling ippiWTGetBufSize_D97_JPEG2K_16s_C1IR.

NOTE. The start address of pSrcDstTile and pBuffer must be 8-byte aligned. It is better to make them 32-byte aligned.

NOTE. `step` is in bytes, and must be an integer multiple of 8.

NOTE. The output image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The input DWT coefficients from the buffer pointed by pSrcDstTile are de-interleaved into 4 subbands as shown in Figure 18-4.

WTInv_D97_JPEG2K_32s_C1IR

Prototype

```
IppStatus ippiWTInv_D97_JPEG2K_32s_C1IR (Ipp32s *pSrcDstTile,
   int step, const IppiRect *pTileRect, Ipp8u *pBuffer);
```

Description

This function interleaves the LL, HL, LH and HH subbands of DWT coefficients and then makes a fix-point implementation of one-level inverse 2D wavelet transformation on them using the 9-7 irreversible filter. The results are written back to the buffer of input data. The image tile data are in 32-bit data type.

NOTE. *The output image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The input DWT coefficients from the buffer pointed by pSrcDstTile are de-interleaved into 4 subbands as shown in* Figure 18-4.

Input Arguments

- `pSrcDstTile` – pointer to the buffer of input LL, HL, LH and HH subbands of DWT coefficients. The start address of `pSrcDstTile` must be 8-byte aligned. It is better to make it 32-byte aligned.
- `step` – specifies the number of bytes in a line of the input data buffer. `step` is in bytes, and must be an integer multiple of 8.
- `pTileRect` – pointer to an IppiRect data structure which indicates the position and size of the image tile
- `pBuffer` – pointer to the work buffer for transform. The start address of `pBuffer` must be 8-byte aligned. It is better to make it 32-byte aligned.

  The work buffer pointed by `pBuffer` must be allocated before the function is called. Its size can be retrieved by calling "WTGetBufSize_D97_JPEG2K_32s_C1IR".

Output Arguments

`pSrcDstTile` – pointer to the buffer of output image tile

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

NOTE. The work buffer pointed by pBuffer must be allocated before the function is called. Its size can be gotten by calling ippiWTGetBufSize_D97_JPEG2K_32s_C1IR.

NOTE. The start address of pSrcDstTile and pBuffer must be 8-byte aligned. It is better to make them 32-byte aligned.

NOTE. `step` is in bytes, and must be an integer multiple of 8.

NOTE. The output image tile can be either one tile-component of the image or the LL subband of the higher level's DWT coefficients. The input DWT coefficients from the buffer pointed by pSrcDstTile are de-interleaved into 4 subbands as shown in Figure 18-4.

NOTE. *pNumValidPrefetchedBits is 0 in following case:

- Initial
- After each restart interval
- After each marker

NOTE. If following conditions are not satisfied, this function returns ippStsBadArgErr.

- Each pointer cannot be NULL.
- *pSrcBitsLen is greater than or equal to 0.
- *pNumValidPrefetchedBits is greater than or equal to 0.
- The start address of pDst must be 4-byte aligned.

Usage Example of DWT Primitives

The general procedure of implementing an n-level DWT with Intel® IPP is as follows:

1. Call `GetBufSize` function to get the size of internal working buffer;
2. Allocate the working buffer for DWT;
3. Initialize the structure which defines the size and position of image tile;
4. Perform DWT on the image tile;
5. Update the tile structure to make LL subband to be next DWT's input;
6. Repeat 3) and 4) until n-level DWT is done;
7. Free allocated buffer.

Below is an example of using forward DWT(5-3) primitive to implement an n-level DWT of an image (stored in the buffer pointed by `pImage`) in JPEG 2000 encoding. The results are stored in subbands in the same buffer (Please refer to Figure1.) The position and size of the image is defined by `pImageRect`, and the byte number of each line is *step*. `pBuffer` points to the allocated work buffer, whose necessary size has been gotten by calling `ippiWTGetBufSize_B53_JPEG2K_16s_C1IR`.

```
#include "ippdefs.h"
#include "ippJP.h"

void DWT_fwd_53(Ipp16s *pImage, IppiRect *pImageRect, int step, Ipp8u *pBuffer,
int n)
{
IppiRect tileRect;

/* Initialize the structure of image tile's size */
tileRect.x = pImageRect->x;
tileRect.y = pImageRect->y;
tileRect.width = pImageRect->width;
tileRect.height = pImageRect->height;

/* n-level DWT */
for (; n > 0; n --) {
    ippiDWTFwd_53R1L_JPEG2K_16s_I (pImage, step, &tileRect, pBuffer);
/* Update the size to be that of the LL subband, which is the input for next DWT
*/
    tileRect.width = (tileRect.width — (tileRect.x&1) + 1 )/2;
    tileRect.height = (tileRect.height - (tileRect.y&1) + 1)/2;
}

return;
}
/* EOF */
```

Cryptography

19

This section describes the set of Intel® Integrated Performance Primitives (Intel® IPP) that are available for cryptography (cryptographic primitives).

The Intel® IPP - Cryptographic Primitives provide a rich and powerful set of cryptographic primitive functions optimized for the Intel® Personal Internet Client Architecture Processors with Intel® Wireless MMX™ technology (PCA processors with MMX™), which is compliant with the ARM Architecture V5TE.

Intel® IPP comprises a rich and powerful set of cryptographic processing functions optimized for maximum performance on the PCA processors with MMX™. Intel® IPP offers application developers a number of significant advantages:

- The primitives are optimized for maximum performance on the PCA processors with MMX™. Therefore, Intel® IPP enables migration to the PCA processors with MMX™ of many computationally intensive security applications that have traditionally required a special purpose DSP or cryptographic accelerator hardware.
- Intel® IPP - Cryptographic Primitives drastically reduces development costs and profoundly accelerates time-to-market by eliminating the need for hand optimization of routines commonly used for cryptographic operations. Developers are provided with an "off-the-shelf" optimized solution.
- Intel® IPP - Cryptographic Primitives is compatible with popular real-time embedded operating systems that run on the PCA processors with MMX™. The primitives are low-level and have been carefully designed to avoid host operating system (OS) dependencies.

Feature Summary

Intel® IPP includes cryptographic operation primitives optimized for the PCA processors with MMX™, and these primitives can be used to develop a security application product for PCA processors with MMX™ platforms.

Primitives available in this release for block cipher operations include the following:

- DES/TDES key scheduling, encryption and decryption under ECB, CBC and CFB modes with non-padding, zeros padding, as well as PKCS7 padding.
- AES/Rijndael key expansion, encryption and decryption under ECB, CBC and CFB modes with non-padding, zeros padding, as well as PKCS7 padding.
- Blowfish key scheduling, encryption and decryption under ECB, CBC and CFB modes with non-padding, zeros padding, as well as PKCS7 padding.
- Twofish key scheduling, encryption and decryption under ECB, CBC and CFB modes with non-padding, zeros padding, as well as PKCS7 padding.

Primitives available in this release for data authentication algorithms include the following:

- DAA-DES, DAA-TDES transform given message text (both streaming and non-streaming) with user authentication key into a data authentication code.
- DAA-Rijndael128, DAA-Rijndael192 and DAA-Rijndael256 transform given message text (both streaming and non-streaming) with user authentication key into a data authentication code.
- DAA-Blowfish transforms given message text (both streaming and non-streaming) with user authentication key into a data authentication code.
- DAA-Twofish transforms given message text (both streaming and non-streaming) with user authentication key into a data authentication code.

Primitives available in this release for one-way hash function operations include the following:

- SHA-1 transforms both streaming and non-streaming message into a 160-bit message digest.
- SHA-256, SHA-384 and SHA-512 transform both streaming and non-streaming message into 256-bit, 384-bit and 512-bit message digests correspondingly.
- MD5 transforms both streaming and non-streaming message into a 128-bit message digest.

Primitives available in this release for keyed-hash message authentication code (HMAC) include the following:

- HMAC-SHA1 transforms both streaming and non-streaming message into a (up to 160-bit) message authentication code with a user supplied authentication key.
- HMAC-SHA256, HMAC-SHA384 and HMAC-SHA512 transform both streaming and non-streaming message into (up to their respective 256-bit, 384-bit and 512-bit) message digests.
- HMAC-MD5 transforms both streaming and non-streaming message into an (up to 128-bit) message digest.

Primitives available in this release for public key cryptographic operations include the following:

- Big number arithmetic operation/Modulo reduction, inversion, exponentiation
- Pseudo-random number generation

- Probable prime number generation
- RSA key generation, encryption and decryption
- DSA key generation, signing, and verification

About This Section

This manual describes the functions that comprise the Intel® Integrated Performance Primitives - Cryptographic Primitives (Intel® IPP) for the PCA processors with MMX™. All of the functions have been organized according to the type of operation they perform. Each function is introduced by its name. This is followed by the function prototype, definitions of its arguments, and description of its purpose. Beyond the description, pseudo code and equations are given whenever appropriate to capture the fine details of the operation associated with a particular primitive.

Intended Audience

This manual assumes that the reader is a skilled programmer having some working knowledge of basic cryptographic vocabulary and principles. In this manual, presentations are geared towards developers who have had practical experience or theoretical exposure to cryptography, data security, or network communication.

Organization

This section provides background and introductory information on the Intel® IPP. The focus is on the essential Intel® IPP API elements, including header files, binary files, and data types.

"Block Cipher Primitives" gives details on the block cipher primitives that are provided for key scheduling, encryption and decryption under ECB, CBC, and CFB modes with multiple padding schemes using the block ciphers DES, Triple-DES, AES, Rijndael (various block lengths with key lengths), Blowfish, and Twofish.

"Data Authentication Algorithm Primitives" describes in detail on various data authentication algorithm (DAA) primitives based on the set of block ciphers listed in Chapter 3.

"One-Way Hash Function Primitives" provides details hash primitives that are used for digesting variable length message into a fixed size hash value or digital fingerprint using SHA-1, SHA-256, SHA-384, SHA-512 and MD5.

"Keyed-Hash Message Authentication Code Primitives" describes in detail on various keyed-hash message authentication code (HMAC) schemes primitives based on the set of on-way hash functions listed in Chapter 5.

"Public Key Cryptographic Primitives"describes primitives for general big number arithmetic operation, modulo reduction, inversion, exponentiation, pseudo-random number generation, as well as probable prime number generation. Also described are the primitive functions for RSA cryptographic operations and DSA digital signature operations.

Binary Image Files

The Intel® Integrated Performance Primitives - Cryptographic Primitives for the PCA processors with MMX™ are contained in a collection of static binary libraries, organized by function type, against which users must link their application object files. The versions of the image files that are included in the release are a debug version and a release version.

The defining difference between the release and debug versions lies in the input argument error/bounds checking. In the debug library build, each function validates its input parameters, and returns corresponding error messages whenever appropriate. The release version maximizes performance by eliminating the overhead associated with parameter error checking. It is recommended that users link against the debug library image during application development and debugging cycles and then link against the release image only to produce a final release candidate Table 19-1 summarizes the library binary files against which application object files should be linked when using particular primitives under particular operating systems.

Table 19-1 Intel® IPP Binary Image Files

Function Types	Windows CE	Linux
Cryptographic Primitives	1. ippCP_XSC41BPPC_r.lib 2. ippCP_XSC41BPPC_d.lib 3. ippCP_WMMX41BPPC_r.lib 4. ippCP_WMMX41BPPC_d.lib	1. ippCP_XSC41BLNX_r.a 2. ippCP_XSC41BLNX_d.a 3. ippCP_WMMX41BLNX_r.a 4. ippCP_WMMX41BLNX_d.a

Header Files

In order to maximize ease of use and portability, the prototypes for the complete set of Intel® IPP functions are defined in a set of ANSI-compliant 'C' language header files.

- <ippCP.h> – Use this header file to link against all primitives described in this manual.

Macros and Constants

Macros are defined symbolic names that are substituted with particular replacement text at compilation or assembly time. The Intel® IPP API makes available for developers several useful macros and constants. Developers should refer to the <ippCP.h> header file for the most up-to-date list of available macros and constants.Function Prototypes

Function Prototypes

The Intel® IPP API is comprised of 'C' language function prototypes for each primitive. The function prototype naming scheme adheres to the following format:

```
ipps<operation>_<data type>(parameter list)
```

Function Arguments

The Intel® IPP API convention for function argument lists can be expressed generally as follows:

```
<Input>, <Input data length>, <Output>, <Output data length>, <parameter list>.
```

Big Number Format

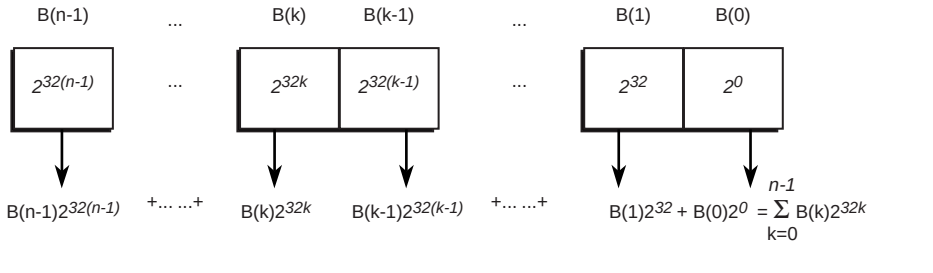
Primitives designed for modulus based public key cryptographic operations for the PCA processors with MMX™ requires an interpretation of the integer's input or output arguments whose values are much larger than what a 32-bit register can represent. The adopted "BigNum" format throughout this manual provides a standard mechanism for representing a big number argument.

For example, an 1024-bit valued big number employs an array of thirty-two 32-bit integers of unsigned long data type, i.e, {B[31]|B[30]| ... |B[1]|B[0]} to represent its value.

$$value = b_{31}2^{32\times 31} + b_{30}2^{32\times 30} + \ldots + b_1 2^{32\times 1} + b_0 2^{32\times 0}$$

where $B[i] = b_i \quad 0 \le i < 32$ is a 32-bit integer of unsigned long data type. In general, a (32*n)-bit valued big number requires an array of n 32-bit unsigned long integers, its value is illustrated in Figure 19-1, and detailed definition for data structure IppsBigNum is given in the "Big Number Arithmetic" section.

Figure 19-1 BN Format



Error Handling

The debug versions of the Intel® IPP provide error handling facilities that monitor and report bad arguments (for example, NULL pointers, out-of-range parameter values etc.) and "out of memory" conditions. An appropriate status code is returned upon detection of an error condition. For each function, a list of possible status codes is given under the detailed function descriptions in the remaining chapters of this manual.

NOTE. The release versions of the Intel® IPP do not provide error handling facilities. Users are responsible for ensuring that all primitives are used correctly. The benefit of disabled error handling is a guarantee of maximum application performance. It is recommended that debug version builds be used during application debug and development cycles. Developers should link against the release version of the Intel® IPP only when building the final release of an application.

List of Primitives

The Intel® IPP is a set of static binary libraries that is designed for optimizing the NIST FIPS-PUB standardized cryptographic algorithms running on both Linux and Microsoft® Windows® CE operating systems for the device that is powered by the Intel® Integrated Performance Primitives on Intel® PCA Processors with Intel® Wireless MMX™ Technology. The Intel® IPP offers the object modules within each static binary library for the construction of standardize cryptographic schemes in the following five fronts:

- The *Symmetric Cryptographic Object Module (SCOM)* implements a wide-range block cipher modules, including the FIPS 46-3 specified DES and Triple-DES, the FIPS 197 specified AES, as well as various Rijndael block ciphers, Blowfish and Twofish. All support the FIPS 81 defined ECB, CBC and CFB modes. SCOM also implements various padding schemes in this release, including no padding, padding with zeros and padding with the PKCS-7 defined scheme.
- The *Data Authentication Object Module (DAOM)* implements various data authentication algorithms defined in FIPS 113 under a set of baseline block ciphers listed in SCOM.They are DAA-DES, DAA-TDES, DAA-Rijndael128, DAA-Rijndael192, DAA-Rijndael256, DAA-Blowfish, as well as DAA-Twofish.
- The *One-way Hash function Object Module (OHOM)* implements a rich set of one-way hash functions listed as the FIPS 180-1 specified SHA-1, the FIPS draft SHA-256, SHA-384, and SHA512, as well as the MD5 hash function.
- The *keyed-hash Message Authentication Object Module (MAOM)* implements various message authentication algorithms defined in FIPS 198 under a set of one-way hash functions listed in OHOM.They are HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, HMAC-SHA512, as well as HMAC-MD5.
- The *Asymmetric Cryptographic Object Module (ACOM)* implements the RSA and DSA cryptographic algorithms of scalable key size. Both the implemented RSA and DSA key generation function could generate randomized system components of variable length with user supplied entropy seed and stimulus context bit stream. Both RSA and DSA modules also include the feature that allow configuring the its respective cryptographic system by taking the user supplied system components in case for resuming either a RSA or a DSA session. Once their systems established, the implemented RSA cryptographic function could be applied for either RSA encryption or decryption,; while DSA primitives could be applied for either DSA digital signature signing and verification.

In addition, Intel® IPP also offers a set of primitives that are optimized for various mathematical operations among big number integers of variable length. This set of primitives provides users a comprehensive feature list, including arithmetical operation, pseudo-random number generation, probable prime number generation, as well as modular reduction, inversion and exponentiation.

All these features are the critical building blocks for constructing various modulo based asymmetric cryptographic systems, such as Diffie-Hellman key exchange algorithm and ElGamal cryptographic scheme.

Symmetric Cryptographic Object Module

"Block Cipher Primitives" lists the set of available cryptographic primitives for block ciphers. In the context of secure data communication, block ciphers are heavily deployed to protect a massive message over open communication media, not only because of its adequate security strength that meets the application's security requirement, but also because of its algorithmic efficiency that allows to secure the communication in real-time.

The Intel® IPP provides primitives that offers cryptographic operations under ECB, CBC, and CFB modes using a rich set of block ciphers, such as DES, Triple DES (TDES), AES, Rijndael128, Rijndael192, Rijndael256, and Blowfish, Twofish. Detailed description of block cipher primitives are given in Block Cipher Primitives. Intel® IPP offers primitives supporting data encryption or decryption in the way that complies with:

- The ECB, CBC, and CFB modes, as specified in FIPS 81
- The DES/Triple-DES block cipher schemes, as specified in FIPS 46-3
- The AES block cipher scheme, as specified in FIPS 197
- The various Rijndael block cipher schemes
- The Blowfish block cipher scheme
- The Twofish block cipher scheme

Data Authentication Objective Module

"Data Authentication Algorithm Primitives" lists the set of available primitives for data authentication algorithm based on the set of the block ciphers. They are DAA-DES, DAA-TDES, DAA-Rijndael128, DAA-Rijndael192, DAA-Rijndael256, and DAA-Blowfish, DAA-Twofish. Detailed description of data authentication algorithm primitives are given in "Data Authentication Algorithm Primitives". In the context of the secure data communication, the data authentication algorithm (DAA) is used for detecting the unauthorized modification, both intentional or accidental, to the data. Intel® IPP offers primitives that generates data authentication code `Ipp8u *pDAC` of the input text in the way complying with:

- The DAA scheme, as specified in FIPS 113
- The DES/Triple-DES block cipher schemes, as specified in FIPS 46-3
- The AES block cipher scheme, as specified in FIPS 197
- The various Rijndael block ciphers
- Blowfish

- Twofish

One-Way Hash Function Object Module

"One-Way Hash Function Primitives" lists the set of available hash primitives for SHA-1, SHA-256, SHA384, SHA512 and MD5. In the context of secure data communication, a hash function is used to digest a message of arbitrary length into a short bit string with a fixed bit length known as a hash value, a message digest, or a digital fingerprint. The one-way characteristics of the hash function makes it computational infeasible to create message hash collision. Based on this one-way characteristics, the digital signature scheme generates a digital signature that binds with the given message. Intel® IPP offers primitives that generates a message digest `Ipp8u *pMD` of the given input message in the way complying with:

- The SHA-1 hash scheme, as specified in FIPS 180-1.
- The SHA-256, SHA-384 and SHA-512, as specified in the FIPS draft
- MD5 as specified in RFC 1321

Keyed-hash Message Authentication Object Module

"Keyed-hash Message Authentication Object Module" lists the set of available keyed-hash message authentication code scheme primitives. They are HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, HMAC-SHA512, and HMAC-MD5. Detailed description of the key-hash message authentication code primitives are given in "Keyed-Hash Message Authentication Code Primitives". In the context of secure data communication, a keyed-hash message authentication code function is used by the message sender to produce a value (the MAC) that is formed by condensing the secrete key and the message input. The message receiver, however, could easily authenticate the integrity of the received message simply by validating the received MAC value against the one computed using the same HMAC function with the same key to the given message. Intel® IPP offers primitives that generates the message authentication code `Ipp8u *pMAC` of the input message in the way complying with:

- The HMAC scheme, as specified in the FIPS 198.
- The SHA-1, as specified in the FIPS 180-1
- The SHA-256, SHA-384 and SHA-512, as specified in the FIPS draft
- MD5 as specified in RFC 1321

Asymmetric Cryptographic Object Module

"Public Key Cryptographic Primitives" lists the set of available cryptographic primitives for a modulus-based public key cipher system. In the context of secure data communication, the public key cipher scheme is the most commonly used for the transport of keys subsequently used for bulk data encryption by block cipher algorithms. Intel® IPP offers primitives supporting:

- Big number arithmetic operations/modulo reduction, inversion and exponentiation
- Pseudo-random number generation
- Probable prime number generation
- Cryptographic operations for RSA system
- DSA digital signature operations

Block Cipher Primitives

This section describes the Intel® IPP cryptographic primitives that are available for block cipher operations.

A block cipher operates on a message block with a fixed-block-size. It encrypts a plain-text message block into a cipher-text block, and it decrypts a cipher-text block into a plain-text message block. For messages exceeding the block size, the simplest approach is to partition it into multiple blocks with possible padding at the trailing end and encrypt each block separately. Intel® IPP offers primitives supporting the three most active modes as specified by FIPS-81:

- ECB (Electronic CodeBook) mode
- CBC (Cipher Block Chain) mode
- CFB (Cipher Feedback) mode

Throughout this section, E_K represents the encryption function of the block cipher `E` using key `K`, while D_K represents the decryption function using key `K`. Figure 19-2 shows both ECB and CBC modes of the operation. Figure 19-3 shows CFB Mode.

Figure 19-2 ECB and CBC Modes

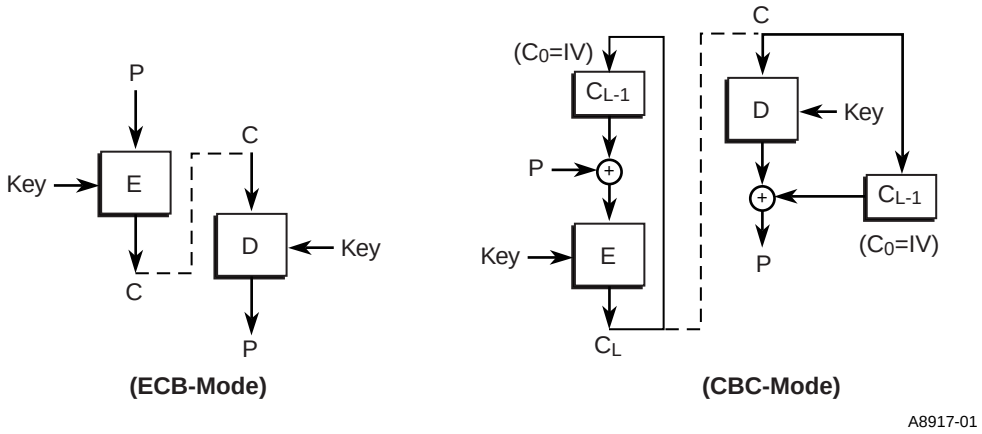
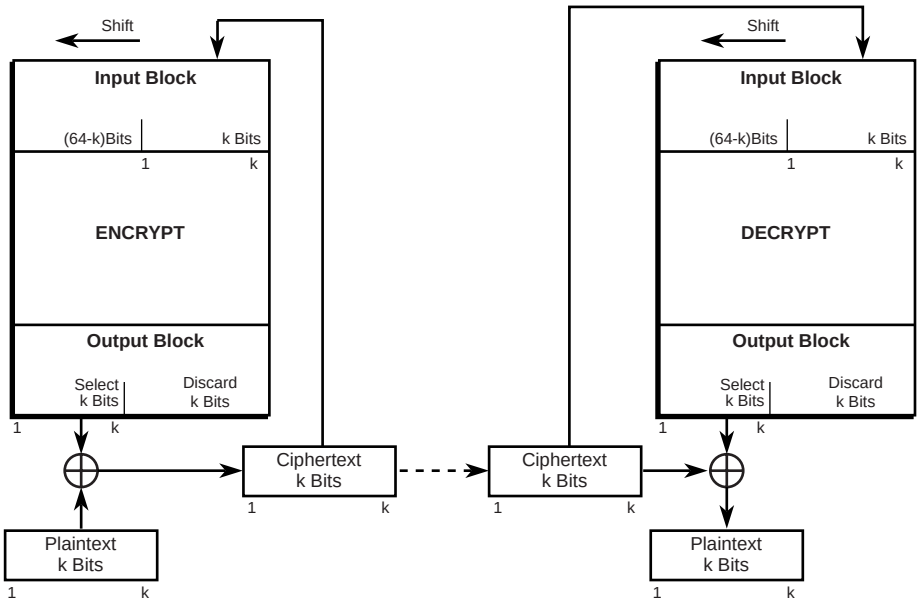


Figure 19-3 CFB Mode



To cipher a data stream of variable (byte) length, the cryptographic primitives in this section require the application to specify the plaintext message length and the desired padding scheme `padding` to pad the message stream to be an integral number of its respective block length. As such, the encryption functions can partition the incoming source plaintext stream `*pSrc` into multiple data blocks with the requested padding assist and encrypt them consecutively.

The decryption functions can also compute the message length (integral number of block length) for the incoming source ciphertext data stream `*pSrc` based on the data length `dstlen` specified for the resulting plaintext data stream `*pDst`. The decryption functions can then read the source ciphertext stream, partition it into multiple data blocks and decrypt them consecutively. Upon completion, the decryption function will be able to validate the decryption result, simply by checking if the final decrypted plaintext data block matches the user specified padding scheme.

This section implements the following three padding schemes:

```
typedef enum
{
```

```
NONE=0, IppsCPPaddingNONE=0,
ZEROS=1,IppsCPPaddingZEROS=1,
PKCS7=2,IppsCPPaddingPKCS7=2
} IppsCPPadding;
```

With the choice of `padding == PKCS7` or `padding == IppsCPPaddingPKCS7`, all the ECB and CBC encryption primitives in this section pad the input data stream `Ipp8u *pSrc` at its trailing end with `(CipherBlkLen - (srclen mod CipherBlkLen))` bytes data all having value `(CipherBlkLen - (srclen mod CipherBlkLen))`, where `CipherBlkLen = 8, 16, 24`, or `32` corresponding to the 64-bit, 128-bit, 192-bit or 256-bit block cipher scheme.

All the CFB mode encryption primitives in this section pad the input data stream `Ipp8u *pSrc` at its trailing end with `(cfbBlkSize - (srclen mod cfbBlkSize))` bytes data all having value `(cfbBlkSize - (srclen mod cfbBlkSize))`.

With the choice of `padding == ZEROS` or `padding == IppsCPPaddingZEROS`, all the ECB and CBC encryption primitives pad the input data stream `Ipp8u *pSrc` at its trailing end with `(CipherBlkLen - srclen mod CipherBlkLen))` bytes data all having value `0`, and all the CFB encryption primitives pad `(cfbBlkSize - (srclen mod cfbBlkSize))` bytes data all having value 0.

With the choice of `padding == NONE` or `padding == IppsCPPaddingNONE`, all the ECB, CBC and CFB encryption primitives in this section will not pad the input data stream `Ipp8u *pSrc` at its trailing end.

In the interest of simplicity and consistency, the mathematical expression and pseudocode given in this section describe the behavior of each function.

The rest of this section is organized as follows:

- Primitive functions supporting ECB, CBC and CFB modes operations using DES and Triple DES block ciphers are described in the "DES/TDES" section.
- Primitive functions supporting ECB, CBC and CFB modes operations using Rijndael block ciphers of 128-bit, 192-bit and 256-bit block length are described in the "Rijndael128/192/256" section.
- Primitive functions supporting ECB, CBC and CFB modes operations using Blowfish block cipher are described in "Blowfish".
- Primitive functions supporting ECB, CBC and CFB modes operations using Twofish block cipher are described in "Twofish".

DES/TDES

DES (Data Encryption Standard) is the most well-known block symmetric cipher and it is the first commercial-grade modern algorithm with open and fully specified implementation details. It consists of a `16` rounds iterated Feistel network. The block size is `64` bits and its effective key size

is 56 bits. TDES is a newly revised block cipher scheme built on the DES system. Its encryption process consists of three consecutive DES operations in the sequence of encryption, decryption, and encryption (E-D-E) as specified by the American Standard FIPS 46-3. Although the primitives supporting TDES operations described in this section require three sets of round keys, they are capable of operating under the TDES cipher system with a two-set round keys, by simply setting the third set of round keys to be the same as the first set.

This section describes the primitives available in the Intel® IPP for performing various operational modes under the DES/TDES cipher systems. All the primitives described in this section are implemented to comply with the American Standard FIPS 46-3. The primitive function `ippsDESInit()` generates a set of round keys for either DES or TDES cryptographic system. Once the round keys are generated, the function `ippsDESEncrypt_I()` is ready for encrypting a single input data block of 64-bit. In the case for TDES, the function `ippsDESInit()` needs to be called three times as to generate all three sets of the round keys used in the sequence of E-D-E operations for TDES. Once three sets of round keys are generated, the Intel® IPP primitive functions `ippsTDESEncrypt_I()` and `ippsTDESDecrypt_I()` are ready for performing their respective cryptographic operations.

All DES and Triple DES primitives in this section support ECB, CBC, and CFB modes. For example, the primitive `ippsDESEncryptECB()` operates under ECB mode for DES encryption, while the primitive `ippsTDESEncryptECB()` operates on the ECB mode for the respective TDES scheme.

Primitives `ippsDESEncryptCBC()` and `ippsTDESEncryptCBC()` operate under CBC mode for encryption using their respective cipher scheme, both require having an initialization vector `iv`. The initialization vector `iv` can be initialized with anything, but the recipient needs to know what it was initialized with to decrypt.

Primitives `ippsDESEncryptCFB()` and `ippsTDESEncryptCFB()` operate under CFB mode for encryption using their respective cipher scheme, both require having an initialization vector `pIV`, and CFB block size `cfbBlkSize`.

Throughout this section, all the primitives employ a symbolic structure `ippsDES` to serve as an operational vehicle which carries a set of round keys. The application code could use these symbolic structures to perform various DES/TDES cryptographic operations.

The application code for conducting a typical encryption under CBC mode using the Triple DES scheme (a similar procedure could be applied for ECB or CFB modes of operation) should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsDES` by calling the primitive `ippsDESBuffersize()`.
2. Call the OS memory-allocation service function to allocate `3` buffers whose sizes are no less than the one specified by the primitive `ippsDESBuffersize()`. Initialize three sets of round keys `IppsDES *pCtx1`, `*pCtx2` and `*pCtx3` by calling the primitive `ippsDESInit()` three times, each with the allocated buffer and the respective DES key.
3. Specify the initialization vector and the padding scheme, than call the primitive `ippsTDESEncryptCBC()` to encrypt the input data stream using the Triple DES CBC encryption function.
4. Call the OS memory-free service function to release the buffers allocated for three symbolic structures `ippsDES` if needed.

DESBufferSize

Prototype

```
IppStatus ippsDESBufferSize(int *psize);
```

Description

`ippsDESBufferSize()` specifies the buffer size (in number of bytes) required for defining a symbolic structure `IppsDES` for the storage of `16`-round keys used for both DES encryption and decryption.

Input Arguments

- None

Output Arguments

- `psize` – the size of required buffer (in bytes) for initializing the respective `IppsDES`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

DESInit

Prototype

```
IppStatus ippsDESInit(const Ipp8u *pkey, IppsDES *pCtx);
```

Description

`ippsDESInit()` initializes the symbolic structure `IppsDES *pCtx`, using user supplied buffer space pointed by the input `*pCtx`. In addition, `ippsDESInit()` uses the DES key expansion scheme specified by the American Standard FIPS 46-3 to compute a set of sixteen `48`-bit DES round keys `IppsDES *PCtx` with the user supplied `56`-bit DES key `Ipp8u *pKey`.

Input Arguments

- `pkey` – a `56`-bit DES key
- `pCtx` – pointer to the user supplied buffer

Output Arguments

- `pCtx` – an initialized symbolic structure `IppsDES`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

DESEncrypt_I

Prototype

```
IppStatus ippsDESEncrypt_I(Ipp64u *pSrcDstData, const IppsDES *pCtx);
```

Description

ippsDESEncrypt_I() is a DES block encryption function. It uses the FIPS 46-3 specified cipher scheme to encrypt a single DES block (64-bit) of plaintext Ipp64u *pSrcDstData with the user supplied set of round keys IppsDES *pCtx, and it returns the ciphertext result back to *pSrcDstData.

The following pseudocode represents this function:

pSrcDstData = E_{pCtx}(pSrcDstData);

where E_{pCtx} represents the DES encryption operation using a set of round keys supplied by pCtx.

Input Arguments

- pSrcDstData – a single DES plaintext data block
- rk – a set of round keys scheduled for DES internal iterations

Output Arguments

- pSrcDstData – the ciphertext result

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers

DESDecrypt_I

Prototype

```
IppStatus ippsDESDecrypt_I(Ipp64u *pSrcDstData, const IppsDES *pCtx);
```

Description

ippsDESDecrypt_I() is a DES block decryption function. It uses the FIPS 46-3 specified cipher scheme to decrypt a single DES block (64-bit) of ciphertext Ipp64u *pSrcDstData with the user supplied set of round keys IppsDES *pCtx, and it returns the plaintext result back to *pSrcDstData.

The following pseudocode represents this function:

pSrcDstData = D_{pCtx}(pSrcDstData);

where D_{pCtx} represents the DES decryption operation using a set of round keys supplied by pCtx.

Input Arguments

- pSrcDstData – a single DES ciphertext data block
- pCtx – a set of round keys scheduled for DES internal iterations

Output Arguments

- pSrcDstData – the plaintext result

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers

TDESEncrypt_I

Prototype

```
IppStatus ippsTDESEncrypt_I(Ipp64u *pSrcDstData, const IppsDES *pCtx1,
   const IppsDES *pCtx2, const IppsDES *pCtx3);
```

Description

ippsTDESEncrypt_I() is the Triple DES encryption function that follows FIP 46-3. It encrypts a single DES data block of plaintext Ipp64u *pSrcDstData using three sets of round keys IppsDES *Ctx1, *Ctx2 and *Ctx3. It returns the ciphertext result back to *pSrcDstData.

The following pseudocode represents this function:

$pSrcDstData = E_{pCtx3}(D_{Ctx2}(E_{Ctx1}(pSrcDstData)));$

Input Arguments

- pSrcDstData – the plaintext data block
- pCtx1 – the first set of round keys
- pCtx2 – the second set of round keys
- pCtx3 – the third set of round keys

Output Arguments

- `pSrcDstData` – the ciphertext result

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – bad arguments

TDESDecrypt_I

Prototype

```
IppStatus ippsTDESDecrypt_I(Ipp64u *pSrcDstData, const IppsDES *pCtx1,
   const IppsDES *pCtx2, const IppsDES *pCtx3);
```

Description

`ippsTDESDecrypt_I()` is the Triple DES decryption function that follows FIP 46-3. It decrypts a single DES data block of ciphertext `Ipp64u *pSrcDstData` using three sets of round keys `IppsDES *pCtx1`, `*pCtx2` and `*pCtx3`. It returns the plaintext result back to `*pSrcDstData`.

The following pseudocode represents this function:

$$pSrcDstData = D_{pCtx1}(E_{pCtx2}(D_{pCtx3}(pSrcDstData)));$$

Input Arguments

- `pSrcDstData` – the ciphertext data block
- `pCtx1` – the first set of round keys
- `pCtx2` – the second set of round keys
- `pCtx3` – the third set of round keys

Output Arguments

- `pSrcDstData` – the plaintext result

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

DESEncryptECB

Prototype

```
IppStatus ippsDESEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
   const IppsDES *pCtx, IppsCPPadding padding);
```

Description

`ippsDESEncryptECB()` is the DES encryption Electronic Code Book (ECB) function. ECB mode follows FIPS 81. It uses a set of round keys `IppsDES *pCtx` to execute the ECB mode of the DES encryption scheme to the given input message stream `Ipp8u *pSrc` of variable (byte) length `srclen`, padding with the user specified scheme `IppsCPPadding padding`. It returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream
- `pCtx` – the set of round keys scheduled for DES internal iterations
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` - if `srclen` is not divisible by 8 in case `padding==IppsCPPaddingNONE`
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS or IppsCPPaddingPKCS7`

DESDecryptECB

Prototype

```
IppStatus ippsDESDecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
   const IppsDES *pCtx, IppsCPPadding padding);
```

Description

`ippsDESDecryptECB()` is the DES decryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round keys `IppsDES *pCtx` to execute the ECB mode of the DES decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+7)/8]*8`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen` – the (byte) length of the decrypted plaintext data stream
- `pCtx` – the set of round keys scheduled for DES internal iterations
- `padding` – the padding scheme code

Output Arguments

- `pdst` – the resulting plaintext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `dstlen < or = 0`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7`

DESEncryptCBC

Prototype

```
IppStatus ippsDESEncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
   const IppsDES *pCtx, const Ipp8u *pIV,
   IppsCPPadding padding);
```

Description

`ippsDESEncryptCBC()` is the DES encryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsDES *pCtx` to execute the CBC mode of the DES encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`, padding with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream
- `pCtx` – the set of round keys scheduled for DES internal iterations
- `pIV` – initial vector
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` – if `srclen` is not divisible by 8 in case `padding==IppsCPPaddingNONE`
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

DESDecryptCBC

Prototype

```
IppStatus ippsDESDecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
   const IppsDES *pCtx, const Ipp8u *pIV,
   IppsCPPadding padding);
```

Description

`ippsDESDecryptCBC()` is the DES decryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsDES *pCtx` to execute the CBC mode of the DES decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+7)/8]*8` with initialization vector `*pIV`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen` – the (byte) length of the decrypted plaintext data stream.
- `pCtx` – the set of round keys scheduled for DES internal iterations
- `pIV` – initial vector
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `dstlen < or = 0`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

DESEncryptCFB

Prototype

```
IppStatus ippsDESEncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
   int cfbBlkSize, const IppsDES *pCtx,
   const Ipp8u *pIV, IppsCPPadding padding);
```

Description

`ippsDESEncryptCFB()` is the DES encryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys `IppsDES *pCtx` to execute the CFB mode of the DES encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`, padding with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of user specified CFB block size `int cfbBlkSize`) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream
- `cfbBlkSize` – CFB block size (in unit of bytes)
- `pCtx` – the set of round keys scheduled for DES internal iterations
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` – if `srclen` is not divisible by `cfbBlkSize` in case `padding == IppsCPPaddingNONE`
- `ippStsCFBSizeErr` – wrong value for `cfbBlkSize`

- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

DESDecryptCFB

Prototype

```
IppStatus ippsDESDecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
   int cfbBlkSize, const IppsDES *pCtx,
   const Ipp8u *pIV, IppsCPPadding padding);
```

Description

ippsDESDecryptCFB() is the DES decryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys IppsDES *pCtx to execute the CFB mode of the DES decryption scheme to the given input ciphertext stream *pSrc of integral number (bytes) of the user specified CFB block size cfbBlkSize calculated as [(dstlen+cfbBlkSize-1)/cfbBlkSize]*cfbBlkSize with initialization vector *pIV, and returns the decrypted plaintext result of user specified message length dstlen to Ipp8u *pDst. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input IppsCPPadding padding.

Input Arguments

- pSrc – input ciphertext data stream
- dstlen – the (byte) length of the decrypted plaintext data stream
- cfbBlkSize – CFB block size (in unit of bytes)
- pCtx– the set of round keys scheduled for DES internal iterations
- pIV – initial vector
- padding– the padding scheme code

Output Arguments

- pDst – the decrypted plaintext data stream

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers

- ippStsLengthErr – if dstlen < or = 0
- ippStsCFBSizeErr – wrong value for cfbBlkSize
- ippStsPaddingErr – the final plaintext data block violate the user specified padding scheme
- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

TDESEncryptECB

Prototype

```
IppStatus ippsTDESEncryptECB (const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsDES *pCtx1, const IppsDES *pCtx2,
    const IppsDES *pCtx3, IppsCPPadding padding);
```

Description

ippsTDESEncryptECB() is the TDES encryption Electronic Code Book (ECB) function. ECB mode follows FIPS 81. It uses three sets of the round keys IppsDES *pCtx1, *pCtx2 and *pCtx3 to execute the ECB mode of the TDES encryption scheme to the given input message stream Ipp8u *pSrc of variable (byte) length srclen, padding with the user specified scheme IppsCPPadding padding. It returns the ciphertext result (integral number of block length) to Ipp8u *pDst.

Input Arguments

- pSrc – input plain text data stream of variable length
- srclen – the (byte) length of the input data stream
- pCtx1 – the first set of round keys scheduled for TDES internal iterations
- pCtx2 – the second set of round keys scheduled for TDES internal iterations
- pCtx3 – the third set of round keys scheduled for TDES internal iterations
- padding – the padding scheme code

Output Arguments

- pDst – the resulting ciphertext data stream

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr – if srclen < or = 0
- ippStsUnderRunErr – if srclen is not divisible by 8 in case padding==IppsCPPaddingNONE
- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

TDESDecryptECB

Prototype

```
IppStatus ippsTDESDecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen, const
    IppsDES *pCtx1, const IppsDES *pCtx2,
    const IppsDES *pCtx3, IppsCPPadding padding);
```

Description

ippsTDESDecryptECB() is the TDES decryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses three sets of round keys IppsDES *pCtx1, *pCtx2, and *pCtx3 to execute the ECB mode of the TDES decryption scheme to the given input ciphertext stream *pSrc of integral number (bytes) of block length calculated as [(dstlen+7)/8]*8, and returns the decrypted plaintext result of user specified message length dstlen to Ipp8u *pDst. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input IppsCPPadding padding.

Input Arguments

- pSrc – input ciphertext data stream
- dstlen – the (byte) length of the decrypted plaintext data stream
- pCtx1 – the first set of round keys scheduled for TDES internal iterations
- pCtx2 – the second set of round keys scheduled for TDES internal iterations
- pCtx3 – the third set of round keys scheduled for TDES internal iterations
- padding – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if dstlen < `or` = `0`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

TDESEncryptCBC

Prototype

```
IppStatus ippsTDESEncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
   const IppsDES *pCtx1,
   const IppsDES *pCtx2,
   const IppsDES *pCtx3,
   const Ipp8u *pIV,
   IppsCPPadding padding);
```

Description

`ippsTDESEncryptCBC()` is the TDES encryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses three sets of round keys `IppsDES *pCtx1`, `*pCtx2` and `*pCtx3` to execute the CBC mode of the TDES encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`, padding with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream
- `pCtx1` – the first set of round keys scheduled for TDES internal iterations

- pCtx2 – the second set of round keys scheduled for TDES internal iterations
- pCtx3 – the third set of round keys scheduled for TDES internal iterations
- pIV – initial vector
- padding– the padding scheme code

Output Arguments

- pDst – the resulting ciphertext data stream

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr – if srclen < or = 0
- ippStsUnderRunErr – if srclen is not divisible by 8 in case padding==IppsCPPaddingNONE
- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

TDESDecryptCBC

Prototype

```
IppStatus ippsTDESDecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
   const IppsDES *pCtx1,
   const IppsDES *pCtx2,
   const IppsDES *pCtx3, const Ipp8u *pIV,
   IppsCPPadding padding);
```

Description

ippsTDESDecryptCBC() is the TDES decryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses three sets of round keys IppsDES *pCtx1, *pCtx2 and *pCtx3 to execute the CBC mode of the TDES decryption scheme to the given input ciphertext stream *pSrc of integral number (bytes) of block length calculated as [(dstlen+7)/8]*8 with initialization vector *pIV, and returns the decrypted plaintext result of user specified message

length dstlen to Ipp8u *pDst. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input IppsCPPadding padding.

Input Arguments

- pSrc – input ciphertext data stream
- dstlen – the (byte) length of the decrypted plaintext data stream
- pCtx1 – the first set of round keys scheduled for TDES internal iterations
- pCtx2 – the second set of round keys scheduled for TDES internal iterations
- pCtx3 – the third set of round keys scheduled for TDES internal iterations
- pIV – initial vector
- padding – the padding scheme code

Output Arguments

- pDst – the decrypted plaintext data stream

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr – if dstlen < or = 0

TDESEncryptCFB

Prototype

```
IppStatus ippsTDESEncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
   int cfbBlkSize, const IppsDES *pCtx1, const IppsDES *pCtx2, const
   IppsDES *pCtx3, const Ipp8u *pIV, IppsCPPadding padding);
```

Description

ippsTDESEncryptCFB() is the TDES encryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses three sets of round keys IppsDES *pCtx1, *pCtx2 and *pCtx3 to execute the CFB mode of the DES encryption scheme to the given input message stream *pSrc of variable (byte)

length `srclen`, padding with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of user specified CFB block size `int cfbBlkSize`) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen`– the (byte) length of the input data stream.
- `cfbBlkSize` – CFB block size (in unit of bytes)
- `pCtx1` – the first set of round keys scheduled for TDES internal iterations.
- `pCtx2` – the second set of round keys scheduled for TDES internal iterations.
- `pCtx3` – the third set of round keys scheduled for TDES internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If `srclen < or = 0.`
- ippStsUnderRunErr - If `srclen` is not divisible by `cfbBlkSize` in case `padding == IppsCPPaddingNONE.`
- ippStsCFBSizeErr - wrong value for `cfbBlkSize`
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

TDESDecryptCFB

Prototype

```
IppStatus ippsTDESDecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
   int cfbBlkSize, const IppsDES *pCtx1, const IppsDES *pCtx2, const
   IppsDES *pCtx3, const Ipp8u *pIV, IppsCPPadding padding);
```

Description

`ippsTDESDecryptCFB()` is the TDES decryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses three sets of round keys `IppsDES *pCtx1, *pCtx2` and `*pCtx3` to execute the CFB mode of the TDES decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of the user specified CFB block size `cfbBlkSize` calculated as `[(dstlen+cfbBlkSize-1)/cfbBlkSize]*cfbBlkSize` with initialization vector `*pIV,` and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding.`

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen`– the (byte) length of the decrypted plaintext data stream.
- `cfbBlkSize` – CFB block size (in unit of bytes)
- `pCtx1` – the first set of round keys scheduled for TDES internal iterations.
- `pCtx2` – the second set of round keys scheduled for TDES internal iterations.
- `pCtx3` – the third set of round keys scheduled for TDES internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If `dstlen < or = 0`
- ippStsCFBSizeErr - wrong value for `cfbBlkSize`
- ippStsPaddingErr - The final plaintext data block violate the user specified padding scheme
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

Rijndael128/192/256

Rijndael cipher scheme is an iterated block cipher with a variable block length and a variable key length. Its data block length and the key length can be independently specified to 128, 192, or 256 bits. Rijndael block ciphers of various data block length and key length combinations are gaining

more acceptance simply because they have gone through the thorough review process with extensive cryptographic analysis required by the US NIST in the process of becoming its Advanced Encryption Standard (AES).

This section describes the primitives available in the Intel® IPP for performing various operational modes under the various Rijndael cipher systems. The primitives in this section are categorized by their data block lengths of the baseline Rijndael cipher functions. Rijndael128 refers to the Rijndael cipher scheme with 128-bit data block length, Rijndael256 refers to the Rijndael cipher scheme with 192-bit data block length, and Rijndael256 refers to the Rijndael cipher scheme with 256-bit data block length. To specify the key length for these baseline Rijndael cipher schemes, all the primitives in this section use the following enumeration

```
typedef enum {
         IppsRijndaelKey128 = 128, // 128-bit key
         IppsRijndaelKey192 = 192, // 192-bit key
         IppsRijndaelKey256 = 256, // 256-bit key
} IppsRijndaelKeyLength;
```

It should be noted that the primitives for Rijndael128 with 128-bit key length described in this section are AES cipher primitive functions and they are implemented in the way to comply with the American Standard FIPS 197, and all other primitives for various other Rijndael block cipher schemes are fully complying to the respective cipher schemes documented in the file <<AES Proposal: Rijndael>> submitted by Joan Daeman and Vincent Rijmen.

Throughout this section, the primitives for Rijndael128 baseline cipher schemes employ the symbolic data structure IppsRijndael128, the primitives for Rijndael192 baseline cipher schemes employ the symbolic data structure IppsRijndael192 and the primitives for Rijndael256 baseline cipher schemes employ the symbolic data structure IppsRijndael256. They are so defined as to serve as the operational vehicles to not only carry both a set of round keys and a set of round inverse keys, but also the key management information.

Once the round keys are generated by the respective initialization function, the primitives for ECB, CBC and CFB modes are ready to the execution of either encrypting or decrypting the streaming data with the padding scheme of the user's choice.

The application code for conducting a typical encryption under CBC mode using AES scheme (That is, the Rijndael128 with 128-bit key) should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsRijndael128` by calling the primitive `ippsRijndael128Buffersize()`.
2. Call OS memory-allocation service function to allocate a buffer whose size is no less than the one specified by the primitive `ippsRijndael128Buffersize()`. Initialize the symbolic structure `IppsRijndael128 *pCtx` by calling the primitive `ippsRijndael128Init()` with the allocated buffer and the respective 128-bit AES key.

3. Specify the initialization vector and the padding scheme, than call the primitive `ippsRijndael128EncryptCBC()` to encrypt the input data stream using the AES encryption function with CBC mode.
4. Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsRijndael128`, if needed

Rijndael128BufferSize

Prototype

```
IppStatus ippsRijndael128BufferSize(int *pSize);
```

Description

`ippsRijndael128BufferSize()` specifies the buffer size (in number bytes) required for defining a symbolic structure `IppsRijndael128`. This carries not only a set of round keys and a set of round inverse keys used for various Rijndael128 cipher functions (with various key lengths) for both encryption and decryption, but also the key management information.

Input Arguments

- None

Output Arguments

- `pSize` – the size of required buffer (in bytes) for initializing `IppsRijndael128`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

Rijndael128Init

Prototype

```
IppStatus ippsRijndael128Init(const Ipp8u *pKey, IppsRijndaelKeyLength keylen,
   IppsRijndael128 *pCtx);
```

Description

`ippsRijndael128Init()` initializes the symbolic structure `IppsRijndael128 *pCtx`, using user supplied buffer space pointed by the input `*pCtx`. In addition, `ippsRijndael128Init()` uses the Rijndael key expansion scheme to compute both a set of round keys and a set of round inverse keys for the symbolic structure `IppsRijndael128 *pCtx` with the user supplied key byte stream `Ipp8u *pKey`, and `IppsRijndaelKeyLength keylen`.

Input Arguments

- `pKey` – key byte stream
- `keylen` – length of the input key bit stream
- `pCtx` – pointer to the user supplied buffer

Output Arguments

- `pCtx` – an initialized symbolic structure `IppsRijndael128`.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `keylen != IppsRijndaelKey128, IppsRijndaelKey192` or `IppsRijndaelKey256`

Rijndael128EncryptECB

Prototype

```
IppStatus ippsRijndael128EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsRijndael128 *pCtx,
    IppsCPPadding padding);
```

Description

`ippsRijndael128EncryptECB()` is the Rijndael128 encryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round keys `IppsRijndael128 *pCtx` to execute the ECB mode of the Rijndael128 encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`. Padding with the user specified scheme `IppsCPPadding padding`. It returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen`– the (byte) length of the input data stream.
- `pCtx` – the set of round keys scheduled for Rijndael128 internal iterations.
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` – if `srclen` is not divisible by 16 in case `padding==IppsCPPaddingNONE`
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael128DecryptECB

Prototype

```
IppStatus ippsRijndael128DecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    const IppsRijndael128 *pCtx,
    IppsCPPadding padding);
```

Description

`ippsRijndael128DecryptECB()` is the Rijndael128 decryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round inverse keys `IppsRijndael128 *pCtx` to execute the ECB mode of the Rijndael128 decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+15)/16]*16`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen` – the (byte) length of the decrypted plaintext data stream
- `pCtx` – the set of round inverse keys scheduled for Rijndael128 internal iterations
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `dstlen < or = 0`
- `ippStsPaddingErr` – The final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` - If `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael128EncryptCBC

Prototype

```
IppStatus ippsRijndael128EncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsRijndael128 *pCtx, const Ipp8u *pIV,
    IppsCPPadding padding);
```

Description

`ippsRijndael128EncryptCBC()` is the Rijndael128 encryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsRijndael128 *pCtx` to execute the CBC mode of the Rijndael128 encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`. Padding is with the user-specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. The function returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream
- `pCtx` – the set of round keys scheduled for Rijndael128 internal iterations
- `pIV` – initial vector
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` – if `srclen` is not divisible by 16 in case `padding==IppsCPPaddingNONE`
- `ippStsPaddingSchemeErr` - If `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael128DecryptCBC

Prototype

```
IppStatus ippsRijndael128DecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    const IppsRijndael128 *pCtx, const Ipp8u *pIV,
    IppsCPPadding padding);
```

Description

`ippsRijndael128DecryptCBC()` is the Rijndael128 decryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round inverse keys `IppsRijndael128 *pCtx` to execute the CBC mode of the Rijndael128 decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+15)/16]*16` with initialization vector `*pIV`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen` – the (byte) length of the decrypted plaintext data stream
- `pCtx` – the set of round inverse keys scheduled for Rijndael128 internal iterations.
- `pIV` – initial vector
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `dstlen < or = 0`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme

- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

Rijndael128EncryptCFB

Prototype

```
IppStatus ippsRijndael128EncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    int cfbBlkSize, const IppsRijndael128 *pCtx,
    const Ipp8u *pIV, IppsCPPadding padding);
```

Description

ippsRijndael128EncryptCFB() is the Rijndael128 encryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys IppsRijndael128 *pCtx to execute the CFB mode of the Rijndael128 encryption scheme to the given input message stream *pSrc of variable (byte) length srclen. Padding is with the user-specified scheme IppsCPPadding padding and with initialization vector *pIV. It returns the ciphertext result (integral number of user specified CFB block size int cfbBlkSize) to Ipp8u *pDst.

Input Arguments

- pSrc – input plain text data stream of variable length
- srclen – the (byte) length of the input data stream.
- cfbBlkSize – CFB block size (in unit of bytes)
- pCtx – the set of round keys scheduled for Rijndael128 internal iterations
- pIV – initial vector
- padding – the padding scheme code

Output Arguments

- pDst – the resulting ciphertext data stream.

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr – if srclen < or = 0

- ippStsUnderRunErr – if srclen is not divisible by cfbBlkSize in case padding == IppsCPPaddingNONE
- ippStsCFBSizeErr – wrong value for cfbBlkSize
- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

Rijndael128DecryptCFB

Prototype

```
IppStatus ippsRijndael128DecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    int cfbBlkSize, const IppsRijndael128 *pCtx
    const Ipp8u *pIV, IppsCPPadding padding);
```

Description

ippsRijndael128DecryptCFB() is the Rijndael128 decryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round inverse keys IppsRijndael128 *pCtx to execute the CFB mode of the Rijndael128 decryption scheme to the given input ciphertext stream *pSrc of integral number (bytes) of the user specified CFB block size cfbBlkSize calculated as [(dstlen+cfbBlkSize-1)/cfbBlkSize]*cfbBlkSize with initialization vector *pIV, and returns the decrypted plaintext result of user specified message length dstlen to Ipp8u *pDst. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input IppsCPPadding padding.

Input Arguments

- pSrc – input ciphertext data stream
- dstlen – the (byte) length of the decrypted plaintext data stream
- cfbBlkSize – CFB block size (in unit of bytes)
- pCtx – the set of round inverse keys scheduled for Rijndael128 internal iterations
- pIV – initial vector
- padding – the padding scheme code

Output Arguments

- pDst – the decrypted plaintext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `dstlen < or = 0`
- `ippStsCFBSizeErr` – wrong value for `cfbBlkSize`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael192BufferSize

Prototype

```
IppStatus ippsRijndael192BufferSize(int *pSize);
```

Description

`ippsRijndael192BufferSize()` specifies the buffer size (in number bytes) required for defining a symbolic structure `IppsRijndael192`. This carries not only a set of round keys and a set of round inverse keys used for various Rijndael192 cipher functions (with various key lengths) for both encryption and decryption, but also the key management information.

Input Arguments

- None

Output Arguments

- `pSize` – the size of required buffer (in bytes) for initializing `IppsRijndael192`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

Rijndael192Init

Prototype

```
IppStatus ippsRijndael192Init(const Ipp8u *pKey, IppsRijndaelKeyLength keylen,
    IppsRijndael192 *pCtx);
```

Description

`ippsRijndael192Init()` initializes the symbolic structure `IppsRijndael192 *pCtx`, using user supplied buffer space pointed by the input `*pCtx`. In addition, `ippsRijndael192Init()` uses the Rijndael key expansion scheme to compute both a set of round keys and a set of round inverse keys for the symbolic structure `IppsRijndael192 *pCtx` with the user supplied key byte stream `Ipp8u *pKey`, and `IppsRijndaelKeyLength keylen`.

Input Arguments

- `pKey` – key byte stream
- `keylen` – length of the input key bit stream
- `pCtx` – pointer to the user supplied buffer

Output Arguments

- `pCtx` – an initialized symbolic structure `IppsRijndael192`.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `keylen != IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256`

Rijndael192EncryptECB

Prototype

```
IppStatus ippsRijndael192EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsRijndael192 *pCtx,
    IppsCPPadding padding);
```

Description

`ippsRijndael192EncryptECB()` is the Rijndael192 encryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round keys `IppsRijndael192 *pCtx` to execute the ECB mode of the Rijndael192 encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`. Padding is with the user-specified scheme `IppsCPPadding padding`. The function returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream.
- `pCtx` – the set of round keys scheduled for Rijndael192 internal iterations
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` – if `srclen` is not divisible by 24 in case `padding==IppsCPPaddingNONE`
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael192DecryptECB

Prototype

```
IppStatus ippsRijndael192DecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    const IppsRijndael192 *pCtx,
    IppsCPPadding padding);
```

Description

`ippsRijndael192DecryptECB()` is the Rijndael192 decryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round inverse keys `IppsRijndael192 *pCtx` to execute the ECB mode of the Rijndael192 decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+23)/24]*24`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen` – the (byte) length of the decrypted plaintext data stream
- `pCtx` – the set of round inverse keys scheduled for Rijndael192 internal iterations
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if dstlen < `or` = `0`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael192EncryptCBC

Prototype

```
IppStatus ippsRijndael192EncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsRijndael192 *pCtx, const Ipp8u *pIV,
    IppsCPPadding padding);
```

Description

`ippsRijndael192EncryptCBC()` is the Rijndael192 encryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsRijndael192 *pCtx` to execute the CBC mode of the Rijndael192 encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`. Padding is with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. The function returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream
- `pCtx` – the set of round keys scheduled for Rijndael192 internal iterations
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` – if `srclen` is not divisible by 24 in case `padding==IppsCPPaddingNONE`
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7`

Rijndael192DecryptCBC

Prototype

```
IppStatus ippsRijndael192DecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    const IppsRijndael192 *pCtx, const Ipp8u *pIV,
    IppsCPPadding padding);
```

Description

`ippsRijndael192DecryptCBC()` is the Rijndael192 decryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round inverse keys `IppsRijndael192 *pCtx` to execute the CBC mode of the Rijndael192 decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+23)/24]*24` with initialization vector `*pIV`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen` – the (byte) length of the decrypted plaintext data stream
- `pCtx` – the set of round inverse keys scheduled for Rijndael192 internal iterations
- `pIV` – initial vector
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if dstlen `<` `or` `=` `0`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme

- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

Rijndael192EncryptCFB

Prototype

```
IppStatus ippsRijndael192EncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    int cfbBlkSize, const IppsRijndael192 *pCtx,
    const Ipp8u *pIV, IppsCPPadding padding);
```

Description

ippsRijndael192EncryptCFB() is the Rijndael192 encryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys IppsRijndael192 *pCtx to execute the CFB mode of the Rijndael192 encryption scheme to the given input message stream *pSrc of variable (byte) length srclen. Padding is with the user-specified scheme IppsCPPadding padding and with initialization vector *pIV. The function returns the ciphertext result (integral number of user specified CFB block size int cfbBlkSize) to Ipp8u *pDst.

Input Arguments

- pSrc – input plain text data stream of variable length
- srclen – the (byte) length of the input data stream
- cfbBlkSize – CFB block size (in unit of bytes)
- pCtx – the set of round keys scheduled for Rijndael192 internal iterations
- pIV – initial vector
- padding – the padding scheme code

Output Arguments

- pDst – the resulting ciphertext data stream

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers

- ippStsLengthErr – if srclen < or = 0
- ippStsUnderRunErr – if srclen is not divisible by cfbBlkSize in case padding == IppsCPPaddingNONE
- ippStsCFBSizeErr – wrong value for cfbBlkSize
- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

Rijndael192DecryptCFB

Prototype

```
IppStatus ippsRijndael192DecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    int cfbBlkSize, const IppsRijndael192 *pCtx,
    const Ipp8u *pIV, IppsCPPadding padding);
```

Description

ippsRijndael192DecryptCFB() is the Rijndael192 decryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round inverse keys IppsRijndael192 *pCtx to execute the CFB mode of the Rijndael192 decryption scheme to the given input ciphertext stream *pSrc of integral number (bytes) of the user specified CFB block size cfbBlkSize calculated as [(dstlen+cfbBlkSize-1)/cfbBlkSize]*cfbBlkSize with initialization vector *pIV, and returns the decrypted plaintext result of user specified message length dstlen to Ipp8u *pDst. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input IppsCPPadding padding.

Input Arguments

- pSrc – input ciphertext data stream
- dstlen – the (byte) length of the decrypted plaintext data stream
- cfbBlkSize – FB block size (in unit of bytes)
- pCtx – the set of round inverse keys scheduled for Rijndael192 internal iterations
- pIV – initial vector
- padding – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if dstlen `<` or `=` `0`
- `ippStsCFBSizeErr` – wrong value for `cfbBlkSize`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael256BufferSize

Prototype

```
IppStatus ippsRijndael256BufferSize(int *pSize);
```

Description

`ippsRijndael256BufferSize()` specifies the buffer size (in number bytes) required for defining a symbolic structure `IppsRijndael256`. This carries not only a set of round keys and a set of round inverse keys used for various Rijndael256 cipher functions (with various key lengths) for both encryption and decryption, but also the key management information.

Input Arguments

- None

Output Arguments

- `pSize` – the size of required buffer (in bytes) for initializing `IppsRijndael256`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

Rijndael256Init

Prototype

```
IppStatus ippsRijndael256Init(const Ipp8u *pKey, IppsRijndaelKeyLength keylen,
    IppsRijndael256 *pCtx);
```

Description

`ippsRijndael256Init()` initializes the symbolic structure `IppsRijndael256 *pCtx`, using user supplied buffer space pointed by the input `*pCtx`. In addition, `ippsRijndael256Init()` uses the Rijndael key expansion scheme to compute both a set of round keys and a set of round inverse keys for the symbolic structure `IppsRijndael256 *pCtx` with the user supplied key byte stream `Ipp8u *pKey`, and `IppsRijndaelKeyLength keylen`.

Input Arguments

- `pKey` – key byte stream
- `keylen` – length of the input key bit stream
- `pCtx` – pointer to the user supplied buffer

Output Arguments

- `pCtx` – an initialized symbolic structure `IppsRijndael256`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `keylen != IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256`

Rijndael256EncryptECB

Prototype

```
IppStatus ippsRijndael256EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsRijndael256 *pCtx,
    IppsCPPadding padding);
```

Description

`ippsRijndael256EncryptECB()` is the Rijndael256 encryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round keys `IppsRijndael256 *pCtx` to execute the ECB mode of the Rijndael256 encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`. `Padding` with the user-specified scheme `IppsCPPadding padding`. The function returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream.
- `pCtx` – the set of round keys scheduled for Rijndael256 internal iterations
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` – if `srclen` is not divisible by 32 in case `padding==IppsCPPaddingNONE`
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael256DecryptECB

Prototype

```
IppStatus ippsRijndael256DecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    const IppsRijndael256 *pCtx,
    IppsCPPadding padding);
```

Description

`ippsRijndael256DecryptECB()` is the Rijndael256 decryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round inverse keys `IppsRijndael256 *pCtx` to execute the ECB mode of the Rijndael256 decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+31)/32]*32`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen` – the (byte) length of the decrypted plaintext data stream
- `pCtx` – the set of round inverse keys scheduled for Rijndael256 internal iterations
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `dstlen < or = 0`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael256EncryptCBC

Prototype

```
IppStatus ippsRijndael256EncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsRijndael256 *pCtx, const Ipp8u *pIV,
    IppsCPPadding padding);
```

Description

`ippsRijndael256EncryptCBC()` is the Rijndael256 encryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsRijndael256 *pCtx` to execute the CBC mode of the Rijndael256 encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`. Padding is with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen` – the (byte) length of the input data stream
- `pCtx` – the set of round keys scheduled for Rijndael256 internal iterations
- `pIV` – initial vector
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `srclen < or = 0`
- `ippStsUnderRunErr` – if `srclen` is not divisible by 32 in case `padding==IppsCPPaddingNONE`
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Rijndael256DecryptCBC

Prototype

```
IppStatus ippsRijndael256DecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    const IppsRijndael256 *pCtx, const Ipp8u *pIV,
    IppsCPPadding padding);
```

Description

`ippsRijndael256DecryptCBC()` is the Rijndael256 decryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round inverse keys `IppsRijndael256 *pCtx` to execute the CBC mode of the Rijndael256 decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+31)/32]*32` with initialization vector `*pIV`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen` – the (byte) length of the decrypted plaintext data stream
- `pCtx` – the set of round inverse keys scheduled for Rijndael256 internal iterations
- `pIV` – initial vector
- `padding` – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `dstlen < or = 0`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme

- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

Rijndael256EncryptCFB

Prototype

```
IppStatus ippsRijndael256EncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    int cfbBlkSize, const IppsRijndael256 *pCtx,
    const Ipp8u *pIV, IppsCPPadding padding);
```

Description

ippsRijndael256EncryptCFB() is the Rijndael256 encryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys IppsRijndael256 *pCtx to execute the CFB mode of the Rijndael256 encryption scheme to the given input message stream *pSrc of variable (byte) length srclen. Padding is with the user specified scheme IppsCPPadding padding and with initialization vector *pIV. The function returns the ciphertext result (integral number of user specified CFB block size int cfbBlkSize) to Ipp8u *pDst.

Input Arguments

- pSrc – input plain text data stream of variable length
- srclen – the (byte) length of the input data stream
- cfbBlkSize – CFB block size (in unit of bytes)
- pCtx – the set of round keys scheduled for Rijndael256 internal iterations
- pIV – initial vector
- padding – the padding scheme code

Output Arguments

- pDst – the resulting ciphertext data stream

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers

- ippStsLengthErr – if srclen < or = 0
- ippStsUnderRunErr – if srclen is not divisible by cfbBlkSize in case padding == IppsCPPaddingNONE
- ippStsCFBSizeErr – wrong value for cfbBlkSize
- ippStsPaddingSchemeErr – if padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7

Rijndael256DecryptCFB

Prototype

```
IppStatus ippsRijndael256DecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int dstlen,
    int cfbBlkSize, const IppsRijndael256 *pCtx,
    const Ipp8u *pIV, IppsCPPadding padding);
```

Description

ippsRijndael256DecryptCFB() is the Rijndael256 decryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round inverse keys IppsRijndael256 *pCtx to execute the CFB mode of the Rijndael256 decryption scheme to the given input ciphertext stream *pSrc of integral number (bytes) of the user specified CFB block size cfbBlkSize calculated as [(dstlen+cfbBlkSize-1)/cfbBlkSize]*cfbBlkSize with initialization vector *pIV, and returns the decrypted plaintext result of user specified message length dstlen to Ipp8u *pDst. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input IppsCPPadding padding.

Input Arguments

- pSrc – input ciphertext data stream
- dstlen – the (byte) length of the decrypted plaintext data stream
- cfbBlkSize – CFB block size (in unit of bytes)
- pCtx – the set of round inverse keys scheduled for Rijndael256 internal iterations
- pIV – initial vector
- padding – the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `dstlen < or = 0`
- `ippStsCFBSizeErr` – wrong value for `cfbBlkSize`
- `ippStsPaddingErr` – the final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` – if `padding != IppsCPPaddingNONE`, `IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`

Blowfish

Blowfish is a16-round Feistel block cipher. The block length is 64 bits and the key can be any size up to 448 bits. Although it requires a computationally intensive key expansion process that creates a set of eighteen 32-bit subkeys plus four 8x32-bit S-boxes derived from the user input key, for a total of 4168 bytes, the actual encryption of streaming data is, however, very efficient for its software implementation on Intel XScale® microprocessor family.

This section describes the primitives available in the Intel® IPP for performing various operational modes under the Blowfish cipher systems. The primitives in this section are implemented in the way to comply to the Blowfish cipher schemes documented in the file <<Description of a New Variable-length Key, 64-Bit Block Cipher (Blowfish)>> submitted by B. Schneier.

Throughout this section, the primitives for Blowfish baseline cipher scheme employ the symbolic data structure `IppsBlowfish`. It is so defined as to serve as the operational vehicles to not only carry both a set of subkeys and a set of S-Boxes, but also the key management information.

Once the set of subkeys and S-Boxes are generated by the respective initialization function, the primitives for ECB, CBC and CFB modes are ready to the execution of either encrypting or decrypting the streaming data with the padding scheme of the user's choice.

The application code for conducting a typical encryption under CBC mode using Blowfish scheme should follow the sequence of operations as outlined below:

Get the buffer size required to configure the symbolic structure `ippsBlowfish` by calling the primitive `ippsBlowfishBuffersize()`.

Call OS memory-allocation service function to allocate a buffer whose size is no less than the one specified by the primitive `ippsBlowfishBuffersize()`. Initialize the symbolic structure `IppsBlowfish *pCtx` by calling the primitive `ippsBlowfishInit()` with the allocated buffer and the respective Blowfish cipher key of the user specified size.

Specify the initialization vector and the padding scheme, than call the primitive `ippsBlowfishEncryptCBC()` to encrypt the input data stream using the Blowfish encryption function with CBC mode.

Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsBlowfish`, if needed.

BlowfishBufferSize

Prototype

```
IppStatus ippsBlowfishBufferSize(int *pSize);
```

Description

`ippsBlowfishBufferSize()` specifies the buffer size (in number bytes) required for defining a symbolic structure `IppsBlowfish` which carries not only a set of round keys and a set of S-Boxes used for various Blowfish cipher functions (with various key lengths) for both encryption and decryption, but also the key management information.

Input Arguments

- None

Output Arguments

- `pSize`– the size of required buffer (in bytes) for initializing `IppsBlowfish`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

BlowfishInit

Prototype

```
IppStatus ippsBlowfishInit(const Ipp8u *pKey, int keylen, IppsBlowfish
   *pCtx);
```

Description

`ippsBlowfishInit()` initializes the symbolic structure `IppsBlowfish *pCtx`, using user supplied buffer space pointed by the input `*pCtx`. In addition, `ippsBlowfishInit()` uses the Blowfish key expansion scheme to compute both a set of round keys and a set of S-Boxes for the symbolic structure `IppsBlowfish *pCtx` with the user supplied key byte stream `Ipp8u *pKey`, and `int keylen`.

- Input Arguments
- `pKey` – key byte stream
- `keylen` – length of the input key byte stream
- `pCtx` – pointer to the user supplied buffer

Output Arguments

- `pCtx` – an initialized symbolic structure `IppsBlowfish`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `keylen < 1` or `keylen > 56`.

BlowfishEncryptECB

Prototype

```
IppStatus ippsBlowfishEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
   srclen, const IppsBlowfish *pCtx, IppsCPPadding padding);
```

Description

`ippsBlowfishEncryptECB()` is the Blowfish encryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round keys `IppsBlowfish *pCtx` to execute the ECB mode of the Blowfish encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`, padding with the user specified scheme `IppsCPPadding padding`. It returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen`– the (byte) length of the input data stream.
- `pCtx`– the set of round keys scheduled for Blowfish internal iterations.
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If `srclen < or = 0`.
- ippStsUnderRunErr - If `srclen` is not divisible by 8 in case `padding==IppsCPPaddingNONE`.
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7`.

BlowfishDecryptECB

Prototype

```
IppStatus ippsBlowfishDecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
   dstlen, const IppsBlowfish *pCtx, IppsCPPadding padding);
```

Description

`ippsBlowfishDecryptECB()` is the Blowfish decryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round keys `IppsBlowfish *pCtx` to execute the ECB mode of the Blowfish decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+7)/8]*8`, and returns the decrypted plaintext result of user specified message length `dstlen to Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen`– the (byte) length of the decrypted plaintext data stream.
- `pCtx`– the set of round inverse keys scheduled for Blowfish internal iterations.
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If dstlen `< or = 0`
- ippStsPaddingErr - The final plaintext data block violate the user specified padding scheme
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

BlowfishEncryptCBC

Prototype

```
IppStatus ippsBlowfishEncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int
   srclen, const IppsBlowfish *pCtx, const Ipp8u *pIV, IppsCPPadding
   padding);
```

Description

`ippsBlowfishEncryptCBC()` is the Blowfish encryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsBlowfish *pCtx` to execute the CBC mode of the Blowfish encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`, padding with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen`– the (byte) length of the input data stream.
- `pCtx`– the set of round keys scheduled for Blowfish internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If `srclen < or = 0.`
- ippStsUnderRunErr - If `srclen` is not divisible by 8 in case `padding==IppsCPPaddingNONE.`
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

BlowfishDecryptCBC

Prototype

```
IppStatus ippsBlowfishDecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int
   dstlen, const IppsBlowfish *pCtx, const Ipp8u *pIV, IppsCPPadding
   padding);
```

Description

`ippsBlowfishDecryptCBC()` is the Blowfish decryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsBlowfish *pCtx` to execute the CBC mode of the Blowfish decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+7)/8]*8` with initialization vector `*pIV`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen`– the (byte) length of the decrypted plaintext data stream.
- `pCtx`– the set of round inverse keys scheduled for Blowfish internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If dstlen `< or = 0`
- ippStsPaddingErr - The final plaintext data block violate the user specified padding scheme
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

BlowfishEncryptCFB

Prototype

```
IppStatus ippsBlowfishEncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int
   srclen, int cfbBlkSize, const IppsBlowfish *pCtx, const Ipp8u *pIV,
   IppsCPPadding padding);
```

Description

`ippsBlowfishEncryptCFB()` is the Blowfish encryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys `IppsBlowfish *pCtx` to execute the CFB mode of the Blowfish encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen,` padding with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of user specified CFB block size `int cfbBlkSize`) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen`– the (byte) length of the input data stream.
- `cfbBlkSize` – CFB block size (in unit of bytes)
- `pCtx`– the set of round keys scheduled for Blowfish internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If `srclen < or = 0.`
- ippStsUnderRunErr - If `srclen` is not divisible by `cfbBlkSize` in case `padding == IppsCPPaddingNONE.`
- ippStsCFBSizeErr - wrong value for `cfbBlkSize`

- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

BlowfishDecryptCFB

Prototype

```
IppStatus ippsBlowfishDecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int
   dstlen, int cfbBlkSize, const IppsBlowfish *pCtx, const Ipp8u *pIV,
   IppsCPPadding padding);
```

Description

`ippsBlowfishDecryptCFB()` is the Blowfish decryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys `IppsBlowfish *pCtx` to execute the CFB mode of the Blowfish decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of the user specified CFB block size `cfbBlkSize` calculated as `[(dstlen+cfbBlkSize-1)/cfbBlkSize]*cfbBlkSize` with initialization vector `*pIV,` and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen`– the (byte) length of the decrypted plaintext data stream.
- `cfbBlkSize` – CFB block size (in unit of bytes)
- `pCtx`– the set of round inverse keys scheduled for Blowfish internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

- ippStsLengthErr – If dstlen `< or = 0`
- ippStsCFBSizeErr - wrong value for `cfbBlkSize`
- ippStsPaddingErr - The final plaintext data block violate the user specified padding scheme
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

Twofish

Twofish is a16-round Feistel block cipher. The block length is 128 bits and the key can be any size up to 256 bits. It is one of the five Advanced Encryption Standard (AES) finalists. Its design for both the round function and key schedule permits an efficient implementation in software. Twofish is a unpatented free for all users and yet efficient and highly secure block cipher.

This section describes the primitives available in theIntel® IPP for performing various operational modes under the Twofish cipher systems. The primitives in this section are implemented in the way to comply to the Twofish cipher schemes documented in the file <<Twofish: A 128-bit block cipher>> submitted to NIST for its candidacy of AES by B. Schneier etc..

Throughout this section, the primitives for Twofish baseline cipher scheme employ the symbolic data structure `IppsTwofish`. It is so defined as to serve as an operational vehicle to not only carry both a set of subkeys and a set of S-Boxes, but also the key management information.

Once the set of subkeys and S-Boxes are generated by the respective initialization function, the primitives for ECB, CBC and CFB modes are ready to the execution of either encrypting or decrypting the streaming data with the padding scheme of the user's choice.

The application code for conducting a typical encryption under CBC mode using Twofish scheme should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsTwofish` by calling the primitive `ippsTwofishBuffersize()`.
2. Call OS memory-allocation service function to allocate a buffer whose size is no less than the one specified by the primitive `ippsTwofishBuffersize()`. Initialize the symbolic structure `IppsBlowfish *pCtx` by calling the primitive `ippsBlowfishInit()` with the allocated buffer and the respective Blowfish cipher key of the user specified size.
3. Specify the initialization vector and the padding scheme, than call the primitive `ippsTwofishEncryptCBC()` to encrypt the input data stream using the Twofish encryption function with CBC mode.
4. Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsTwofish`, if needed

TwofishBufferSize

Prototype

```
IppStatus ippsTwofishBufferSize(int *pSize);
```

Description

`ippsTwofishBufferSize()` specifies the buffer size (in number bytes) required for defining a symbolic structure `IppsTwofish` which carries not only a set of round keys and a set of S-Boxes used for various Twofish cipher functions (with various key lengths) for both encryption and decryption, but also the key management information.

Input Arguments

- None

Output Arguments

- `pSize`– the size of required buffer (in bytes) for initializing `IppsTwofish`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

TwofishInit

Prototype

```
IppStatus ippsTwofishInit(const Ipp8u *pKey, int keylen, IppsTwofish
   *pCtx);
```

Description

ippsTwofishInit() initializes the symbolic structure IppsTwofish *pCtx, using user supplied buffer space pointed by the input *pCtx. In addition, ippsTwofishInit() uses the Twofish key expansion scheme to compute both a set of round keys and a set of S-Boxes for the symbolic structure IppsTwofish *pCtx with the user supplied key byte stream Ipp8u *pKey, and int keylen.

Input Arguments

- pKey – key byte stream
- keylen – length of the input key byte stream
- pCtx – pointer to the user supplied buffer

Output Arguments

- pCtx – an initialized symbolic structure IppsTwofish.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If keylen < 1 or keylen > 32.

TwofishEncryptECB

Prototype

```
IppStatus ippsTwofishEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
   srclen, const IppsTwofish *pCtx, IppsCPPadding padding);
```

Description

ippsTwofishEncryptECB() is the Twofish encryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round keys IppsTwofish *pCtx to execute the ECB mode of the Twofish encryption scheme to the given input message stream *pSrc of variable (byte) length srclen, padding with the user specified scheme IppsCPPadding padding. It returns the ciphertext result (integral number of block length) to Ipp8u *pDst.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen`– the (byte) length of the input data stream.
- `pCtx`– the set of round keys scheduled for Twofish internal iterations.
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If `srclen < or = 0`.
- ippStsUnderRunErr - If `srclen` is not divisible by 16 in case `padding==IppsCPPaddingNONE`.
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7`.

TwofishDecryptECB

Prototype

```
IppStatus ippsTwofishDecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
   dstlen, const IppsTwofish *pCtx, IppsCPPadding padding);
```

Description

`ippsTwofishDecryptECB()` is the Twofish decryption Electronic Code Book (ECB) function. The ECB mode follows FIPS 81. It uses a set of round keys `IppsTwofish *pCtx` to execute the ECB mode of the Twofish decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+15)/16]*16`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen`– the (byte) length of the decrypted plaintext data stream.
- `pCtx`– the set of round inverse keys scheduled for Twofish internal iterations.
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If dstlen `< or = 0`
- ippStsPaddingErr - The final plaintext data block violate the user specified padding scheme
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

TwofishEncryptCBC

Prototype

```
IppStatus ippsTwofishEncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int
   srclen, const IppsTwofish *pCtx, const Ipp8u *pIV, IppsCPPadding
   padding);
```

Description

`ippsTwofishEncryptCBC()` is the Twofish encryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsTwofish *pCtx` to execute the CBC mode of the Twofish encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`, padding with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of block length) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen`– the (byte) length of the input data stream.
- `pCtx`– the set of round keys scheduled for Twofish internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If `srclen < or = 0`.
- ippStsUnderRunErr - If `srclen` is not divisible by 16 in case `padding==IppsCPPaddingNONE`.
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7`.

TwofishDecryptCBC

Prototype

```
IppStatus ippsTwofishDecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int
   dstlen, const IppsTwofish *pCtx, const Ipp8u *pIV, IppsCPPadding
   padding);
```

Description

`ippsTwofishDecryptCBC()` is the Twofish decryption Cipher Block Chaining (CBC) function. The CBC mode follows FIPS 81. It uses a set of round keys `IppsTwofish *pCtx` to execute the CBC mode of the Twofish decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of block length calculated as `[(dstlen+15)/16]*16` with initialization vector `*pIV`, and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen`– the (byte) length of the decrypted plaintext data stream.
- `pCtx`– the set of round inverse keys scheduled for Twofish internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- `ippStsNoErr` – No Error
- `ippStsNullPtrErr` – Null pointers
- `ippStsLengthErr` – If dstlen `< or = 0`
- `ippStsPaddingErr` - The final plaintext data block violate the user specified padding scheme
- `ippStsPaddingSchemeErr` - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS` or `IppsCPPaddingPKCS7`.

TwofishEncryptCFB

Prototype

```
IppStatus ippsTwofishEncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int
   srclen, int cfbBlkSize, const IppsTwofish *pCtx, const Ipp8u *pIV,
   IppsCPPadding padding);
```

Description

`ippsTwofishEncryptCFB()` is the Twofish encryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys `IppsTwofish *pCtx` to execute the CFB mode of the Twofish encryption scheme to the given input message stream `*pSrc` of variable (byte) length `srclen`, padding with the user specified scheme `IppsCPPadding padding` and with initialization vector `*pIV`. It returns the ciphertext result (integral number of user specified CFB block size `int cfbBlkSize`) to `Ipp8u *pDst`.

Input Arguments

- `pSrc` – input plain text data stream of variable length
- `srclen`– the (byte) length of the input data stream.
- `cfbBlkSize` – CFB block size (in unit of bytes)
- `pCtx`– the set of round keys scheduled for Twofish internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the resulting ciphertext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If `srclen < or = 0.`
- ippStsUnderRunErr - If `srclen` is not divisible by `cfbBlkSize` in case `padding == IppsCPPaddingNONE.`
- ippStsCFBSizeErr - wrong value for `cfbBlkSize`
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

TwofishDecryptCFB

Prototype

```
IppStatus ippsTwofishDecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int
   dstlen, int cfbBlkSize, const IppsTwofish *pCtx, const Ipp8u *pIV,
   IppsCPPadding padding);
```

Description

`ippsTwofishDecryptCFB()` is the Twofish decryption Cipher Feedback (CFB) function. The CFB mode follows FIPS 81. It uses a set of round keys `IppsTwofish *pCtx` to execute the CFB mode of the Twofish decryption scheme to the given input ciphertext stream `*pSrc` of integral number (bytes) of the user specified CFB block size `cfbBlkSize` calculated as

`[(dstlen+cfbBlkSize-1)/cfbBlkSize]*cfbBlkSize` with initialization vector `*pIV,` and returns the decrypted plaintext result of user specified message length `dstlen` to `Ipp8u *pDst`. Furthermore, this function validates the final decrypted plaintext block to see if it matches the padding scheme specified by the user input `IppsCPPadding padding`.

Input Arguments

- `pSrc` – input ciphertext data stream
- `dstlen`– the (byte) length of the decrypted plaintext data stream.
- `cfbBlkSize` – CFB block size (in unit of bytes)
- `pCtx`– the set of round inverse keys scheduled for Twofish internal iterations.
- `pIV` – initial vector
- `padding`– the padding scheme code

Output Arguments

- `pDst` – the decrypted plaintext data stream.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If dstlen `< or = 0`
- ippStsCFBSizeErr - wrong value for `cfbBlkSize`
- ippStsPaddingErr - The final plaintext data block violate the user specified padding scheme
- ippStsPaddingSchemeErr - If `padding != IppsCPPaddingNONE, IppsCPPaddingZEROS or IppsCPPaddingPKCS7.`

Data Authentication Algorithm Primitives

This section describes the Intel® IPP - Cryptographic Primitives for the PCA processors with MMX™ that are available for generating the data authentication code using the Data Authentication Algorithm (DAA) specified in the FIPS 113.

DAA has been widely used in the applications to detect unauthorized modifications, both intentional and accidental, to data. In this release, Intel® IPP offers primitives that are designed for applications to use various DAA schemes based on a set of block ciphers described in the "Block Cipher Primitives" section. The related standards that the primitives in this section conform to are listed as the following:

- The DAA scheme, specified in FIPS 113
- The CBC mode, specified in FIPS 81

- The DES/TDES block ciphers, specified in FIPS 46
- The AES/Rijndael block ciphers, specified in FIPS 197
- Blowfish block cipher
- Twofish block cipher

This section describes the following primitives:

- "DAA-DES/TDES"
- "DAA-Rijndael128/192/256"
- "DAA-Blowfish"
- "DAA-Twofish"

DAA-DES/TDES

DAA-DES and DAA-TDES are two data authentication algorithms with respect to their baseline block ciphers DES and Triple-DES as described in "DES/TDES". Both DAA-DES and DAA-TDES use their respective encryption key to transform the input message of any length into a data authentication code of the user specified length (no more than 8 bytes), while DAA-DES takes a DES key, and DAA-TDES takes a set of three DES keys.

This section describes two sets of primitives that completes the operations required for generating the data authentication code using DAA-DES and DAA-TDES. All of the primitives described in this section are implemented to comply with FIPS 113, FIPS 81 and FIPS 46.

To use DAA-DES (same description applies to DAA-TDES) to generate a data authentication code, the application should use a set of five DAA-DES primitives. The `ippsDAADESInit()` sets up an initial chaining state and computes a set of DES round keys derived from the user supplied key. Once initialized, the primitive function `ippsDAADESUpdate()` executes a DES encryption scheme under CBC mode to the input message stream till it exhausts all the DES data blocks. The `ippsDAADESFinal()` is designed to pad the possible final partial DES data block with the `IppCPPaddingZEROS` specified padding scheme, it then further executes the DES encryption scheme under CBC mode to the final DES data block into a `64`-bit DES ciphertext that yields the data authentication code `Ipp8u *pDAC` of the user specified length `daclen` (no more than 8 bytes).

This Intel® IPP also provides a primitive `ippsDAADESMessageDigest()` to simplify the calling convention for generating the data authentication code with respect to the input message (non streaming data) as outlined above. This primitive uses the user supplied DES key and applies the DAA-DES scheme to transform the input message into a data authentication code `Ipp8u *pDAC` of the user specified length `daclen` (no more than 8 bytes).

Throughout this section, all the DAA-DES primitives employ a symbolic structure `ippsDAADES`, and all the DAA-TDES primitives employ a symbolic structure `ippsDAATDES`, both serve as operational vehicles which carry the information necessary to the operation, such as the set of round keys derived from the user input key, a partial message block, and the intermediate state values.

The application code for applying DAA-DES (similar procedure applies to DAA-TDES) to transform the input message into a data authentication code should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsDAADES` by calling the primitive `ippsDAADESBuffersize()`.
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive `ippsDAADESBuffersize()`. With the allocated buffer, call the primitive `ippsDAADESInit()` to setup the initial chaining state and derives a set DES round keys.
3. Keep calling the primitive `ippsDAADESUpdate()` to digest incoming message stream in the queue till its completion.
4. Call the primitive `ippsDAADESFinal()` for the DAA-DES final processing step as to compute the desired data authentication code `pDAC`.
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsDAADES`, if needed.

DAADESBufferSize

Prototype

```
IppStatus ippsDAADESBufferSize(int *pSize);
```

Description

`ippsDAADESBufferSize()` specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure `IppsDAADES` to carry the information needed for generating an (up to `64` bits) data authentication code of the given input message.

Input Arguments

- `None`

Output Arguments

- `pSize`– the size of required buffer (in bytes) for `IppsDAADES`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAADESInit

Prototype

```
IppStatus ippsDAADESInit(const Ipp8u *pKey, IppsDAADES *pState);
```

Description

`ippsDAADESInit()` initializes the symbolic structure `IppsDAADES *pState`, using user supplied buffer space pointed by the input `*pState`. In addition, `ippsDAADESInit()` sets up the initial chaining state `IppsDAADES *pState`, computes a set of DES round keys derived from the user supplied key `*pKey`.

Input Arguments

- `pKey` – the input DES key
- `pState` – pointer to the user supplied buffer.

Output Arguments

- `pState` – the initialized symbolic structure `IppsDAADES`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAADESUpdate

Prototype

```
IppStatus ippsDAADESUpdate(const Ipp8u *pSrcMesg, int mesglen,
   IppsDAADES *pState);
```

Description

`ippsDAADESUpdate()` uses the DAA-DES scheme to digest the current input message stream `*pSrcMesg` whose (byte) length is specified by the input `mesglen`.

`ippsDAADESUpdate()` starts with integrating the previous partial DES data block carried by the input state variable `IppsDAADES *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple DES data blocks (each is `64`-bit) with a possible additional partial block. This function then uses the chaining state value carried by `*pState` as an initialization vector, and executes the DES encryption scheme under CBC mode to each DES data block consecutively.

Upon the completion, `ippsDAADESUpdate()` updates the state variable `IppsDAADES *pState` with the newly generated chaining state and a possible partial DES data block for further message integration or padding.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

DAADESFinal

Prototype

```
IppStatus ippsDAADESFinal(Ipp8u *pDAC, int daclen, IppsDAADES *pState);
```

Description

`ippsDAADESFinal()` applies DAA-DES scheme to further transform the final chaining state with a possible remaining partial DES data block both carried by the input `IppsDAADES *pState` into a `64`-bit DES ciphertext which yields the desired data authentication code of the user specified length `daclen` (no more than `8` bytes).

Input Arguments

- `daclen` – the user specified message authentication code length.
- `pState` – the input chaining state

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either `daclen < 1` or `daclen > 8`.

DAADESMessageDigest

Prototype

```
IppStatus ippsDAADESMessageDigest(const Ipp8u *pSrcMesg, int mesglen,
   const Ipp8u *pKey, Ipp8u *pDAC, int daclen);
```

Description

`ippsDAADESMessageDigest()` computes a set of DES round keys derived from the input DES key and applies the DAA-DES scheme to transform the input message `*pSrcMesg` whose (byte) length is specified by input `mesglen` into its respective data authentication code `*pDAC` of the user specified length `daclen` (no more than 8 bytes).

The following pseudocode represents this function:

pDAC = $\text{DESEncrypt}_{\text{pKey}}$-CBC([pSrcMesg||IppsCPPaddingZEROS], pIV),

where `pIV == 0`, $\texttt{DESEncrypt}_{\texttt{pKey}}\texttt{-CBC}$ represents a function that computes the last ciphertext block using the DES encryption algorithm under CBC mode with input key `pKey`, and `||` is the operation for concatenating two bit-streams.

Input Arguments

- `pSrcMesg` – input message stream
- `mesglen` – the (byte) length of message stream
- `pKey` - input DES key
- `daclen` – the user desired data authentication code length (in unit of bytes)

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) `mesglen < 0,` or (2) `daclen < 1 or daclen > 8.`

DAATDESBufferSize

Prototype

```
IppStatus ippsDAATDESBufferSize(int *pSize);
```

Description

ippsDAATDESBufferSize() specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure IppsDAATDES to carry the information needed for generating an (up to 64 bits) data authentication code of the given input message.

Input Arguments

- None

Output Arguments

- pSize– the size of required buffer (in bytes) for IppsDAATDES.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAATDESInit

Prototype

```
IppStatus ippsDAATDESInit(const Ipp8u *pKey1, const Ipp8u *pKey2, const
   Ipp8u *pKey3, IppsDAATDES *pState);
```

Description

ippsDAATDESInit() initializes the symbolic structure IppsDAATDES *pState, using user supplied buffer space pointed by the input *pState. In addition, ippsDAATDESInit() sets up the initial chaining state IppsDAATDES *pState, computes three sets of DES round keys derived from the user supplied a set of three DES keys *pKey1, *pKey2 and *pKey3.

Input Arguments

- pKey1 – the input DES key1
- pKey2 – the input DES key2
- pKey3 – the input DES key3
- pState – pointer to the user supplied buffer.

Output Arguments

- `pState` – the initialized symbolic structure `IppsDAATDES`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAATDESUpdate

Prototype

```
IppStatus ippsDAATDESUpdate(const Ipp8u *pSrcMesg, int mesglen,
   IppsDAATDES *pState);
```

Description

`ippsDAATDESUpdate()` uses the DAA-TDES scheme to digest the current input message stream `*pSrcMesg` whose (byte) length is specified by the input `mesglen`.

`ippsDAATDESUpdate()` starts with integrating the previous partial DES data block carried by the input state variable `IppsDAATDES *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple DES data blocks (each is `64`-bit) with a possible additional partial block. This function then uses the chaining state value carried by `*pState` as an initialization vector, and executes the TDES encryption scheme under CBC mode to each DES data block consecutively.

Upon the completion, `ippsDAATDESUpdate()` updates the state variable `IppsDAATDES *pState` with the newly generated chaining state and a possible partial DES data block for further message integration or padding.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

DAATDESFinal

Prototype

```
IppStatus ippsDAATDESFinal(Ipp8u *pDAC, int daclen, IppsDAATDES
   *pState);
```

Description

`ippsDAATDESFinal()` applies DAA-TDES scheme to further transform the final chaining state with a possible remaining partial DES data block both carried by the input `IppsDAATDES *pState` into a `64`-bit DES ciphertext which yields the desired data authentication code of the user specified length `daclen` (no more than `8` bytes).

Input Arguments

- `daclen` – the user specified message authentication code length.
- `pState` – the input chaining state

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either `daclen < 1` or `daclen > 8`.

DAATDESMessageDigest

Prototype

```
IppStatus ippsDAATDESMessageDigest(const Ipp8u *pSrcMesg, int mesglen,
   const Ipp8u *pKey1, const Ipp8u *pKey2, const Ipp8u *pKey3, Ipp8u
   *pDAC, int daclen);
```

Description

`ippsDAATDESMessageDigest()` computes three sets of DES round keys derived from the input DES keys (`*pKey1, *pKey2` and `*pKey3`) and applies the DAA-TDES scheme to transform the input message `*pSrcMesg` whose (byte) length is specified by input `mesglen` into its respective data authentication code `*pDAC` of the user specified length `daclen` (no more than 8 bytes).

The following pseudocode represents this function:

pDAC = $\text{TDESEncrypt}_{[pKey1,pKey2,pKey3]}$-CBC([pSrcMesg||IppsCPPaddingZEROS], pIV),

where `pIV == 0`, $\text{TDESEncrypt}_{[pKey1,pKey2,pKey3]}$-CBC represents a function that computes the last ciphertext block using the TDES encryption algorithm under CBC mode with input key set [`pKey1`, `pKey2`, `pKey3`] and `||` is the operation that concatenates two bit-streams.

Input Arguments

- `pSrcMesg` – input message stream
- `mesglen` – the (byte) length of message stream
- `pKey1` - input DES key1
- `pKey2` - input DES key2
- `pKey3` - input DES key3
- `daclen` – the user desired data authentication code length (in unit of bytes)

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

- ippStsLengthErr - If any of the following occurs: (1) `mesglen < 0,` or (2) `daclen < 1 or daclen > 8.`

DAA-Rijndael128/192/256

DAA-Rijndael128, DAA-Rijndael192 and DAA-Rijndael256 are three data authentication algorithms with respect to their baseline block ciphers Rijndael128, Rijndael192, and Rijndael256 as described in Block Cipher Primitives. All three schemes use their respective encryption key to transform the input message of any length into a data authentication code of the user specified length (no more than 16, 24, and 32 bytes corresponding to DAA-Rijndael128, DAA-Rijndael192, as well as DAA-Rijndael256).

This section describes three sets of primitives that complete the operations required for generating the data authentication code using DAA-Rijndael128, DAA-Rijndael192 and DAA-Rijndael256. All of the primitives described in this section are implemented to comply with FIPS 113, FIPS 81 and FIPS 197.

To use DAA-Rijndael128 (same description applies to DAA-Rijndael192 and DAA-Rijndael256) to generate a data authentication code, the application should use a set of five DAA-Rijndael128 primitives. The `ippsDAARijndael128Init()` sets up an initial chaining state and computes a set of Rijndael128 round keys derived from the user supplied key. Once initialized, the primitive function `ippsDAARijndael128Update()` executes a Rijndael128 encryption scheme under CBC mode to the input message stream till it exhausts all the Rijndael128 data blocks. The primitive function `ippsDAARijndael128Final()` is designed to pad the possible final partial Rijndael128 data block with the `IppCPPaddingZEROS` specified padding scheme, it then further executes the Rijndael128 encryption scheme under CBC mode to the final Rijndael128 data block into an `128`-bit Rijndael128 ciphertext that yields the data authentication code `Ipp8u *pDAC` of the user specified length `daclen` (no more than `16` bytes).

This Intel® IPP also provides a primitive `ippsDAARijndael128MessageDigest()` to simplify the calling convention for generating the data authentication code with respect to the input message (non streaming data) as outlined above. This primitive uses the user supplied Rijndael128 key and applies the DAA-Rijndael128 scheme to transform the input message into a data authentication code `Ipp8u *pDAC` of the user specified length `daclen` (no more than `16` bytes).

Throughout this section, all the DAA-Rijndael128 primitives employ a symbolic structure `ippsDAARijndael128`, all the DAA-Rijndael192 primitives employ a symbolic structure `ippsDAARijndael192`, and all the DAA-Rijndael256 primitives employ a symbolic structure `ippsDAARijndael256`. They all serve as operational vehicles which carry the information necessary to the operation, such as the set of round keys derived from the user input key, a partial message block, and the intermediate state values.

The application code for applying DAA-Rijndael128 (similar procedure applies to both DAA-Rijndael192, and DAA-Rijndael256) to transform the input message into a data authentication code should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsDAARijndael128` by calling the primitive `ippsDAARijndael128Buffersize()`.
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive `ippsDAARijndael128Buffersize()`. With the allocated buffer, call the primitive `ippsDAARijndael128Init()` to setup the initial chaining state and derives a set Rijndael128 round keys.
3. Keep calling the primitive `ippsDAARijndael128Update()` to digest incoming message stream in the queue till its completion.
4. Call the primitive `ippsDAARijndael128Final()` for the DAA-Rijndael128 final processing step as to compute the desired data authentication code.
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsDAARijndael128`, if needed.

DAARijndael128BufferSize

Prototype

```
IppStatus ippsDAARijndael128BufferSize(int *pSize);
```

Description

`ippsDAARijndael128BufferSize()` specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure `IppsDAARijndael128` to carry the information needed for generating an (up to `128` bits) data authentication code of the given input message.

Input Arguments

- `None`

Output Arguments

- `pSize`– the size of required buffer (in bytes) for `IppsDAARijndael128`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAARijndael128Init

Prototype

```
IppStatus ippsDAARijndael128Init(const Ipp8u *pKey,
   IppsRijndaelKeyLength keylen, IppsDAARijndael128 *pState);
```

Description

`ippsDAARijndael128Init()` initializes the symbolic structure `IppsDAARijndael128 *pState`, using user supplied buffer space pointed by the input `*pState`. In addition, this function sets up the initial chaining state `IppsDAARijndael128 *pState`, computes a set of Rijndael128 round keys derived from the user supplied key `*pKey`.

Input Arguments

- `pKey` – the input Rijndael128 key
- `pState` – pointer to the user supplied buffer.

Output Arguments

- `pState` – the initialized symbolic structure `IppsDAARijndael128`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `keylen != IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256`.

DAARijndael128Update

Prototype

```
IppStatus ippsDAARijndael128Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsDAARijndael128 *pState);
```

Description

`ippsDAARijndael128Update()` uses the DAA-Rijndael128 scheme to digest the current input message stream `*pSrcMesg` whose (byte) length is specified by the input `mesglen`.

`ippsDAARijndael128Update()` starts with integrating the previous partial Rijndael128 data block carried by the input state variable `IppsDAARijndael128 *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple Rijndael128 data blocks (each is `128`-bit) with a possible additional partial block. This function then uses the chaining state value carried by `*pState` as an initialization vector, and executes the Rijndael128 encryption scheme under CBC mode to each Rijndael128 data block consecutively.

Upon the completion, `ippsDAARijndael128Update()` updates the state variable `IppsDAARijndael128 *pState` with the newly generated chaining state and a possible partial Rijndael128 data block for further message integration or padding.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

DAARijndael128Final

Prototype

```
IppStatus ippsDAARijndael128Final(Ipp8u *pDAC, int daclen,
   IppsDAARijndael128 *pState);
```

Description

`ippsDAARijndael128Final()` applies DAA-Rijndael128 scheme to further transform the final chaining state with a possible remaining partial Rijndael128 data block both carried by the input `IppsDAARijndael128 *pState` into an `128`-bit Rijndael128 ciphertext which yields the desired data authentication code of the user specified length `daclen` (no more than `16` bytes).

Input Arguments

- `daclen` – the user specified message authentication code length.
- `pState` – the input chaining state

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either `daclen < 1` or `daclen > 16`.

DAARijndael128MessageDigest

Prototype

```
IppStatus ippsDAARijndael128MessageDigest(const Ipp8u *pSrcMesg, int
   mesglen, const Ipp8u *pKey, IppsRijndaelKeyLength keylen, Ipp8u
   *pDAC, int daclen);
```

Description

`ippsDAARijndael128MessageDigest()` computes a set of Rijndael128 round keys derived from the input Rijndael128 key `*pKey` of the user specified key length `keylen` and applies the DAA-Rijndael128 scheme to transform the input message `*pSrcMesg` whose (byte) length is specified by input `mesglen` into its respective data authentication code `*pDAC` of the user specified length `daclen` (no more than 16 bytes).

The following pseudocode represents this function:

pDAC = Rijndael128Encrypt$_{pKey}$-CBC([pSrcMesg||IppsCPPaddingZEROS], pIV),

where `pIV == 0`, `Rijndael128Encrypt`$_{pKey}$`-CBC` represents a function that computes the last ciphertext block using the Rijndael128 encryption algorithm under CBC mode with input key `pKey`, and `||` is the operation for concatenating two bit-streams.

Input Arguments

- `pSrcMesg` – input message stream
- `mesglen` – the (byte) length of message stream
- `pKey` - input Rijndael128 key
- `daclen` – the user desired data authentication code length (in unit of bytes)

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) `mesglen < 0,` (2) `keylen != IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256` or (3) `daclen < 1 or daclen > 16.`

DAARijndael192BufferSize

Prototype

```
IppStatus ippsDAARijndael192BufferSize(int *pSize);
```

Description

`ippsDAARijndael192BufferSize()` specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure `IppsDAARijndael192` to carry the information needed for generating an (up to `192` bits) data authentication code of the given input message.

Input Arguments

- `None`

Output Arguments

- `pSize`– the size of required buffer (in bytes) for `IppsDAARijndael192`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAARijndael192Init

Prototype

```
IppStatus ippsDAARijndael192Init(const Ipp8u *pKey,
   IppsRijndaelKeyLength keylen, IppsDAARijndael192 *pState);
```

Description

`ippsDAARijndael192Init()` initializes the symbolic structure `IppsDAARijndael192 *pState`, using user supplied buffer space pointed by the input `*pState`. In addition, this function sets up the initial chaining state `IppsDAARijndael192 *pState`, computes a set of Rijndael192 round keys derived from the user supplied key `*pKey`.

Input Arguments

- `pKey` – the input Rijndael192 key
- `pState` – pointer to the user supplied buffer.

Output Arguments

- `pState` – the initialized symbolic structure `IppsDAARijndael192`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `keylen != IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256`.

DAARijndael192Update

Prototype

```
IppStatus ippsDAARijndael192Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsDAARijndael192 *pState);
```

Description

`ippsDAARijndael192Update()` uses the DAA-Rijndael192 scheme to digest the current input message stream `*pSrcMesg` whose (byte) length is specified by the input `mesglen`.

`ippsDAARijndael192Update()` starts with integrating the previous partial Rijndael192 data block carried by the input state variable `IppsDAARijndael192 *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple Rijndael192 data blocks (each is `192`-bit) with a possible additional partial block. This function then uses the chaining state value carried by `*pState` as an initialization vector, and executes the Rijndael192 encryption scheme under CBC mode to each Rijndael192 data block consecutively.

Upon the completion, `ippsDAARijndael192Update()` updates the state variable `IppsDAARijndael192 *pState` with the newly generated chaining state and a possible partial Rijndael192 data block for further message integration or padding.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

DAARijndael192Final

Prototype

```
IppStatus ippsDAARijndael192Final(Ipp8u *pDAC, int daclen,
   IppsDAARijndael192 *pState);
```

Description

`ippsDAARijndael192Final()` applies DAA-Rijndael192 scheme to further transform the final chaining state with a possible remaining partial Rijndael192 data block both carried by the input `IppsDAARijndael192 *pState` into an `192`-bit Rijndael192 ciphertext which yields the desired data authentication code of the user specified length `daclen` (no more than `24` bytes).

Input Arguments

- `daclen` – the user specified message authentication code length.
- `pState` – the input chaining state

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either `daclen < 1` or `daclen > 24`.

DAARijndael192MessageDigest

Prototype

```
IppStatus ippsDAARijndael192MessageDigest(const Ipp8u *pSrcMesg, int
   mesglen, const Ipp8u *pKey, IppsRijndaelKeyLength keylen, Ipp8u
   *pDAC, int daclen);
```

Description

`ippsDAARijndael192MessageDigest()` computes a set of Rijndael192 round keys derived from the input Rijndael192 key `*pKey` of the user specified key length `keylen` and applies the DAA-Rijndael192 scheme to transform the input message `*pSrcMesg` whose (byte) length is specified by input `mesglen` into its respective data authentication code `*pDAC` of the user specified length `daclen` (no more than 24 bytes).

The following pseudocode represents this function:

- pDAC = Rijndael192Encrypt$_{pKey}$-CBC([pSrcMesg||IppsCPPaddingZEROS], pIV),

where `pIV == 0`, `Rijndael192Encrypt`$_{pKey}$`-CBC` represents a function that computes the last ciphertext block using the Rijndael192 encryption algorithm under CBC mode with input key `pKey`, and `||` is the operation for concatenating two bit-streams.

Input Arguments

- `pSrcMesg` – input message stream
- `mesglen` – the (byte) length of message stream
- `pKey` - input Rijndael192 key
- `daclen` – the user desired data authentication code length (in unit of bytes)

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) `mesglen < 0,` (2) `keylen != IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256` or (3) `daclen < 1 or daclen > 24.`

DAARijndael256BufferSize

Prototype

```
IppStatus ippsDAARijndael256BufferSize(int *pSize);
```

Description

`ippsDAARijndael256BufferSize()` specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure `IppsDAARijndael256` to carry the information needed for generating an (up to `256` bits) data authentication code of the given input message.

Input Arguments

- `None`

Output Arguments

- `pSize`– the size of required buffer (in bytes) for `IppsDAARijndael256`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAARijndael256Init

Prototype

```
IppStatus ippsDAARijndael256Init(const Ipp8u *pKey,
   IppsRijndaelKeyLength keylen, IppsDAARijndael256 *pState);
```

Description

ippsDAARijndael256Init() initializes the symbolic structure IppsDAARijndael256 *pState, using user supplied buffer space pointed by the input *pState. In addition, this function sets up the initial chaining state IppsDAARijndael256 *pState, computes a set of Rijndael256 round keys derived from the user supplied key *pKey.

Input Arguments

- pKey – the input Rijndael256 key
- pState – pointer to the user supplied buffer.

Output Arguments

- pState – the initialized symbolic structure IppsDAARijndael256.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If keylen != IppsRijndaelKey128, IppsRijndaelKey192 or IppsRijndaelKey256.

DAARijndael256Update

Prototype

```
IppStatus ippsDAARijndael256Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsDAARijndael256 *pState);
```

Description

ippsDAARijndael256Update() uses the DAA-Rijndael256 scheme to digest the current input message stream *pSrcMesg whose (byte) length is specified by the input mesglen.

ippsDAARijndael256Update() starts with integrating the previous partial Rijndael256 data block carried by the input state variable IppsDAARijndael256 *pState with the input message stream *pSrcMesg and it then partitions them into multiple Rijndael256 data blocks (each is 256-bit) with a

possible additional partial block. This function then uses the chaining state value carried by `*pState` as an initialization vector, and executes the Rijndael256 encryption scheme under CBC mode to each Rijndael256 data block consecutively.

Upon the completion, `ippsDAARijndael256Update()` updates the state variable `IppsDAARijndael256 *pState` with the newly generated chaining state and a possible partial Rijndael256 data block for further message integration or padding.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

DAARijndael256Final

Prototype

```
IppStatus ippsDAARijndael256Final(Ipp8u *pDAC, int daclen,
   IppsDAARijndael256 *pState);
```

Description

`ippsDAARijndael256Final()` applies DAA-Rijndael256 scheme to further transform the final chaining state with a possible remaining partial Rijndael256 data block both carried by the input `IppsDAARijndael256 *pState` into a `256`-bit Rijndael256 ciphertext which yields the desired data authentication code of the user specified length `daclen` (no more than `32` bytes).

Input Arguments

- `daclen` – the user specified message authentication code length.
- `pState` – the input chaining state

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either `daclen < 1` or `daclen > 32.`

DAARijndael256MessageDigest

Prototype

```
IppStatus ippsDAARijndael256MessageDigest(const Ipp8u *pSrcMesg, int
   mesglen, const Ipp8u *pKey, IppsRijndaelKeyLength keylen, Ipp8u
   *pDAC, int daclen);
```

Description

`ippsDAARijndael256MessageDigest()` computes a set of Rijndael256 round keys derived from the input Rijndael256 key `*pKey` of the user specified key length `keylen` and applies the DAA-Rijndael256 scheme to transform the input message `*pSrcMesg` whose (byte) length is specified by input `mesglen` into its respective data authentication code `*pDAC` of the user specified length `daclen` (no more than 32 bytes).

The following pseudocode represents this function:

- pDAC = Rijndael256Encrypt$_{pKey}$-CBC([pSrcMesg||IppsCPPaddingZEROS], pIV),

where `pIV == 0`, `Rijndael256Encrypt`$_{pKey}$`-CBC` represents a function that computes the last ciphertext block using the Rijndael256 encryption algorithm under CBC mode with input key `pKey`, and `||` is the operation for concatenating two bit-streams.

Input Arguments

- `pSrcMesg` – input message stream

- `mesglen` – the (byte) length of message stream
- `pKey` - input Rijndael256 key
- `daclen` – the user desired data authentication code length (in unit of bytes)

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) `mesglen < 0`, (2) `keylen != IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256` or (3) `daclen < 1 or daclen > 32`.

DAA-Blowfish

DAA-Blowfish is a Blowfish block cipher based data authentication algorithm. All the Blowfish block cipher primitives are described in "Blowfish". DAA-Blowfish uses its respective encryption key to transform the input message of any length into a data authentication code of the user specified length (no more than 8 bytes).

This section describes a set of primitives that completes the operations required for generating the data authentication code using DAA-Blowfish. All of the primitives described in this section are implemented to comply with FIPS 113 and FIPS 81.

To use DAA-Blowfish to generate a data authentication code, the application should use a set of five DAA-Blowfish primitives. The `ippsDAABlowfishInit()` sets up an initial chaining state and computes a set of Blowfish round keys derived from the user supplied key. Once initialized, the primitive function `ippsDAABlowfishUpdate()` executes a Blowfish encryption scheme under CBC mode to the input message stream till it exhausts all the Blowfish data blocks. The primitive `ippsDAABlowfishFinal()` is designed to pad the possible final partial Blowfish data block with the `IppCPPaddingZEROS` specified padding scheme, it then further executes the Blowfish encryption scheme under CBC mode to the final Blowfish data block into a `64`-bit Blowfish ciphertext that yields the data authentication code `Ipp8u *pDAC` of the user specified length `daclen` (no more than 8 bytes).

This Intel® IPP also provides a primitive `ippsDAABlowfishMessageDigest()` to simplify the calling convention for generating the data authentication code with respect to the input message (non streaming data) as outlined above. This primitive uses the user supplied Blowfish key and applies the DAA-Blowfish scheme to transform the input message into a data authentication code `Ipp8u *pDAC` of the user specified length `daclen` (no more than 8 bytes).

Throughout this section, all the DAA-Blowfish primitives employ a symbolic structure `ippsDAABlowfish`, to serve as an operational vehicle which carries the information necessary to the operation, such as the set of round keys derived from the user input key, a partial message block, and the intermediate state values.

The application code for applying DAA-Blowfish to transform the input message into a data authentication code should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsDAABlowfish` by calling the primitive `ippsDAABlowfishBuffersize()`.
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive `ippsDAABlowfishBuffersize()`. With the allocated buffer, call the primitive `ippsDAABlowfishInit()` to setup the initial chaining state and derives a set Blowfish round keys.
3. Keep calling the primitive `ippsDAABlowfishUpdate()` to digest incoming message stream in the queue till its completion.
4. Call the primitive `ippsDAABlowfishFinal()` for the DAA-Blowfish final processing step as to compute the desired data authentication code.
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsDAABlowfish`, if needed.

DAABlowfishBufferSize

Prototype

```
IppStatus ippsDAABlowfishBufferSize(int *pSize);
```

Description

`ippsDAABlowfishBufferSize()` specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure `IppsDAABlowfish` to carry the information needed for generating an (up to `64` bits) data authentication code of the given input message.

Input Arguments

- None

Output Arguments

- `pSize`– the size of required buffer (in bytes) for `IppsDAABlowfish`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAABlowfishInit

Prototype

```
IppStatus ippsDAABlowfishInit(const Ipp8u *pKey, int keylen,
   IppsDAABlowfish *pState);
```

Description

`ippsDAABlowfishInit()` initializes the symbolic structure `IppsDAABlowfish *pState`, using user supplied buffer space pointed by the input `*pState`. In addition, `ippsDAABlowfishInit()` sets up the initial chaining state `IppsDAABlowfish *pState`, computes a set of Blowfish round keys derived from the user supplied key `*pKey`.

Input Arguments

- `pKey` – the input Blowfish key
- `pState` – pointer to the user supplied buffer.

Output Arguments

- `pState` – the initialized symbolic structure `IppsDAABlowfish`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `keylen < 0` or `keylen > 56`.

DAABlowfishUpdate

Prototype

```
IppStatus ippsDAABlowfishUpdate(const Ipp8u *pSrcMesg, int mesglen,
   IppsDAABlowfish *pState);
```

Description

`ippsDAABlowfishUpdate()` uses the DAA-Blowfish scheme to digest the current input message stream `*pSrcMesg` whose (byte) length is specified by the input `mesglen`.

`ippsDAABlowfishUpdate()` starts with integrating the previous partial Blowfish data block carried by the input state variable `IppsDAABlowfish *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple Blowfish data blocks (each is `64`-bit) with a possible additional partial block. This function then uses the chaining state value carried by `*pState` as an initialization vector, and executes the Blowfish encryption scheme under CBC mode to each Blowfish data block consecutively.

Upon the completion, `ippsDAABlowfishUpdate()` updates the state variable `IppsDAABlowfish *pState` with the newly generated chaining state and a possible partial Blowfish data block for further message integration or padding.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

DAABlowfishFinal

Prototype

```
IppStatus ippsDAABlowfishFinal(Ipp8u *pDAC, int daclen, IppsDAABlowfish
   *pState);
```

Description

`ippsDAABlowfishFinal()` applies DAA-Blowfish scheme to further transform the final chaining state with a possible remaining partial Blowfish data block both carried by the input `IppsDAABlowfish *pState` into a `64`-bit Blowfish ciphertext which yields the desired data authentication code of the user specified length `daclen` (no more than `8` bytes).

Input Arguments

- `daclen` – the user specified message authentication code length.
- `pState` – the input chaining state

Output Arguments

- `pDAC` – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either `daclen < 1` or `daclen > 8`.

DAABlowfishMessageDigest

Prototype

```
IppStatus ippsDAABlowfishMessageDigest(const Ipp8u *pSrcMesg, int
   mesglen, const Ipp8u *pKey, int keylen, Ipp8u *pDAC, int daclen);
```

Description

ippsDAABlowfishMessageDigest() computes a set of Blowfish round keys derived from the input Blowfish key and applies the DAA-Blowfish scheme to transform the input message *pSrcMesg whose (byte) length is specified by input mesglen into its respective data authentication code *pDAC of the user specified length daclen (no more than 8 bytes).

The following pseudocode represents this function:

- pDAC = BlowfishEncrypt$_{pKey}$-CBC([pSrcMesg||IppsCPPaddingZEROS], pIV),

where pIV == 0, BlowfishEncrypt$_{pKey}$-CBC represents a function that computes the last ciphertext block using the Blowfish encryption algorithm under CBC mode with input key pKey, and || is the operation for concatenating two bit-streams.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream
- pKey - input Blowfish key
- daclen – the user desired data authentication code length (in unit of bytes)

Output Arguments

- pDAC – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) mesglen < 0, (2) keylen < 0 or keylen > 56, or (3) daclen < 1 or daclen > 8.

DAA-Twofish

DAA-Twofish is a Twofish block cipher based data authentication algorithm. All the Twofish block cipher primitives are described in "Twofish" uses its respective encryption key to transform the input message of any length into a data authentication code of the user specified length (no more than 16 bytes).

This section describes a set of primitives that completes the operations required for generating the data authentication code using DAA-Twofish. All of the primitives described in this section are implemented to comply with FIPS 113 and FIPS 81.

To use DAA-Twofish to generate a data authentication code, the application should use a set of five DAA-Twofish primitives. The `ippsDAATwofishInit()` sets up an initial chaining state and computes a set of Twofish round keys derived from the user supplied key. Once initialized, the primitive function `ippsDAATwofishUpdate()` executes a Twofish encryption scheme under CBC mode to the input message stream till it exhausts all the Twofish data blocks. The primitive `ippsDAATwofishFinal()` is designed to pad the possible final partial Twofish data block with the `IppCPPaddingZEROS` specified padding scheme, it then further executes the Twofish encryption scheme under CBC mode to the final Twofish data block into a `64`-bit Twofish ciphertext that yields the data authentication code `Ipp8u *pDAC` of the user specified length `daclen` (no more than 16 bytes).

This Intel® IPP also provides a primitive `ippsDAATwofishMessageDigest()` to simplify the calling convention for generating the data authentication code with respect to the input message (non streaming data) as outlined above. This primitive uses the user supplied Twofish key and applies the DAA-Twofish scheme to transform the input message into a data authentication code `Ipp8u *pDAC` of the user specified length `daclen` (no more than 16 bytes).

Throughout this section, all the DAA-Twofish primitives employ a symbolic structure `ippsDAATwofish`, to serve as an operational vehicle which carries the information necessary to the operation, such as the set of round keys derived from the user input key, a partial message block, and the intermediate state values.

The application code for applying DAA-Twofish to transform the input message into a data authentication code should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsDAATwofish` by calling the primitive `ippsDAATwofishBuffersize()`.
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive `ippsDAATwofishBuffersize()`. With the allocated buffer, call the primitive `ippsDAATwofishInit()` to setup the initial chaining state and derives a set Twofish round keys.
3. Keep calling the primitive `ippsDAATwofishUpdate()` to digest incoming message stream in the queue till its completion.
4. Call the primitive `ippsDAATwofishFinal()` for the DAA-Twofish final processing step as to compute the desired data authentication code.
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsDAATwofish`, if needed.

DAATwofishBufferSize

Prototype

```
IppStatus ippsDAATwofishBufferSize(int *pSize);
```

Description

`ippsDAATwofishBufferSize()` specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure `IppsDAATwofish` to carry the information needed for generating an (up to `128` bits) data authentication code of the given input message.

Input Arguments

- `None`

Output Arguments

- `pSize`– the size of required buffer (in bytes) for `IppsDAATwofish`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

DAATwofishInit

Prototype

```
IppStatus ippsDAATwofishInit(const Ipp8u *pKey, int keylen,
   IppsDAATwofish *pState);
```

Description

ippsDAATwofishInit() initializes the symbolic structure IppsDAATwofish *pState, using user supplied buffer space pointed by the input *pState. In addition, ippsDAATwofishInit() sets up the initial chaining state IppsDAATwofish *pState, computes a set of Twofish round keys derived from the user supplied key *pKey.

Input Arguments

- pKey – the input Twofish key
- pState – pointer to the user supplied buffer.

Output Arguments

- pState – the initialized symbolic structure IppsDAATwofish.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If keylen < 0 or keylen > 32.

DAATwofishUpdate

Prototype

```
IppStatus ippsDAATwofishUpdate(const Ipp8u *pSrcMesg, int mesglen,
   IppsDAATwofish *pState);
```

Description

ippsDAATwofishUpdate() uses the DAA-Twofish scheme to digest the current input message stream *pSrcMesg whose (byte) length is specified by the input mesglen.

ippsDAATwofishUpdate() starts with integrating the previous partial Twofish data block carried by the input state variable IppsDAATwofish *pState with the input message stream *pSrcMesg and it then partitions them into multiple Twofish data blocks (each is 128-bit) with a possible additional partial block. This function then uses the chaining state value carried by *pState as an initialization vector, and executes the Twofish encryption scheme under CBC mode to each Twofish data block consecutively.

Upon the completion, `ippsDAATwofishUpdate()` updates the state variable `IppsDAATwofish *pState` with the newly generated chaining state and a possible partial Twofish data block for further message integration or padding.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

DAATwofishFinal

Prototype

```
IppStatus ippsDAATwofishFinal(Ipp8u *pDAC, int daclen, IppsDAATwofish
   *pState);
```

Description

`ippsDAATwofishFinal()` applies DAA-Twofish scheme to further transform the final chaining state with a possible remaining partial Twofish data block both carried by the input `IppsDAATwofish *pState` into an `128`-bit Twofish ciphertext which yields the desired data authentication code of the user specified length `daclen` (no more than `16` bytes).

Input Arguments

- `daclen` – the user specified message authentication code length.
- `pState` – the input chaining state

Output Arguments

- pDAC – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either daclen < 1 or daclen > 16.

DAATwofishMessageDigest

Prototype

```
IppStatus ippsDAATwofishMessageDigest(const Ipp8u *pSrcMesg, int
   mesglen, const Ipp8u *pKey, int keylen, Ipp8u *pDAC, int daclen);
```

Description

ippsDAATwofishMessageDigest() computes a set of Twofish round keys derived from the input Twofish key and applies the DAA-Twofish scheme to transform the input message *pSrcMesg whose (byte) length is specified by input mesglen into its respective data authentication code *pDAC of the user specified length daclen (no more than 16 bytes).

The following pseudocode represents this function:

- pDAC = TwofishEncrypt$_{pKey}$-CBC([pSrcMesg||IppsCPPaddingZEROS], pIV),

where pIV == 0, TwofishEncrypt$_{pKey}$-CBC represents a function that computes the last ciphertext block using the Twofish encryption algorithm under CBC mode with input key pKey, and || is the operation for concatenating two bit-streams.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream
- pKey - input Twofish key
- daclen – the user desired data authentication code length (in unit of bytes)

Output Arguments

- pDAC – the data authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) `mesglen < 0`, (2) `keylen < 0` or `keylen > 32`, or (3) `daclen < 1 or daclen > 16`.

One-Way Hash Function Primitives

This section describes the Intel® IPP cryptographic primitives that provide message digest operations using a FIPS-approved hash scheme.

A hash function is a one-way transformation designed to digest a message of an arbitrary bit length into a short bit stream with a fixed bit length that is known as a hash value, a message digest, or a digital fingerprint. Its one-way characteristics makes it computationally infeasible to create a message hash collision. Intel® IPP provides primitives that support message digest operations complying with:

- The SHA-1 hash scheme, specified in FIPS 180-1
- The SHA-256, SHA-384 and SHA-512 schemes, specified by NIST in "Secure Hash Standard October 2000"
- MD5 hash scheme, specified by RFC 1321

In the interest of simplicity and consistency, both the mathematical expressions and pseudocode given in this section describe the behavior of each function in this section. This section describes the following primitive:

- "SHA-1"
- "SHA-256/384/512"
- "MD5"

SHA-1

SHA-1 is a technical revision of SHA (Secure Hash Algorithm) that was published by the NIST in FIPS-180, it corrected an unpublished flaw in the original SHA scheme, and became the NIST message digest standard as FIPS 180-1 in 1995. SHA-1 takes a message of less than 264 bits in length and generates an 160-bit message digest of the input. SHA-1 scheme has a wide range of application for secure data communication. Especially, it serves as a message digest function for DSA (Digital Signature Algorithm) specified by the DSS (Digital Signature Standard) published in FIPS 186-2.

This section describes the set of five primitives that completes the operations required for the message digesting using the SHA-1 scheme. All of the primitives described in this section are implemented to comply with the American Standard FIPS 180-1. The primitive ippsSHA1Init()

sets up an initial chaining state with FIPS 180-1 specified initialization vectors. Once initialized, the primitive ippsSHA1Update() digests the input message stream with the SHA-1 hash scheme till it exhausts all the SHA-1 message block. In addition, it makes the symbolic structure ippsSHA1 to carry the final partial message block. The function ippsSHA1Final() is designed to pad the partial message block into a final SHA-1 message block with the padding scheme specified in FIPS 180-1, and it further uses the SHA-1 hash scheme to transform the final block into an 160-bit message digest value Ipp8u *pMD.

This Intel® IPP also provides a primitive ippsSHA1MessageDigest() to simplify the calling convention for digesting the input message as outlined above. This primitive takes the input message stream and transforms them into an 160-bit message digest value Ipp8u *pMD.

Throughout this section, all the primitives employ a symbolic structure ippsSHA1 to serve as an operational vehicle which carries the bitlength of the processed message and a message container containing a partial message block, as well as it manages the chaining state variable.

The application code for applying SHA-1 hash standard to digest an input message stream should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure ippsSHA1 by calling the primitive ippsSHA1Buffersize().
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive ippsSHA1Buffersize(). With the allocated buffer, call the primitive ippsSHA1Init()to setup the initial chaining state with the SHA-1 specified initialization vectors.
3. Keep calling the primitive ippsSHA1Update() to digest incoming message stream in the queue till its completion.
4. Call the primitive ippsSHA1Final()for padding the partial block into a final SHA-1 message block and transforming it into an 160-bit message digest value
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure ippsSHA1 if needed.

SHA1BufferSize

Prototype

```
IppStatus ippsSHA1BufferSize(int *pSize);
```

Description

ippsSHA1BufferSize() specifies the buffer size (in number of bytes) required for defining a structured working buffer of symbolic data type IppsSHA1 to carry the chaining state in the process of generating an 160-bit hash value for the text of the given message.

Input Arguments

- None

Output Arguments

- pSize – the size of required buffer (in bytes) for initializing respective IppsSHA1

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers

SHA1Init

Prototype

```
IppStatus ippsSHA1Init(IppsSHA1 *pState);
```

Description

ippsSHA1Init() initializes the symbolic data type IppsSHA1 *pState, using user supplied buffer space pointed by the input *pState. In addition, ippsSHA1Init() sets up the initial chaining state IppsSHA1 *pState with the FIPS 180-1 specified 160-bit initialization vector IV.

Input Arguments

- pState – pointer to the user supplied buffer.

Output Arguments

- pState – the chaining state initialized with SHA-1 initialization vector.

Returns

- ippStsNoErr – no error

- ippStsNullPtrErr – NULL pointers

SHA1Update

Prototype

```
IppStatus ippsSHA1Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA1
   *pState);
```

Description

ippsSHA1Update() digests the current input message stream const Ipp8u *pSrcMesg whose (byte) length is specified by the input int length.

ippsSHA1Update() starts with integrating the previous partial block carried by the input state variable IppsSHA1 *pState with the input message stream *pSrcMesg and it then partitions them into multiple SHA-1 message blocks (each is 512-bit) with a possible additional partial block. For each SHA-1 message block, this function uses FIPS 180-1 defined SHA-1 hash scheme to transform it into a new chaining state.

Upon the completion, ippsSha1Update() updates the state variable IppsSHA1 *pState with the newly generated chaining state and the bit length of the processed message. At the end of the execution, the symbolic structure IppsSHA1 *pState carries a possible partial message block for future padding or integration.

Input Arguments

- pSrcMesg – message stream
- mesglen – length of message stream (in bytes)
- pState – the input chaining state

Output Arguments

- pState – the updated chaining state

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr – if mesglen < 0

SHA1Final

Prototype

```
IppStatus ippsSHA1Final(Ipp8u *pMD, IppsSHA1 *pState);
```

Description

`ippsSHA1Final()` transforms the final chaining state carried by the input `IppsSHA1 *pState` into an `160`-bit hash value using the method specified by the SHA1 scheme published in FIPS 180-1.

`ippsSHA1Final()` starts with applying the SHA1 padding scheme to pad the final partial message block carried by input `state` into a final SHA-1 message block with the sum of the bit lengths of the processed message the partial block. With this newly established final SHA-1 message block, `ippsSHA1Final()` further uses the SHA-1 hash scheme to transform it into a new chaining state and turns it into an `160`-bit message digest value `Ipp8u *pMD`.

Input Arguments

- `pState` – the input chaining state

Output Arguments

- `pMD` – the `160`-bit message digest value result

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

SHA1MessageDigest

Prototype

```
IppStatus ippsSHA1MessageDigest(const Ipp8u *pSrcMesg, int mesglen,
   Ipp8u *pMD);
```

Description

`ippsSHA1MessageDigest()` uses the SHA1 hash scheme published in FIPS 180-1 to transform the input message stream `const Ipp8u *pSrcMesg` whose (byte) length is specified by input `int length` into an `160`-bit message digest value `Ipp8u *OutMd`.

`ippsSHA1MessageDigest()` starts with the SHA1 padding scheme to pad the input message stream `*pSrcMesg` into multiple SHA-1 message blocks (each block is `512-bit)`. For each SHA-1 message block, `ippsSHA1MessageDigest()` transforms it into the respective internal chaining state. Upon the completion, `ippsSHA1MessageDigest()` converts the final internal state into an `160`-bit message digest value `Ipp8u *pMD`.

Input Arguments

- `pSrcMesg` – input message stream
- `mesglen` – the (byte) length of message stream

Output Arguments

- `pMD` – the `160`-bit message digest value result

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `length` < `0`

SHA-256/384/512

SHA-256, SHA-384 and SHA-512 are newly proposed FIPS standard secure hash algorithms (draft). They are intended to be a set of companion standard hash algorithms that offers a similar level of security enhanced by the AES cipher systems. SHA-256 takes a message of less than 2^{64} bits in length, operates on `512`-bit message blocks with `256`-bit intermediate hashing state values and generates a `256`-bit message digest of the input. Similarly, SHA-384 and SHA-512 compress a message of less than 2^{64} bits in length into a `384`-bit and `512`-bit hash values respectively. Both hash algorithms operate on `1024`-bit message blocks with `512`-bit intermediate hashing state values.

This section describes three sets of primitives that complete the operations required for the message digesting using hash schemes SHA-256, SHA-384 and SHA-512. All of the primitives described in this section are implemented to comply with the FIPS proposed draft, the Secure Hash Standard October 2000.

Throughout this section, all the SHA-256 hash function primitives employ a symbolic structure `ippsSHA256`, similarly, all the SHA-384 and SHA-512 hash function primitives employ symbolic structures `ippsSHA384`, and `ippsSHA512`. This set of symbolic structures serves as operational vehicles which carry the information necessary to the respective hashing mechanism, such as the bitlength of the processed message, a partial message block, and the intermediate hashing state values.

The application code for applying SHA-256 hash standard (same for SHA-384 or SHA-512) to digest an input message (streaming data) should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsSHA1` by calling the primitive `ippsSHA256Buffersize()`.
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive `ippsSHA256Buffersize()`. With the allocated buffer, call the primitive `ippsSHA256Init()` to setup the initial chaining state with the SHA-256 specified initialization vectors.
3. Keep calling the primitive `ippsSHA256Update()` to digest incoming message stream in the queue till its completion.
4. Call the primitive `ippsSHA256Final()` for padding the partial block into a final SHA-256 message block and transforming it into a `256`-bit message digest value
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsSHA256` if needed.

The application code for applying SHA-256 hash standard (same for SHA-384 or SHA-512) to digest an input message (non-streaming data) could also use `ippsSHA256MessageDigest()`.

SHA256BufferSize

Prototype

```
IppStatus ippsSHA256BufferSize(int *pSize);
```

Description

`ippsSHA256BufferSize()` specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure `IppsSHA256` to carry the information necessary to the process of generating an `256`-bit hash value for the text of the given input message.

Input Arguments

- None

Output Arguments

- `pSize`– the size of required buffer (in bytes) for initializing `IppsSHA256`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA256Init

Prototype

```
IppStatus ippsSHA256Init(IppsSHA256 *pState);
```

Description

`ippsSHA256Init()` initializes the symbolic structure `IppsSHA256 *pState,` using user supplied buffer space pointed by the input `*pState.` In addition, `ippsSHA256Init()` sets up the initial chaining state `IppsSHA256 *pState` with the FIPS specified `256`-bit initialization vector IV.

Input Arguments

- `pState` – pointer to the user supplied buffer.

Output Arguments

- `pState` – the chaining state initialized with SHA-256 initialization vector.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA256Update

Prototype

```
IppStatus ippsSHA256Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsSHA256 *pState);
```

Description

`ippsSHA256Update()` digests the current input message stream `const Ipp8u *pSrcMesg` whose (byte) length is specified by the input `int mesglen`.

`ippsSHA256Update()` starts with integrating the previous partial block carried by the input state variable `IppsSHA256 *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple SHA-256 message blocks (each is `512`-bit) with a possible additional partial block. For each SHA-256 message block, this function uses the SHA-256 hash scheme to transform it into a new chaining state.

Upon the completion, `ippsSha256Update()` updates the state variable `IppsSHA256 *pState` with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure `IppsSHA256 *pState` carries a possible partial message block for future padding or integration.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

SHA256Final

Prototype

```
IppStatus ippsSHA256Final(Ipp8u *pMD, IppsSHA256 *pState);
```

Description

`ippsSHA256Final()` transforms the final chaining state carried by the input `IppsSHA256 *pState` into a 256-bit hash value in the way specified by the SHA-256 scheme.

`ippsSHA256Final()` starts with applying the SHA-256 padding scheme to pad the final partial message block carried by input `pState` into a final SHA-256 message block with the sum of the bitlengths of the processed message in the partial block. With this newly established final SHA-256 message block, `ippsSHA256Final()` further uses the SHA-256 hash scheme to transform it into a new chaining state and turns it into an `256`-bit message digest value `Ipp8u *pMD`.

Input Arguments

- `pState` – the input chaining state

Output Arguments

- `pMD` – the `256`-bit message digest value result

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA256MessageDigest

Prototype

```
IppStatus ippsSHA256MessageDigest(const Ipp8u *pSrcMesg, int mesglen,
   Ipp8u *pMD);
```

Description

ippsSHA256MessageDigest() uses the SHA-256 hash scheme to transform the input message const Ipp8u *pSrcMesg whose (byte) length is specified by input int mesglen into an 256-bit message digest value Ipp8u *OutMd.

ippsSHA256MessageDigest() starts with the SHA-256 padding scheme to pad the input message *pSrcMesg and partition it into multiple SHA-256 message blocks (each block is 512-bit). For each SHA-256 message block, ippsSHA256MessageDigest() transforms it into the respective internal chaining state. Upon the completion, ippsSHA256MessageDigest() converts the final internal state into an 256-bit message digest value Ipp8u *pMD.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream

Output Arguments

- pMD – the 256-bit message digest value result

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If mesglen < 0.

SHA384BufferSize

Prototype

```
IppStatus ippsSHA384BufferSize(int *pSize);
```

Description

ippsSHA1384BufferSize() specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure IppsSHA384 to carry the information necessary to the process of generating a 384-bit hash value for the text of the given input message.

Input Arguments

- None

Output Arguments

- `pSize`– the size of required buffer (in bytes) for initializing `IppsSHA384`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA384Init

Prototype

```
IppStatus ippsSHA384Init(IppsSHA384 *pState);
```

Description

`ippsSHA384Init()` initializes the symbolic structure `IppsSHA384 *pState,` using user supplied buffer space pointed by the input `*pState.` In addition, `ippsSHA384Init()` sets up the initial chaining state `IppsSHA384 *pState` with the FIPS specified `384`-bit initialization vector IV.

Input Arguments

- `pState` – pointer to the user supplied buffer.

Output Arguments

- `pState` – the chaining state initialized with SHA-384 initialization vector.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA384Update

Prototype

```
IppStatus ippsSHA384Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsSHA384 *pState);
```

Description

`ippsSHA384Update()` digests the current input message stream `const Ipp8u *pSrcMesg` whose (byte) length is specified by the input `int mesglen`.

`ippsSHA384Update()` starts with integrating the previous partial block carried by the input state variable `IppsSHA384 *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple SHA-384 message blocks (each is `1024`-bit) with a possible additional partial block. For each SHA-384 message block, this function uses the SHA-384 hash scheme to transform it into a new chaining state.

Upon the completion, `ippsSha384Update()` updates the state variable `IppsSHA384 *pState` with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure `IppsSHA384 *pState` carries a possible partial message block for future padding or integration.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

SHA384Final

Prototype

```
IppStatus ippsSHA384Final(Ipp8u *pMD, IppsSHA384 *pState);
```

Description

`ippsSHA384Final()` transforms the final chaining state carried by the input `IppsSHA384 *pState` into a 384-bit hash value in the way specified by the SHA-384 scheme.

`ippsSHA384Final()` starts with applying the SHA-384 padding scheme to pad the final partial message block carried by input `pState` into a final SHA-384 message block with the sum of the bitlengths of the processed message and the partial block. With this newly established final SHA-384 message block, `ippsSHA384Final()` further uses the SHA-384 hash scheme to transform it into a new chaining state and turns it into an `384`-bit message digest value `Ipp8u *pMD`.

Input Arguments

- `pState` – the input chaining state

Output Arguments

- `pMD` – the `384`-bit message digest value result

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA384MessageDigest

Prototype

```
IppStatus ippsSHA384MessageDigest(const Ipp8u *pSrcMesg, int mesglen,
   Ipp8u *pMD);
```

Description

ippsSHA384MessageDigest() uses the SHA-384 hash scheme to transform the input message const Ipp8u *pSrcMesg whose (byte) length is specified by input int mesglen into an 384-bit message digest value Ipp8u *OutMd.

ippsSHA384MessageDigest() starts with the SHA-384 padding scheme to pad the input message *pSrcMesg and partition it into multiple SHA-384 message blocks (each block is 1024-bit). For each SHA-384 message block, ippsSHA384MessageDigest() transforms it into the respective internal chaining state. Upon the completion, ippsSHA384MessageDigest() converts the final internal state into an 384-bit message digest value Ipp8u *pMD.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream

Output Arguments

- pMD – the 384-bit message digest value result

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If mesglen < 0.

SHA512BufferSize

Prototype

```
IppStatus ippsSHA512BufferSize(int *pSize);
```

Description

ippsSHA1512BufferSize() specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure IppsSHA512 to carry the information necessary to the process of generating a 512-bit hash value for the text of the given input message.

Input Arguments

- None

Output Arguments

- `pSize`– the size of required buffer (in bytes) for initializing `IppsSHA512`.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA512Init

Prototype

```
IppStatus ippsSHA512Init(IppsSHA512 *pState);
```

Description

`ippsSHA512Init()` initializes the symbolic structure `IppsSHA512 *pState,` using user supplied buffer space pointed by the input `*pState.` In addition, `ippsSHA512Init()` sets up the initial chaining state `IppsSHA512 *pState` with the FIPS specified `512`-bit initialization vector IV.

Input Arguments

- `pState` – pointer to the user supplied buffer.

Output Arguments

- `pState` – the chaining state initialized with SHA-512 initialization vector.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA512Update

Prototype

```
IppStatus ippsSHA512Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsSHA512 *pState);
```

Description

`ippsSHA512Update()` digests the current input message stream `const Ipp8u *pSrcMesg` whose (byte) length is specified by the input `int mesglen`.

`ippsSHA512Update()` starts with integrating the previous partial block carried by the input state variable `IppsSHA512 *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple SHA-512 message blocks (each is `1024`-bit) with a possible additional partial block. For each SHA-512 message block, this function uses the SHA-512 hash scheme to transform it into a new chaining state.

Upon the completion, `ippsSha512Update()` updates the state variable `IppsSHA512 *pState` with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure `IppsSHA512 *pState` carries a possible partial message block for future padding or integration.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

SHA512Final

Prototype

```
IppStatus ippsSHA512Final(Ipp8u *pMD, IppsSHA512 *pState);
```

Description

`ippsSHA512Final()` transforms the final chaining state carried by the input `IppsSHA512 *pState` into a 512-bit hash value in the way specified by the SHA-512 scheme.

`ippsSHA512Final()` starts with applying the SHA-512 padding scheme to pad the final partial message block carried by input `pState` into a final SHA-512 message block with the sum of the bitlengths of the processed message and the partial block. With this newly established final SHA-512 message block, `ippsSHA512Final()` further uses the SHA-512 hash scheme to transform it into a new chaining state and turns it into an `512`-bit message digest value `Ipp8u *pMD`.

Input Arguments

- `pState` – the input chaining state

Output Arguments

- `pMD` – the `512`-bit message digest value result

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

SHA512MessageDigest

Prototype

```
IppStatus ippsSHA512MessageDigest(const Ipp8u *pSrcMesg, int mesglen,
   Ipp8u *pMD);
```

Description

ippsSHA512MessageDigest() uses the SHA-512 hash scheme to transform the input message const Ipp8u *pSrcMesg whose (byte) length is specified by input int mesglen into an 512-bit message digest value Ipp8u *OutMd.

ippsSHA512MessageDigest() starts with the SHA-512 padding scheme to pad the input message *pSrcMesg and partition it into multiple SHA-512 message blocks (each block is 1024-bit). For each SHA-512 message block, ippsSHA512MessageDigest() transforms it into the respective internal chaining state. Upon the completion, ippsSHA512MessageDigest() converts the final internal state into an 512-bit message digest value Ipp8u *pMD.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream

Output Arguments

- pMD – the 512-bit message digest value result

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If mesglen < 0

MD5

MD5 was designed to be a security enhanced version of the hash scheme over MD4. It has been long standing in the wide-range applications for data communication. MD5 compresses a message of any length into an 128-bit hash value. It operates on 512-bit message blocks with 128-bit intermediate hashing state values.

This section describes a set of primitives that completes the operations required for the message digesting using hash scheme MD5. All of the primitives described in this section are implemented to comply with RFC 1321.

Throughout this section, all the primitives employ a symbolic structure ippsMD5. This symbolic structure serves as operational vehicles which carry the information necessary to the respective hashing mechanism, such as the bitlength of the processed message, a partial message block, and the intermediate hashing state values.

The application code for applying MD5 hash algorithm to digest an input message (streaming data) should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure `ippsMD5` by calling the primitive `ippsMD5Buffersize()`.
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive `ippsMD5Buffersize()`. With the allocated buffer, call the primitive `ippsMD5Init()` to setup the initial chaining state with the MD5 specified initialization vectors.
3. Keep calling the primitive `ippsMD5Update()` to digest incoming message stream in the queue till its completion.
4. Call the primitive `ippsMD5Final()` for padding the partial block into a final MD5 message block and transforming it into an `128`-bit message digest value
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure `ippsMD5` if needed.

The application code for applying MD5 hash standard to digest an input message (non-streaming data) could also use `ippsMD5MessageDigest()`.

MD5BufferSize

Prototype

```
IppStatus ippsMD5BufferSize(int *pSize);
```

Description

`ippsMD5BufferSize()` specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure `IppsMD5` to carry the information necessary to the process of generating an `128`-bit hash value for the text of the given input message.

Input Arguments

- None

Output Arguments

- `pSize`– the size of required buffer (in bytes) for initializing `IppsMD5`.

Returns

- ippStsNoErr – No Error

- ippStsNullPtrErr – Null pointers

MD5Init

Prototype

```
IppStatus ippsMD5Init(IppsMD5 *pState);
```

Description

`ippsMD5Init()` initializes the symbolic structure `IppsMD5 *pState`, using user supplied buffer space pointed by the input `*pState`. In addition, `ippsMD5Init()` sets up the initial chaining state `IppsMD5 *pState` with the FIPS specified `128`-bit initialization vector IV.

Input Arguments

- `pState` – pointer to the user supplied buffer.

Output Arguments

- `pState` – the chaining state initialized with MD5 initialization vector.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

MD5Update

Prototype

```
IppStatus ippsMD5Update(const Ipp8u *pSrcMesg, int mesglen, IppsMD5
   *pState);
```

Description

`ippsMD5Update()` digests the current input message stream `const Ipp8u *pSrcMesg` whose (byte) length is specified by the input `int mesglen`.

`ippsMD5Update()` starts with integrating the previous partial block carried by the input state variable `IppsMD5 *pState` with the input message stream `*pSrcMesg` and it then partitions them into multiple MD5 message blocks (each is `512`-bit) with a possible additional partial block. For each MD5 message block, this function uses the MD5 hash scheme to transform it into a new chaining state.

Upon the completion, `ippsMD5Update()` updates the state variable `IppsMD5 *pState` with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure `IppsMD5 *pState` carries a possible partial message block for future padding or integration.

Input Arguments

- `pSrcMesg` – message stream
- `mesglen` – length of message stream (in bytes)
- `pState` – the input chaining state

Output Arguments

- `pState` – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`

MD5Final

Prototype

```
IppStatus ippsMD5Final(Ipp8u *pMD, IppsMD5 *pState);
```

Description

ippsMD5Final() transforms the final chaining state carried by the input IppsMD5 *pState into an 128-bit hash value in the way specified by the MD5 scheme.

ippsMD5Final() starts with applying the MD5 padding scheme to pad the final partial message block carried by input pState into a final MD5 message block with the sum of the bitlengths of the processed message and that of the partial block. With this newly established final MD5 message block, ippsMD5Final() further uses the MD5 hash scheme to transform it into a new chaining state and turns it into an 128-bit message digest value Ipp8u *pMD.

Input Arguments

- pState – the input chaining state

Output Arguments

- pMD – the 128-bit message digest value result

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

MD5MessageDigest

Prototype

```
IppStatus ippsMD5MessageDigest(const Ipp8u *pSrcMesg, int mesglen, Ipp8u
   *pMD);
```

Description

ippsMD5MessageDigest() uses the MD5 hash scheme to transform the input message const Ipp8u *pSrcMesg whose (byte) length is specified by input int mesglen into an 128-bit message digest value Ipp8u *OutMd.

`ippsMD5MessageDigest()` starts with the MD5 padding scheme to pad the input message `*pSrcMesg` and partition it into multiple MD5 message blocks (each block is `512-bit`). For each MD5 message block, `ippsMD5MessageDigest()` transforms it into the respective internal chaining state. Upon the completion, `ippsMD5MessageDigest()` converts the final internal state into an `128`-bit message digest value `Ipp8u *pMD`.

Input Arguments

- `pSrcMesg` – input message stream
- `mesglen` – the (byte) length of message stream

Output Arguments

- `pMD` – the `128`-bit message digest value result

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If `mesglen < 0`.

Keyed-Hash Message Authentication Code Primitives

This section describes the Intel® IPP for the PCA processors with MMX™ that are available for generating the keyed-hash based message authentication code using the HMAC scheme specified in the FIPS-198.

The keyed-hash based message authentication code (HMAC) has been widely used in the applications requiring message authentication, as well as data integrity check. It has been put forward initially in RFC 2104, late adopted by ANSI X9.71 and FIPS-198. In this release,Intel® IPP offers primitives that are designed for applications to use various HMAC schemes based on a set of one-way hash functions described in the Section 5, “One-Way Hash Function Primitives” on page 91. The related standards that the primitives in this section conform to are listed as the following:

- The HMAC scheme, specified in FIPS 198
- The SHA-1 hash scheme, specified in FIPS 180-1
- The SHA-256, SHA-384 and SHA-512 schemes, specified by NIST in “Secure Hash Standard October 2000”
- MD5 hash scheme, specified by RFC 1321

This section describes the following primitives:

“HMAC-SHA1”

"HMAC-SHA256/384/512"
"HMAC-MD5"

HMAC-SHA1

HMAC-SHA1 is the keyed SHA-1 hash based message authentication code scheme. HMAC-SHA1 takes the input message with the user supplied key (no less than 10 bytes) to generate the respective message authentication code of the user specified length (no more than 20 bytes).

This section describes the set of five primitives that completes the operations required for generating the message authentication code using HMAC-SHA1. All of the primitives described in this section are implemented to comply with the American Standard FIPS 198 and FIPS 180-1. The primitive ippsHMACSHA1Init() sets up an initial chaining state and computes both the keyed-ipad and keyed-opad derived from the user supplied key. Once initialized, the primitive function ippsHMACSHA1Update() applies SHA-1 to digest the input message stream till it exhausts all the SHA-1 message block. The function ippsHMACSHA1Final() is designed to pad the partial message block into a final SHA-1 message block with the padding scheme specified in FIPS 180-1, it then uses the SHA-1 hash scheme to transform the final block into an 160-bit SHA-1 message digest value. By concatenating the key-opad with the hash value, this function further generates the message authentication code Ipp8u *pMAC of the user specified length maclen (no more than 20 bytes).

This Intel® IPP also provides a primitive ippsHMACSHA1MessageDigest() to simplify the calling convention for generating the message authentication code with respect to the input message (non streaming data) as outlined above. This primitive takes the input message with the user supplied key and transforms it into an HMAC-SHA1 message authentication code Ipp8u *pMAC of user specified length maclen (no more than 20 bytes).

Throughout this section, all the primitives employ a symbolic structure ippsHMACSHA1 to serve as an operational vehicle which carries the information necessary to the operation, such as the keyed-ipad and keyed-opad values derived from the user input key, the bitlength of the processed message, a partial message block, and the intermediate hashing state values.

The application code for applying HMAC-SHA1 to generate a message authentication code of the input should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure ippsHMACSHA1 by calling the primitive ippsHMACSHA1Buffersize().
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive ippsHMACSHA1Buffersize(). With the allocated buffer, call the primitive ippsHMACSHA1Init()to setup the initial chaining state and derives both keyed-ipad and keyed-opad values.

3. Keep calling the primitive ippsHMACSHA1Update() to digest incoming message stream in the queue till its completion.
4. Call the primitive ippsHMACSHA1Final()for processing the final partial block and with the concatenation of the keyed-opad value, it finally transforms it into the desired message authentication code.
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure ippsHMACSHA1, if needed.

HMACSHA1BufferSize

Prototype

```
IppStatus ippsHMACSHA1BufferSize(int *pSize);
```

Description

ippsHMACSHA1BufferSize() specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure IppsHMACSHA1 to carry the information needed for generating an (up to 160 bits) message authentication code of the given input message.

Input Arguments

- none.

Output Arguments

- pSize– the size of required buffer (in bytes) for IppsHMACSHA1.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

HMACSHA1Init

Prototype

```
IppStatus ippsHMACSHA1Init(const Ipp8u *pKey, int keylen, IppsHMACSHA1
   *pState);
```

Description

ippsHMACSHA1Init() initializes the symbolic structure IppsHMACSHA1 *pState, using user supplied buffer space pointed by the input *pState. In addition, ippsHMACSHA1Init() sets up the initial chaining state IppsHMACSHA1 *pState, computes both the keyed-ipad and the keyed-opad values derived from the user supplied key *pKey, and put the keyed-ipad values in a data container that is carried by the symbolic structure IppsHMACSHA1 *pState for the concatenation with the incoming message stream.

Input Arguments

- pKey – pointer to the user supplied key byte stream
- keylen – key length (in unit of byte)
- pState – pointer to the user supplied buffer.

Output Arguments

- pState – the initialized symbolic structure IppsHMACSHA1.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If keylen < 0.

HMACSHA1Update

Prototype

```
IppStatus ippsHMACSHA1Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsHMACSHA1 *pState);
```

Description

ippsHMACSHA1Update() digests the current input message stream const Ipp8u *pSrcMesg whose (byte) length is specified by the input int mesglen.

ippsHMACSHA1Update() starts with integrating the previous partial block carried by the input state variable IppsHMACSHA1 *pState with the input message stream *pSrcMesg and it then partitions them into multiple SHA-1 message blocks (each is 512-bit) with a possible additional partial block. This function then uses FIPS 180-1 defined SHA-1 hash scheme to digest each SHA-1 message block consecutively.

Upon the completion, ippsHMACSHA1Update() updates the state variable IppsHMACSHA1 *pState with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure IppsHMACSHA1 *pState carries a new chaining state value with a possible partial message block for further message integration or padding.

Input Arguments

- pSrcMesg – message stream
- mesglen – length of message stream (in bytes)
- pState – the input chaining state

Output Arguments

- pState – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If mesglen < 0

HMACSHA1Final

Prototype

```
IppStatus ippsHMACSHA1Final(Ipp8u *pMAC, int maclen, IppsHMACSHA1
   *pState);
```

Description

ippsHMACSHA1Final() transforms the final chaining state carried by the input IppsHMACSHA1 *pState into an 160-bit hash value using the method specified by the SHA1 scheme published in FIPS 180-1.

By concatenating the keyed-opad value with the generated hash value, this function further generates the message authentication code of the user specified length maclen (no more than 20 bytes).

Input Arguments

- maclen – the user specified message authentication code length.
- pState – the input chaining state

Output Arguments

- pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either maclen < 1 or maclen > 20.

HMACSHA1MessageDigest

Prototype

```
IppStatus ippsHMACSHA1MessageDigest(const Ipp8u *pSrcMesg, int mesglen,
   const Ipp8u *pKey, int keylen, Ipp8u *pMAC, int maclen);
```

Description

ippsHMACSHA1MessageDigest() takes the input key of the user specified key length keylen (no less than 10 bytes) and applies the keyed SHA-1 hash based message authentication code scheme to transform the input message *pSrcMesg whose (byte) length is specified by input mesglen into its respective message authentication code Ipp8u *pMAC of user specified length maclen (no more than 20 bytes).

The following pseudocode represents this function:

pMAC = SHA1([pKey XOR opad]||[SHA1([pKey XOR ipad]||[pSrcMesg])])

where SHA1 represents the SHA-1 hash function, XOR is the bitwise exclusive-OR and || is the operation for concatenating two bit-streams.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream
- pKey – the user specified authentication key
- keylen – the key length (in unit of bytes)
- maclen – the user desired message authentication code length (in unit of bytes)

Output Arguments

- pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) mesglen < 0, (2) keylen < 1, or (3) maclen < 1 or maclen > 20.

HMAC-SHA256/384/512

HMAC-SHA256, HMAC-SHA384 and HMAC-SHA512 are the keyed hash message authentication code schemes with respect to their baseline one-way hash functions SHA-256, SHA-384 and SHA-512. HMAC-SHA256 takes the input message with the user supplied key (no less than 16 bytes) to generate the respective message authentication code of the user specified length (no more than 32 bytes). Similarly, HMAC-SHA384 uses the authentication key (no less than 24 bytes) to transform the input message into the respective message authentication code (no more than 48 bytes), and HMAC-SHA512 requires an authentication key (no less than 32 bytes) for processing the input message to generate its message authentication code (no more than 64 bytes).

This section describes three sets of primitive functions that are developed to complete the operations required for generating the message authentication code using HMAC-SHA256, HMAC-SHA384, and HMAC-SHA512. All of the primitives described in this section are implemented to comply with the American Standard FIPS 198 and the FIPS proposed draft, the Secure Hash Standard October 2000.

To use the keyed SHA-256 based HMAC scheme, application should take the set of five HMAC-SHA256 primitives. The primitive ippsHMACSHA256Init() sets up an initial chaining state and computes both the keyed-ipad and keyed-opad derived from the user supplied key. Once initialized, the primitive function ippsHMACSHA256Update() applies SHA-256 to digest the input message stream till it exhausts all the SHA-256 message block. The function ippsHMACSHA256Final() is designed to pad the partial message block into a final SHA-256 message block with the padding scheme specified in FIPS draft, the Secure Hash Standard October 2000, it then uses the SHA-256 hash scheme to transform the final block into a 256-bit SHA-1 message digest value. By concatenating the key-opad with the hash value, this function further generates the message authentication code Ipp8u *pMAC of the user specified length maclen (no more than 32 bytes).

This Intel® IPP also provides a primitive ippsHMACSHA256MessageDigest() to simplify the calling convention for generating the message authentication code with respect to the input message (non streaming data) as outlined above. This primitive takes the input message with the user supplied key and transforms it into an HMAC-SHA256 message authentication code Ipp8u *pMAC of user specified length maclen (no more than 32 bytes).

Similarly, to use the keyed SHA-384 based HMAC scheme, application should use the set of five HMAC-SHA384 primitives, and for using the keyed SHA-512 based HMAC scheme, application should use the set of five HMAC-SHA512 primitives.

Throughout this section, all the HMAC-SHA256 primitives employ a symbolic structure ippsHMACSHA256, similarly, all the HMAC-SHA384 primitives employ a symbolic structure ippsHMACSHA384, and all the HMAC-SHA512 primitives employ a symbolic structure ippsHMACSHA512. These three symbolic structures are designed to serve as operational vehicles

which carry the information necessary to the operation, such as the keyed-ipad and keyed-opad values derived from the user input key, the bitlength of the processed message, a partial message block, and the intermediate hashing state values.

The application code for applying HMAC-SHA256 (similar procedure applies to both HMAC-SHA384 and HMAC-SHA512) to generate a message authentication code of the input should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure ippsHMACSHA256 by calling the primitive ippsHMACSHA256Buffersize().
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive ippsHMACSHA256Buffersize(). With the allocated buffer, call the primitive ippsHMACSHA256Init()to setup the initial chaining state and derives both keyed-ipad and keyed-opad values.
3. Keep calling the primitive ippsHMACSHA256Update() to digest incoming message stream in the queue till its completion.
4. Call the primitive ippsHMACSHA256Final()for processing the final partial block and with the concatenation of the keyed-opad value, it finally transforms it into the desired message authentication code.
5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure ippsHMACSHA256, if needed.

HMACSHA256BufferSize

Prototype

```
IppStatus ippsHMACSHA256BufferSize(int *pSize);
```

Description

ippsHMACSHA256BufferSize() specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure IppsHMACSHA256 to carry the information needed for generating an (up to 256 bits) message authentication code of the given input message.

Input Arguments

- none.

Output Arguments

pSize– the size of required buffer (in bytes) for initializing respective IppsHMACSHA256.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

HMACSHA256Init

Prototype

```
IppStatus ippsHMACSHA256Init(const Ipp8u *pKey, int keylen,
   IppsHMACSHA256 *pState);
```

Description

ippsHMACSHA256Init() initializes the symbolic structure IppsHMACSHA256 *pState, using user supplied buffer space pointed by the input *pState. In addition, ippsHMACSHA256Init() sets up the initial chaining state IppsHMACSHA256 *pState, computes both the keyed-ipad and the keyed-opad values derived from the user supplied key *pKey, and put the keyed-ipad values in a data container that is carried by the symbolic structure IppsHMACSHA256 *pState for the concatenation with the incoming message stream.

Input Arguments

- pKey – pointer to the user supplied key byte stream
- keylen – key length (in unit of byte)
- pState – pointer to the user supplied buffer.

Output Arguments

- pState – the initialized symbolic structure IppsHMACSHA256.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If keylen < 0.

HMACSHA256Update

Prototype

```
IppStatus ippsHMACSHA256Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsHMACSHA256 *pState);
```

Description

ippsHMACSHA256Update() digests the current input message stream const Ipp8u *pSrcMesg whose (byte) length is specified by the input int mesglen.

ippsHMACSHA256Update() starts with integrating the previous partial block carried by the input state variable IppsHMACSHA256 *pState with the input message stream *pSrcMesg and it then partitions them into multiple SHA-256 message blocks (each is 512-bit) with a possible additional partial block. This function then uses SHA-256 hash scheme to digest each SHA-256 message block consecutively.

Upon the completion, ippsHMACSHA256Update() updates the state variable IppsHMACSHA256 *pState with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure IppsHMACSHA256 *pState carries a new chaining state value with a possible partial message block for further message integration or padding.

Input Arguments

- pSrcMesg – message stream
- mesglen – length of message stream (in bytes)
- pState – the input chaining state

Output Arguments

- pState – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If mesglen < 0

HMACSHA256Final

Prototype

```
IppStatus ippsHMACSHA256Final(Ipp8u *pMAC, int maclen, IppsHMACSHA256
   *pState);
```

Description

ippsHMACSHA256Final() transforms the final chaining state carried by the input IppsHMACSHA256 *pState into an 256-bit hash value using the method specified by the SHA-256 hash scheme.

By concatenating the keyed-opad value with the generated hash value, this function further generates the message authentication code of the user specified length maclen (no more than 32 bytes).

Input Arguments

- maclen – the user specified message authentication code length.
- pState – the input chaining state

Output Arguments

pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either maclen < 1 or maclen > 32.

HMACSHA256MessageDigest

Prototype

```
IppStatus ippsHMACSHA256MessageDigest(const Ipp8u *pSrcMesg, int
   mesglen, const Ipp8u *pKey, int keylen, Ipp8u *pMAC, int maclen);
```

Description

ippsHMACSHA256MessageDigest() takes the input key of the user specified key length keylen (no less than 16 bytes) and applies the keyed SHA-256 hash based message authentication code scheme to transform the input message *pSrcMesg whose (byte) length is specified by input mesglen into its respective message authentication code Ipp8u *pMAC of user specified length maclen (no more than 32 bytes).

The following pseudocode represents this function:

pMAC = SHA256([pKey XOR opad]||[SHA256([pKey XOR ipad]||[pSrcMesg])])

where SHA256 represents the SHA-256 hash function, XOR is the bitwise exclusive-OR and || is the operation for concatenating two bit-streams.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream
- pKey – the user specified authentication key
- keylen – the key length (in unit of bytes)
- maclen – the user desired message authentication code length (in unit of bytes)

Output Arguments

- pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) mesglen < 0, (2) keylen < 1, or (3) maclen < 1 or maclen > 32.

HMACSHA384BufferSize

Prototype

```
IppStatus ippsHMACSHA384BufferSize(int *pSize);
```

Description

ippsHMACSHA384BufferSize() specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure IppsHMACSHA384 to carry the information needed for generating an (up to 384 bits) message authentication code of the given input message.

Input Arguments

- none.

Output Arguments

- pSize– the size of required buffer (in bytes) for initializing respective IppsHMACSHA384.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

HMACSHA384Init

Prototype

```
IppStatus ippsHMACSHA384Init(const Ipp8u *pKey, int keylen,
   IppsHMACSHA384 *pState);
```

Description

ippsHMACSHA384Init() initializes the symbolic structure IppsHMACSHA384 *pState, using user supplied buffer space pointed by the input *pState. In addition, ippsHMACSHA384Init() sets up the initial chaining state IppsHMACSHA384 *pState, computes both the keyed-ipad and the

keyed-opad values derived from the user supplied key *pKey, and put the keyed-ipad values in a data container that is carried by the symbolic structure IppsHMACSHA384 *pState for the concatenation with the incoming message stream.

Input Arguments

- pKey – pointer to the user supplied key byte stream
- keylen – key length (in unit of byte)
- pState – pointer to the user supplied buffer.

Output Arguments

- pState – the initialized symbolic structure IppsHMACSHA384.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If keylen < 0.

HMACSHA384Update

Prototype

```
IppStatus ippsHMACSHA384Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsHMACSHA384 *pState);
```

Description

ippsHMACSHA384Update() digests the current input message stream const Ipp8u *pSrcMesg whose (byte) length is specified by the input int mesglen.

ippsHMACSHA384Update() starts with integrating the previous partial block carried by the input state variable IppsHMACSHA384 *pState with the input message stream *pSrcMesg and it then partitions them into multiple SHA-384 message blocks (each is 1024-bit) with a possible additional partial block. This function then uses SHA-384 hash scheme to digest each SHA-384 message block consecutively.

Upon the completion, ippsHMACSHA384Update() updates the state variable IppsHMACSHA384 *pState with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure IppsHMACSHA384 *pState carries a new chaining state value with a possible partial message block for further message integration or padding.

Input Arguments

- pSrcMesg – message stream
- mesglen – length of message stream (in bytes)
- pState – the input chaining state

Output Arguments

- pState – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If mesglen < 0

HMACSHA384Final

Prototype

```
IppStatus ippsHMACSHA384Final(Ipp8u *pMAC, int maclen, IppsHMACSHA384
   *pState);
```

Description

ippsHMACSHA384Final() transforms the final chaining state carried by the input IppsHMACSHA384 *pState into an 384-bit hash value using the method specified by the SHA-384 hash scheme.

By concatenating the keyed-opad value with the generated hash value, this function further generates the message authentication code of the user specified length maclen (no more than 48 bytes).

Input Arguments

- maclen – the user specified message authentication code length.
- pState – the input chaining state

Output Arguments

- pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either maclen < 1 or maclen > 48.

HMACSHA384MessageDigest

Prototype

```
IppStatus ippsHMACSHA384MessageDigest(const Ipp8u *pSrcMesg, int
   mesglen, const Ipp8u *pKey, int keylen, Ipp8u *pMAC, int maclen);
```

Description

ippsHMACSHA384MessageDigest() takes the input key of the user specified key length keylen (no less than 24 bytes) and applies the keyed SHA-384 hash based message authentication code scheme to transform the input message *pSrcMesg whose (byte) length is specified by input mesglen into its respective message authentication code Ipp8u *pMAC of user specified length maclen (no more than 48 bytes).

The following pseudocode represents this function:

pMAC = SHA384([pKey XOR opad]||[SHA384([pKey XOR ipad]||[pSrcMesg])])

where SHA384 represents the SHA-384 hash function, XOR is the bitwise exclusive-OR and || is the operation for concatenating two bit-streams.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream
- pKey – the user specified authentication key

- keylen – the key length (in unit of bytes)
- maclen – the user desired message authentication code length (in unit of bytes)

Output Arguments

- pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) mesglen < 0, (2) keylen < 1, or (3) maclen < 1 or maclen > 48.

HMACSHA512BufferSize

Prototype

```
IppStatus ippsHMACSHA512BufferSize(int *pSize);
```

Description

ippsHMACSHA512BufferSize() specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure IppsHMACSHA512 to carry the information needed for generating an (up to 512 bits) message authentication code of the given input message.

Input Arguments

- none.

Output Arguments

- pSize– the size of required buffer (in bytes) for initializing respective IppsHMACSHA512.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

HMACSHA512Init

Prototype

```
IppStatus ippsHMACSHA512Init(const Ipp8u *pKey, int keylen,
   IppsHMACSHA512 *pState);
```

Description

ippsHMACSHA512Init() initializes the symbolic structure IppsHMACSHA512 *pState, using user supplied buffer space pointed by the input *pState. In addition, ippsHMACSHA512Init() sets up the initial chaining state IppsHMACSHA512 *pState, computes both the keyed-ipad and the keyed-opad values derived from the user supplied key *pKey, and put the keyed-ipad values in a data container that is carried by the symbolic structure IppsHMACSHA512 *pState for the concatenation with the incoming message stream.

Input Arguments

- pKey – pointer to the user supplied key byte stream
- keylen – key length (in unit of byte)
- pState – pointer to the user supplied buffer.

Output Arguments

- pState – the initialized symbolic structure IppsHMACSHA512.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If keylen < 0.

HMACSHA512Update

Prototype

```
IppStatus ippsHMACSHA512Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsHMACSHA512 *pState);
```

Description

ippsHMACSHA512Update() digests the current input message stream const Ipp8u *pSrcMesg whose (byte) length is specified by the input int mesglen.

ippsHMACSHA512Update() starts with integrating the previous partial block carried by the input state variable IppsHMACSHA512 *pState with the input message stream *pSrcMesg and it then partitions them into multiple SHA-512 message blocks (each is 1024-bit) with a possible additional partial block. This function then uses SHA-512 hash scheme to digest each SHA-512 message block consecutively.

Upon the completion, ippsHMACSHA512Update() updates the state variable IppsHMACSHA512 *pState with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure IppsHMACSHA512 *pState carries a new chaining state value with a possible partial message block for further message integration or padding.

Input Arguments

- pSrcMesg – message stream
- mesglen – length of message stream (in bytes)
- pState – the input chaining state

Output Arguments

- pState – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If mesglen < 0

HMACSHA512Final

Prototype

```
IppStatus ippsHMACSHA512Final(Ipp8u *pMAC, int maclen, IppsHMACSHA512
   *pState);
```

Description

ippsHMACSHA512Final() transforms the final chaining state carried by the input IppsHMACSHA512 *pState into an 512-bit hash value using the method specified by the SHA-512 hash scheme.

By concatenating the keyed-opad value with the generated hash value, this function further generates the message authentication code of the user specified length maclen (no more than 64 bytes).

Input Arguments

- maclen – the user specified message authentication code length.
- pState – the input chaining state

Output Arguments

pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - Either maclen < 1 or maclen > 64.

HMACSHA512MessageDigest

Prototype

```
IppStatus ippsHMACSHA512MessageDigest(const Ipp8u *pSrcMesg, int
   mesglen, const Ipp8u *pKey, int keylen, Ipp8u *pMAC, int maclen);
```

Description

ippsHMACSHA512MessageDigest() takes the input key of the user specified key length keylen (no less than 32 bytes) and applies the keyed SHA-512 hash based message authentication code scheme to transform the input message *pSrcMesg whose (byte) length is specified by input mesglen into its respective message authentication code Ipp8u *pMAC of user specified length maclen (no more than 64 bytes).

The following pseudocode represents this function:

pMAC = SHA512([pKey XOR opad]||[SHA512([pKey XOR ipad]||[pSrcMesg])])

where SHA512 represents the SHA-512 hash function, XOR is the bitwise exclusive-OR and || is the operation for concatenating two bit-streams.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream
- pKey – the user specified authentication key
- keylen – the key length (in unit of bytes)
- maclen – the user desired message authentication code length (in unit of bytes)

Output Arguments

- pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr - If any of the following occurs: (1) mesglen < 0, (2) keylen < 1, or (3) maclen < 1 or maclen > 64.

HMAC-MD5

HMAC-MD5 is the keyed MD5 hash based message authentication code scheme. HMAC-MD5 takes the input message with the user supplied key (no less than 8 bytes) to generate the respective message authentication code of the user specified length (no more than 16 bytes).

This section describes the set of five primitives that completes the operations required for generating the message authentication code using HMAC-MD5. All of the primitives described in this section are implemented to comply with the American Standard FIPS 198 and RFC 1321. The primitive ippsHMACMD5Init() sets up an initial chaining state and computes both the keyed-ipad and keyed-opad derived from the user supplied key. Once initialized, the primitive function ippsHMACMD5Update() applies MD5 to digest the input message stream till it exhausts all the MD5 message block. The function ippsHMACMD5Final() is designed to pad the partial message block into a final MD5 message block with the padding scheme specified in RFC 1321, it then uses the MD5 hash scheme to transform the final block into an 128-bit MD5 message digest value. By concatenating the key-opad with the hash value, this function further generates the message authentication code Ipp8u *pMAC of the user specified length maclen (no more than 16 bytes).

ThisIntel® IPP also provides a primitive ippsHMACMD5MessageDigest() to simplify the calling convention for generating the message authentication code with respect to the input message (non streaming data) as outlined above. This primitive takes the input message with the user supplied key and transforms it into an HMAC-MD5 message authentication code Ipp8u *pMAC of user specified length maclen (no more than 16 bytes).

Throughout this section, all the primitives employ a symbolic structure ippsHMACMD5 to serve as an operational vehicle which carries the information necessary to the operation, such as the keyed-ipad and keyed-opad values derived from the user input key, the bitlength of the processed message, a partial message block, and the intermediate hashing state values.

The application code for applying HMAC-MD5 to generate a message authentication code of the input should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the symbolic structure ippsHMACMD5 by calling the primitive ippsHMACMD5Buffersize().
2. Ensure that the required buffer is properly allocated and its size is no less than the one computed by the primitive ippsHMACMD5Buffersize(). With the allocated buffer, call the primitive ippsHMACMD5Init()to setup the initial chaining state and derives both keyed-ipad and keyed-opad values.
3. Keep calling the primitive ippsHMACMD5Update() to digest incoming message stream in the queue till its completion.
4. Call the primitive ippsHMACMD5Final()for processing the final partial block and with the concatenation of the keyed-opad value, it finally transforms it into the desired message authentication code.

5. Call the OS memory-free service function to release the buffer allocated for the symbolic structure ippsHMACMD5, if needed.

HMACMD5BufferSize

Prototype

```
IppStatus ippsHMACMD5BufferSize(int *pSize);
```

Description

ippsHMACMD5BufferSize() specifies the buffer size (in number of byes) required for defining a structured working buffer of symbolic structure IppsHMACMD5 to carry the information needed for generating an (up to 128 bits) message authentication code of the given input message.

Input Arguments

- none.

Output Arguments

- pSize– the size of required buffer (in bytes) for IppsHMACMD5.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – NULL pointers

HMACMD5Init

Prototype

```
IppStatus ippsHMACMD5Init(const Ipp8u *pKey, int keylen, IppsHMACMD5
   *pState);
```

Description

ippsHMACMD5Init() initializes the symbolic structure IppsHMACMD5 *pState, using user supplied buffer space pointed by the input *pState. In addition, ippsHMACMD5Init() sets up the initial chaining state IppsHMACMD5 *pState, computes both the keyed-ipad and the keyed-opad values derived from the user supplied key *pKey, and put the keyed-ipad values in a data container that is carried by the symbolic structure IppsHMACMD5 *pState for the concatenation with the incoming message stream.

Input Arguments

- pKey – pointer to the user supplied key byte stream
- keylen – key length (in unit of byte)
- pState – pointer to the user supplied buffer.

Output Arguments

- pState – the initialized symbolic structure IppsHMACMD5.

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr - If keylen < 1.

HMACMD5Update

Prototype

```
IppStatus ippsHMACMD5Update(const Ipp8u *pSrcMesg, int mesglen,
   IppsHMACMD5 *pState);
```

Description

ippsHMACMD5Update() digests the current input message stream const Ipp8u *pSrcMesg whose (byte) length is specified by the input int mesglen.

ippsHMACMD5Update() starts with integrating the previous partial block carried by the input state variable IppsHMACMD5 *pState with the input message stream *pSrcMesg and it then partitions them into multiple MD5 message blocks (each is 512-bit) with a possible additional partial block. This function then uses the RFC 1321 defined MD5 hash scheme to digest each MD5 message block consecutively.

Upon the completion, ippsHMACMD5Update() updates the state variable IppsHMACMD5 *pState with the newly generated chaining state and the bitlength of the processed message. At the end of the execution, the symbolic structure IppsHMACMD5 *pState carries a new chaining state value with a possible partial message block for further message integration or padding.

Input Arguments

- pSrcMesg – message stream
- mesglen – length of message stream (in bytes)
- pState – the input chaining state

Output Arguments

- pState – the updated chaining state

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr - If mesglen < 0

HMACMD5Final

Prototype

```
IppStatus ippsHMACMD5Final(Ipp8u *pMAC, int maclen, IppsHMACMD5
   *pState);
```

Description

ippsHMACMD5Final() transforms the final chaining state carried by the input IppsHMACMD5 *pState into an 128-bit hash value using the method specified by the MD5 scheme.

By concatenating the keyed-opad value with the generated hash value, this function further generates the message authentication code of the user specified length maclen (no more than 16 bytes).

Input Arguments

- maclen – the user specified message authentication code length.
- pState – the input chaining state

Output Arguments

- pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr - Either maclen < 1 or maclen > 16.

HMACMD5MessageDigest

Prototype

```
IppStatus ippsHMACMD5MessageDigest(const Ipp8u *pSrcMesg, int mesglen,
   const Ipp8u *pKey, int keylen, Ipp8u *pMAC, int maclen);
```

Description

ippsHMACMD5MessageDigest() takes the input key of the user specified key length keylen (no less than 8 bytes) and applies the keyed MD5 hash based message authentication code scheme to transform the input message *pSrcMesg whose (byte) length is specified by input mesglen into its respective message authentication code Ipp8u *pMAC of user specified length maclen (no more than 16 bytes).

The following pseudocode represents this function:

pMAC = MD5([pKey XOR opad]||[MD5([pKey XOR ipad]||[pSrcMesg])])

where MD5 represents the MD5 hash function, XOR is the bitwise exclusive-OR and || is the operation for concatenating two bit-streams.

Input Arguments

- pSrcMesg – input message stream
- mesglen – the (byte) length of message stream
- pKey – the user specified authentication key
- keylen – the key length (in unit of bytes)
- maclen – the user desired message authentication code length (in unit of bytes)

Output Arguments

- pMAC – the message authentication code

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr - If any of the following occurs: (1) mesglen < 0, (2) keylen < 1, or (3) maclen < 1 or maclen > 16.

Public Key Cryptographic Primitives

This section describes the Intel® IPP cryptographic primitives that can generate a pseudo-random number, a probable prime number, and other functions that meet the needs for public key cryptographic system design. In addition, this section also describes primitive functions for arithmetic and modular operations among big number integers of variable length. Furthermore, it describes the primitive functions for RSA cryptographic systems as well.

The primitives described in this section not only offer a complete set of highly optimized primitives for constructing a big number module-based public key cryptographic system, they also provide the following two key features to enhance security and key-size scalability:

- Security enhancement: The RSA cryptographic primitives operate exclusively on user supplied buffers to maintain domain separation for key generation, encryption and decryption operations. In addition, each primitive validates buffer usage and size to ensure buffer overflow is avoided.
- Cryptographic key size scalability: The RSA cryptographic primitives enable developers to balance their performance and memory usage by offering a flexible interface that supports both a binary method and a sliding-windows method. The binary method is provided for memory limited environments, while the sliding-windows method is offered for applications demanding the highest performance levels.

```
typedef enum {
    IppsBinaryMethod=0,
    IppsSlidingWindows=1,
} IppsExpMethod;
```

In the interest of simplicity and consistency, the mathematical expressions and pseudocode given in this section describes the behavior of each function.

This section describes the following primitives:

- “Big Number Arithmetic”.
- “Montgomery Reduction Scheme”.

- "Pseudo-Random Number Generation"
- "Prime Number Generation"
- "RSA Cryptographic Primitives".
- "DSA Digital Signature Primitives"

Big Number Arithmetic

This section describes the primitives available for performing arithmetic operations among big number integers of variable length. Throughout this section, the value of a big number integer specified by an array `Ipp32u rp[length]` is defined as

$$r = \sum_{0 \le i < length} rp[i] \times 2^{32i}$$

and `length` is defined as the length of the big number integer specified by the array `rp`. This section also uses the following definition to define the sign of a big number integer:

```
typedef enum {
     IppsBigNumNEG=0,
     IppsBigNumPOS=1
} IppsBigNumSGN;
```

Furthermore, the primitives in this section employ a symbolic structure `IppsBigNum` to serve as an operational vehicle, which carries not only the sign and value of the data, but also the working buffer reserved sufficiently for various arithmetic operations that it may involve. The length of the symbolic structure `IppsBigNum` is defined as the data length of the big number integer carried by the structure. The size of the symbolic structure `IppsBigNum` is therefore defined as the maximum data length of such a big number integer that this operational vehicle could carry.

A completed primitive function list for performing arithmetic operations operated on the big number integers, for both signed or unsigned, is detailed in the following subsections. This list encompasses all of the necessary functions required by the modulo based public key cryptographic system design.

Add_BNU

Prototype

```
IppStatus ippsAdd_BNU (const Ipp32u *a, const Ipp32u *b, Ipp32u *r, int n,
    Ipp32u *carry);
```

Description

`ippsAdd_BNU()` adds two unsigned big number integers of same length pointed by `const Ipp32u *a` and `*b`, the result of the operation is returned as `Ipp32u *r` with a possible carry `Ipp32u *carry`.

All integers pointed by `a`, `b` and `r` are of (`n*32`)-bit long.

The following pseudocode represents this function:

```
(carry|(*r)) <- (*a) + (*b);
```

Input Arguments

- `a` – an unsigned big number integer of `n*32` bits
- `b` – an unsigned big number integer of `n*32` bits
- `n` – the length (in number of 32-bit words) specified for inputs `a` and `b`, as well as the result `r`
- `r` – the result of the addition
- `carry` – the carry of the addition

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `n < or = 0`

Sub_BNU

Prototype

```
IppStatus ippsSub_BNU(const Ipp32u *a, const Ipp32u *b, Ipp32u *r, int n,
   Ipp32u *carry);
```

Description

`ippsSub_BNU()` performs subtraction between two unsigned big numbers of same length. It subtracts a big-number integer pointed by `const Ipp32u *a` from a big-number integer pointed by `const Ipp32u *b`, the result of the operation is a return to the memory pointed by `Ipp32u *r` with a possible borrow returned as`Ipp32u *carry`.

Integers pointed by `a`, `b` and `r` are all of (n*32)-bit.

The following pseudocode represents this function:

```
(*r) <- (*a) - (*b);
carry = ((*a) < (*b));
```

Input Arguments

- `a` – a big number integer of `n*32` bits
- `b` – a big number integer of `n*32` bits
- `n` – length (in number of 32-bit words) specified for inputs `a` and `b`, as well as the result `r`

Output Arguments

- `r` – the result of the subtraction.
- `carry` – the borrow of the subtraction

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `n < or = 0`

MulOne_BNU

Prototype

```
IppStatus ippsMulOne_BNU(const Ipp32u *a, Ipp32u *r, int n, Ipp32u w,
   Ipp32u *carry);
```

Description

`ippsMulOne_BNU()` computes the multiplication of an unsigned big-number integer pointed by `const Ipp32u *a` with a 32-bit unsigned integer `Ipp32u w`. The result of the operation is returned to a same length array pointed by `*r`, and the carry of the result is returned at `Ipp32u *carry`.

The following pseudocode represents this function:

```
(carry|*r) <- (*a) w
```

NOTE. *Integers pointed by both unsigned* `long *a` *and* `*r` *are of* (`n*32`)*-bit long.*

Input Arguments

- `a` – an unsigned big number integer serving as a multiplicand.
- `n` – length (in 32-bit words) of big-number integers `r` and `a`
- `w` – a 32-bit unsigned long integer serving as multiplier for the operation

Output Arguments

- `r` – the result held in an array of the same size as that of the input array `a`
- `carry` – the carry of the operation

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `n < or = 0`

MACOne_BNU_I

Prototype

```
IppStatus ippsMACOne_BNU_I (const Ipp32u *a, Ipp32u *r, int n, Ipp32u w,
   Ipp32u *carry);
```

Description

`ippsMACOne_BNU_I()` computes the multiplication of an unsigned big-number integer pointed by `const Ipp32u *a` with a 32-bit integer `Ipp32u w` and accumulates the result with an another big number integer of same length pointed by `Ipp32u *r`. The result of the operation is returned to `*r`, and the carry of the result is returned at `Ipp32u *carry`.

The following pseudocode represents this function:

```
(carry|*r) <- (*r) + (*a) w
```

NOTE. *Integers pointed by both unsigned* `long *a` *and* `*r` *are of* (`n*32`)*-bit long.*

Input Arguments

- `a` – an unsigned big number integer multiplicand
- `r` – an unsigned big number integer accumulator
- `n` – length of big number integers `r` and `a`
- `w` – a 32-bit unsigned long integer multiplier

Output Arguments

- `r` – the result
- `carry` – the carry of the operation

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

- ippStsLengthErr – if n < or = 0

Mul_BNU4

Prototype

```
IppStatus ippsMul_BNU4(const Ipp32u *a, const Ipp32u *b, Ipp32u *r);
```

Description

ippsMul_BNU4() computes the multiplication between two unsigned integers of 4*32-bit pointed by const Ipp32u *a and *b, and returns the result in the memory pointed by Ipp32u *r.

The *r argument should be a buffer sufficiently large to hold an 8*32-bit result.

The following pseudocode represents this function:

```
(*r) <- (*a)(*b)
```

Input Arguments

- a – a multiplicand of 4*32-bit
- b – a multiplier of 4*32-bit.

Output Arguments

- r – the multiplication result of 8*32-bit

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers

Mul_BNU8

Prototype

```
IppStatus ippsMul_BNU8(const Ipp32u *a, const Ipp32u *b, Ipp32u *r);
```

Description

This function computes the multiplication between two unsigned integers of 8*32-bit pointed by `const Ipp32u *a` and `*b`, and stores the result in the memory pointed by `Ipp32u *r`.

The `*r` argument should be a buffer sufficiently large to hold a 16*32-bit result.

The following pseudocode represents this function:

```
(*r) <- (*a) (*b)
```

Input Arguments

- a – a multiplicand of 8*32-bit
- b – a multiplier of 8*32-bit

Output Arguments

- r – the multiplication result of 16*32-bit

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr - NULL pointers

Div_64u32u

Prototype

```
IppStatus ippsDiv_64u32u(Ipp64u a, Ipp32u b, Ipp32u *q, Ipp32u *r);
```

Description

`ippsDiv_64u32u()` computes an integer division for a 64-bit unsigned integer dividend `Ipp64u a` by a 32-bit unsigned divisor `Ipp32u b`, it returns the quotient at `Ipp32u *q` and the remainder at `Ipp32u *r`. This function executes only under the condition that the quotient result is a 32-bit data.

The following pseudocode represents this function:

```
a = b*q + r
```

Input Arguments

- `a` – the 64-bit dividend
- `b` – the 32-bit divisor

Output Arguments

- `q` – the 32-bit quotient
- `r` – the 32-bit remainder

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsOutOfRangeErr` – if the quotient is not a 32-bit data
- `ippStsDivByZeroErr` – zero divisor

Sqr_32u64u

Prototype

```
IppStatus ippsSqr_32u64u(const Ipp32u *src, int n, Ipp64u *dst);
```

Description

`ippsSqr_32u64u()` computes the square of each unsigned long word (32-bit) within the input array pointed by `const Ipp32u *src`. The result of the operation is an array of 64-bit unsigned integers that are returned to the memory pointed by `Ipp64u *dst`.

(`*src`) is the pointer to the array of (n*32)-bit, and the result (`*dst`) is the pointer to the array of (n*64)-bit.

The following pseudocode represents this function:

```
dst[i] <- src[i] * src[i], i= 0, 1, 2, ..., n-1.
```

Input Arguments

- `src` – array of 32-bit words
- `n` – length of the input array

Output Arguments

- `dst` – pointer to the result

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `n < or = 0`.

Sqr_BNU4

Prototype

```
IppStatus ippsSqr_BNU4(const Ipp32u *a, Ipp32u *r);
```

Description

`ippsSqr_BNU4()` computes the square for an unsigned big number integer of 4*32-bit pointed by `const Ipp32u *a`, and stores the result in the memory pointed by `Ipp32u *r`.

The `*r` argument should be a buffer sufficiently large to hold an 8*32-bit result.

The following pseudocode represents this function:

```
(*r) <- (*a) (*a)
```

Input Arguments

- `a` – a multiplicand of 4*32-bit

Output Arguments

- `r` – the square operation result of 8*32-bit

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

Sqr_BNU8

Prototype

```
IppStatus ippsSqr_BNU8(const Ipp32u *a, Ipp32u *r);
```

Description

`ippsSqr_BNU8()` computes the square for an unsigned big number integer of 8*32-bit pointed by `const Ipp32u *a`, and stores the result in the memory pointed by `Ipp32u *r`.

The `*r` argument should be a buffer sufficiently large to hold a 16*32-bit result.

The following pseudocode represents this function:

```
(*r) <- (*a) (*a)
```

Input Arguments

- `a` – a multiplicand of 8*32-bit

Output Arguments

- `r` – the square operation result of 16*32-bit

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

BigNumBufferSize

Prototype

```
IppStatus ippsBigNumBufferSize(int length, int *size);
```

Description

`ippsBigNumBufferSize()` specifies the buffer size (in number of bytes) required for defining a symbolic structure `IppsBigNum` for both the storage and operations on a big number integer with length number of `Ipp32u`.

Input Arguments

- `length` – the length of big number integer

Output Arguments

- `size` – the size of required buffer (in bytes) for initializing respective `IppsBigNum`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `length < or = 0`

BigNumInit

Prototype

```
IppStatus ippsBigNumInit(int length, IppsBigNum *b);
```

Description

ippsBigNumInit() initializes the symbolic structure IppsBigNum *b, using a user-supplied buffer space pointed by the input *b. It partitions the given buffer in such a way that is able to store and to execute various arithmetic operations on a big number integer with length number of Ipp32u.

Input Arguments

- length – the size (the maximum data length) of the symbolic structure IppsBigNum
- b – pointer to the user supplied buffer

Output Arguments

- b – initialized symbolic structure IppsBigNum

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr – if length < or = 0

CmpZero_BN

Prototype

```
IppStatus ippsCmpZero_BN(const IppsBigNum *b, Ipp32u *result);
```

Description

ippsCmpZero_BN() scans the data field of the input const IppsBigNum *b and report IS_ZERO, if the value held by IppsBigNum *b is zero. In case of nonzero, it reports GREATER_THAN_ZERO if the input is greater than zero or LESS_THAN_ZERO if the input is less than zero.

Input Arguments

- b – a big number integer of data type IppsBigNum

Output Arguments

- `result` – the nature (zero/positive/negative) of the input big number integer

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

GetSize_BN

Prototype

```
IppStatus ippsGetSize_BN(const IppsBigNum *b, int *size);
```

Description

`ippsGetSize_BN()` evaluates the working buffer assigned to the symbolic structure `IppsBigNum *b`, and it returns `int *size`, the size of the structure, to report the maximum length of the big number integer that `IppsBigNum *b` can carry.

Input Arguments

- `b` – a big number integer of the symbolic structure`IppsBigNum`

Output Arguments

- `size` – the size of the structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

Set_BN

Prototype

```
IppStatus ippsSet_BN (IppsBigNumSGN sgn, int length, const Ipp32u *data,
   IppsBigNum *b);
```

Description

`ippsSet_BN()` defines the sign and value for `IppsBigNum *b` with user supplied inputs `IppsBigNumSGN sgn` and `const Ipp32u *data` that holds an array of data (`length` number of `Ipp32u`).

Input Arguments

- `sgn` – sign of `IppsBigNum *b`
- `length` – array length of input `const Ipp32u *data`
- `data` – array of data

Output Arguments

- `b` – the symbolic structure `IppsBigNum` loaded with input data

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `length < or = 0`
- `ippStsOutOfRangeErr` – if `length` is larger than the size of the `IppsBigNum *b`

Get_BN

Prototype

```
IppStatus ippsGet_BN  (IppsBigNumSGN *sgn, int *length, Ipp32u *data,
   const IppsBigNum *b);
```

Description

`ippsGet_BN()` extracts the sign and value of the big number integer from the input `IppsBigNum *b`.

Input Arguments

- `b` – the big number integer of symbolic structure `IppsBigNum`

Output Arguments

- `sgn` – sign of `IppsBigNum *b`
- `length` – data length
- `data` – array of data

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

Add_BN

Prototype

```
IppStatus ippsAdd_BN(IppsBigNum *a, IppsBigNum *b, IppsBigNum *r);
```

Description

ippsAdd_BN() adds two big number integers IppsBigNum *a and IppsBigNum *b, regardless of their signs and sizes, and returns the result of the operation to IppsBigNum *r. It executes only under the condition that size of IppsBigNum *r is no less than either length of IppsBigNum *a and IppsBigNum *b.

The following pseudocode represents this function:

```
(*r) <- (*a) + (*b);
```

Input Arguments

- a – a big number integer of the symbolic structure IppsBigNum
- b – a big number integer of the symbolic structure IppsBigNum

Output Arguments

- r – the addition result of the symbolic structure IppsBigNum

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsOutOfRangeErr – The size of IppsBigNum *r is smaller than the result data length

Sub_BN

Prototype

```
IppStatus ippsSub_BN(IppsBigNum *a, IppsBigNum *b, IppsBigNum *r);
```

Description

ippsSub_BN() subtracts a big number integer IppsBigNum *a from the big number integer IppsBigNum *b, regardless of their signs and sizes, the result of the operation is returned to IppsBigNum *r. It executes only under the condition that size of IppsBigNum *r is no less than either length of IppsBigNum *a and IppsBigNum *b.

The following pseudocode represents this function:

```
(*r) <- (*a) - (*b);
```

Input Arguments

- a – a big number integer of the symbolic structure IppsBigNum
- b – a big number integer of the symbolic structure IppsBigNum

Output Arguments

- r – the subtraction result of the symbolic structure IppsBigNum

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsOutOfRangeErr – The size of IppsBigNum *r is smaller than the result data length

Mul_BN

Prototype

```
IppStatus ippsMul_BN(IppsBigNum *a, IppsBigNum *b, IppsBigNum *r);
```

Description

ippsMul_BN() multiplies a big number integer pointed by IppsBigNum *a with another big number integer IppsBigNum *b, regardless of their signs and sizes, the result of the operation is returned to IppsBigNum *r. It executes only under the condition that size of IppsBigNum *r is no less than the sum of lengths of IppsBigNum *a and IppsBigNum *b minus one.

The following pseudocode represents this function:

```
r <- a * b;
```

Input Arguments

- a – a multiplicand of IppsBigNum
- b – a multiplier of IppsBigNum

Output Arguments

- r – the multiplication result of IppsBigNum

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsOutOfRangeErr – The size of IppsBigNum *r is smaller than the result data length

MAC_BN_I

Prototype

```
IppStatus ippsMAC_BN_I(IppsBigNum *a, IppsBigNum *b, IppsBigNum *r);
```

Description

ippsMAC_BN_I() multiplies a big number integer pointed by IppsBigNum *a with another big number integer IppsBigNum *b, and accumulates the result with the third input big number integer pointed by IppsBigNum *r, regardless of their signs and sizes. The result of the operation is returned to IppsBigNum *r. It executes only under the condition that size of IppsBigNum *r is no less than the sum of lengths of IppsBigNum *a and IppsBigNum *b minus one.

The following pseudocode represents this function:

```
r <- r + a * b;
```

Input Arguments

- a – a multiplicand of IppsBigNum
- b – a multiplier of IppsBigNum
- r – accumulator of IppsBigNum

Output Arguments

- r – the accumulation result of IppsBigNum

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsOutOfRangeErr – The size of IppsBigNum *r is smaller than the result data length

Div_BN

Prototype

```
IppStatus ippsDiv_BN(IppsBigNum *a, IppsBigNum *b, IppsBigNum *q, IppsBigNum *r);
```

Description

`ippsDiv_BN()` computes a division for a big number integer dividend pointed by `IppsBigNum *a` to be divided by another big number integer divisor `IppsBigNum *b`, regardless of their signs and sizes. The quotient of the division is returned to `IppsBigNum *q`, and the respective remainder is returned to `IppsBigNum *r`.

It requires that the size of `IppsBigNum *q` should be no less than `(length of *a)- (length of *b)`, and the size of `IppsBigNum *r` should be no less than the length of `IppsBigNum *b`.

The following pseudocode represents this function:

```
q <- a / b;
r <- a - b*q;
```

Input Arguments

- `a` – a dividend of `IppsBigNum`
- `b` – a divisor of `IppsBigNum`

Output Arguments

- `q` – the quotient of `IppsBigNum`
- `r` – the remainder of `IppsBigNum`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsOutOfRangeErr` – if either the size of `IppsBigNum *q` is smaller than the quotient result data length, or the size of `IppsBigNum *r` is smaller than the length of `IppsBigNum *b`
- `ippStsDivByZeroErr` – zero divisor

Mod_BN

Prototype

```
IppStatus ippsMod_BN(IppsBigNum *a, IppsBigNum *m, IppsBigNum *r);
```

Description

`ippsMod_BN()` computes the modular reduction for an input big number integer pointed by `IppsBigNum *a` with respect to the modulus specified by a positive big number integer `IppsBigNum *m`. The modular reduction result is returned to `IppsBigNum *r` in the range of `[0, (m-1)]`.

It requires that the size of `IppsBigNum *r` should be no less than the length of `IppsBigNum *m`.

The following pseudocode represents this function:

```
r <- a mod m;
```

Input Arguments

- `a` – a big number integer of `IppsBigNum`
- `m` – a modulus integer of `IppsBigNum`

Output Arguments

- `r` – the modular reduction result

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsOutOfRangeErr` – if size of `IppsBigNum *r` is less than the length of `IppsBigNum *m`
- `ippStsBadModulusErr` – if modulus `IppsBigNum *m` is not an positive integer

Gcd_BN

Prototype

```
IppStatus ippsGcd_BN(IppsBigNum *a, IppsBigNum *b, IppsBigNum *g);
```

Description

`ippsGcd_BN()` computes the greatest common divisor (gcd) of two big number integers pointed by `IppsBigNum *a`, and `IppsBigNum *b` respectively. It executes only under the condition that the size of `IppsBigNum *g` should be no less than either length of `IppsBigNum *a` and `IppsBigNum *b`.

The following pseudocode represents this function:

```
g <- gcd(a, b);
```

Input Arguments

- `a` – a big number integer of `IppsBigNum`
- `b` – the second input big number integer of `IppsBigNum`

Output Arguments

- `g` – the greatest common divisor of `a` and `b`

Returns

- `ippStsNullPtrErr` – NULL pointers
- `ippStsOutOfRangeErr` – if size of `IppsBigNum *g` is less than either length of `IppsBigNum *a` and `IppsBigNum *b`

ModInv_BN

Prototype

```
IppStatus ippsModInv_BN(IppsBigNum *e, IppsBigNum *m, IppsBigNum *d);
```

Description

ippsModInv_BN() uses the extended Euclidean algorithm to compute the multiplicative inverse of a given positive big number integer pointed by IppsBigNum *e with respect to the modulus specified by a positive big number integer IppsBigNum *m, where gcd(e, m) = 1.

It requires that the size of IppsBigNum *d should be no less than the length of IppsBigNum *m.

The following pseudocode represents this function:

```
compute d, such that d*e = 1 mod m;
```

Input Arguments

- e – a big number integer of IppsBigNum
- m – modulus integer of IppsBigNum

Output Arguments

- d – the multiplicative inverse

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – if e < or = 0
- ippStsNullPtrErr – NULL pointers
- ippStsOutOfRangeErr – if size of IppsBigNum *d is less than the length of IppsBigNum *m
- ippStsBadModulusErr – if e > m, or gcd(e, m)>1, or m < or = 0

Montgomery Reduction Scheme

Montgomery reduction is a technique which allows an efficient implementation of modular multiplication without explicitly carrying out the classical modular reduction step.

This section describes the primitives available for Montgomery modular reduction, Montgomery modular multiplication as well as Montgomery modular exponentiation. Let n be a positive integer, and let R and T be integers such that R>n, gcd(n, R)=1, and 0<T<nR. The Montgomery reduction T of modulo n with respect to R is defined as $TR^{-1} \bmod n$. To fully take advantage of the Intel® IPP, Intel® IPP chooses $R=b^k$ where $b=2^{32}$ and k to be the integer computed by the ceiling function of bit length of the integer n over 32.

Throughout this section, all the primitives employ a symbolic structure IppsMont to serve as an operational vehicle which carries the Montgomery reduction index k, the big number integer module n, the lease significant word n0 of the multiplicative inverse of the module n with respect to the Montgomery reduction factor R, as well as the working buffer reserved sufficiently for various Montgomery modular operations.

Furthermore, two new terms are introduced in this section. The length of the symbolic structure IppsMont is defined as the data length of the modulus n carried by the structure; the size of the symbolic structure IppsMont is therefore defined as the maximum data length of such an integer modulus n that could be carried by this operational vehicle.

To briefly illustrate the procedure of using the primitives in this section for computing a classical modular exponentiation $T=x^e \bmod n$, we start with a simple example. Consider computing $T=x^4 \bmod n$, for some integer x with $0<x<n$. First get the buffer size required to configure the symbolic structure ippsMont by calling ippsMontBuffersize() and then allocate the working buffer using OS service function, with allocated buffer to call ippsMontInit() as to initialize the symbolic structure IppsMont.

Set the module n by calling ippsMontSet() and then convert x into its respective Montgomery form by calling ippsMontForm() – that is, computing $\underline{x}=xR \bmod n$. Then compute the Montgomery reduction of xx using the Intel® IPP primitive ippsMontMul() to generate $T=\underline{xx}R^{-1} \bmod n$. The Montgomery reduction of $T*T \bmod n$ with respect to R is $T^2R^{-1} \bmod n = (x^2R^{-1})^2R^{-1} \bmod n = x^4R \bmod n$. Further applying ippspMontMul() with this value and the value of 1 yields the desired result $T=x^4 \bmod n$.

The classical modular exponentiation should be computed by performing the following sequence of operations:

1. Get the buffer size required to configure the symbolic structure ippsMont by calling the primitive ippsMontBuffersize(). For a limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer size significantly, while using the sliding window method enhances the performance.
2. Allocate working buffer through an OS memory allocation function and configure the structure ippsMont by calling the primitive ippsMontInit() with the allocated buffer and the choice made on the modular exponential method at the time of invoking ippsMontBuffersize().
3. Call the primitive ippsMontSet() to set the big number integer module for ippsMont.
4. Call the primitive ippsMontForm() to convert the integer x to be its Montgomery form.
5. Call the primitive ippspMontExp() to compute the Montgomery modular exponentiation.
6. Call the primitive ippspMontMul() to compute the Montgomery modular multiplication of the above result with the integer 1 to convert the above result back to the desired classical modular exponential result.
7. Free the memory using an OS memory-free function if needed.

MontBufferSize

Prototype

```
IppStatus ippsMontBufferSize(IppsExpMethod method, int length, int *size);
```

Description

`ippsMontBufferSize()` specifies the buffer size (in number of bytes) required for defining a structured working buffer of the symbolic structure `IppsMont` for both the storage of module (`length` number of `Ipp32u`) and the operations using various Montgomery module schemes.

`ippsMontBufferSize()` returns the required buffer size `int *size` which is also based on the exponential method of user's choice. Using binary method reduces buffer size significantly, while using sliding windows method enhances the performance.

Input Arguments

- `method` – specifies the exponential method of user's choice
- `length` – the data length for underlined module

Output Arguments

- `size` – the size of required buffer (in bytes) for initializing respective `IppsMont`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `length < or = 0`

MontInit

Prototype

```
IppStatus ippsMontInit(IppsExpMethod method, int length, IppsMont *m);
```

Description

`ippsMontInit()` initializes the symbolic data type `IppsMont *m`, using a user supplied buffer pointed by the input `*m`. Based on the modular exponential method of users choice specified by the input `IppsExpMethod method`, `ippsMontInit()` partitions the given buffer in such a way that is able to carry up to `length*size of(Ipp32u)`-bit big number module and to execute its associated various Montgomery module operations.

Input Arguments

- `method` – specifies the exponential method of user's choice
- `length` – the data field length for underlined module
- `m` – pointer to the user supplied buffer

Output Arguments

- `m` – the initialized symbolic data structure `IppsMont`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `length < or = 0`

MontSet

Prototype

```
IppStatus ippsMontSet(const Ipp32u *n, int length, IppsMont *m);
```

Description

`ippsMontSet()` sets the input positive big odd number integer (`const Ipp32u *n`) to be the modulus for the symbolic structure `IppsMont *m`. In addition, it computes the Montgomery reduction index `k` with respect to the input big number module `n` and the least significant 32-bit word of the multiplicative inverse `Ni` with respect to the modulus `R`, that satisfies $R*R^{-1}-n*Ni=1$.

Input Arguments

- `n` – the data field of input big number modulus

- length – the data length

Output Arguments

- m – the symbolic structure IppsMont loaded with the input modulus and its associated the least significant word of the multiplicative inverse Ni.

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr – if length < or = 0
- ippStsOutOfRangeErr – if length is larger than the size of the structure IppsMont *m
- ippStsBadModulusErr – if modulus carried by Ipp32u *n is not a positive odd integer

MontGet

Prototype

```
IppStatus ippsMontGet(Ipp32u *n, int *length, const IppsMont *m);
```

Description

ippsMontGet() extracts the big number modulus (Ipp32u *n, int *length) from the input IppsMont *m.

Input Arguments

- m – the symbolic structure IppsMont

Output Arguments

- n – the modulus data field
- length – the modulus data length

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers

MontForm

Prototype

```
IppStatus ippsMontForm(IppsBigNum *a, IppsMont *m, IppsBigNum *r);
```

Description

`ippsMontForm()` converts an input positive big number integer pointed by `IppsBigNum *a` into its Montgomery form `IppsBigNum *r` with respect to the big number module `IppsMont *m`.

This function requires that the size of `IppsBigNum *r` should be no less than the data length of the underlined module `m`.

The following pseudocode represents this function:

```
r <- a*R mod m
```

Input Arguments

- `a` – an input big number integer within the range `[0,m-1]`
- `m` – an input module of symbolic structure `IppsMont`

Output Arguments

- `r` – the result Montgomery form `r = a*R mod m`

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – if `a` is a negative integer
- `ippStsNullPtrErr` – NULL pointers
- `ippStsScaleRangeErr` – if `a>m`
- `ippStsOutOfRangeErr` – if the size of `IppsBigNum *r` is less than the size of `IppsMont m`

MontMul

Prototype

```
IppStatus ippsMontMul(IppsBigNum *a, IppsBigNum *b, IppsMont *m, IppsBigNum *r);
```

Description

`ippsMontMul()` computes the Montgomery modular multiplication `IppsBigNum *r` for positive big number integers of Montgomery form pointed by `IppsBigNum *a` and `IppsBigNum *b` with respect to the module `ippsMont *m`.

This function requires that the size of `IppsBigNum *r` should be no less than the data length of the underlined module `m`.

The following pseudocode represents this function:

$r <- a*b*R^{-1} \bmod m$

Input Arguments

- `a` – multiplicand within the range `[0,m-1]`
- `b` – multiplier within the range `[0,m-1]`
- `m` – modulus

Output Arguments

- `r` – the Montgomery multiplication result

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – if either `a` or `b` is a negative integer
- `ippStsNullPtrErr` – NULL pointers
- `ippStsScaleRangeErr` – if `a>m` or `b>m`
- `ippStsOutOfRangeErr` – if the size of `IppsBigNum *r` is less than the size of `IppsMont *m`

MontExp

Prototype

```
IppStatus ippsMontExp(IppsBigNum *a, IppsBigNum *e, IppsMont *m, IppsBigNum *r);
```

Description

`ippsMontExp()` computes the Montgomery exponentiation with the exponent specified by an input positive big number integer `IppsBigNum *e` to the given positive big number integer of Montgomery form `IppsBigNum *a` with respect to the modulus `IppsMont *m`.

This function requires that the size of `IppsBigNum *r` should be no less than the length of the underlined big number integer module `m`.

The following pseudocode represents this function:

$r \leftarrow a^{e}R^{-(e-1)} \bmod m$

Input Arguments

- `a` – big number integer of Montgomery form within the range `[0,m-1]`
- `e` – big number exponent
- `m` – Montgomery modulus of `IppsMont`

Output Arguments

- `r` – the Montgomery exponentiation result

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – if `a` or `e` is a negative integer
- `ippStsNullPtrErr` – NULL pointers
- `ippStsScaleRangeErr` – if `a>m` or `e>m`
- `ippStsOutOfRangeErr` – if the size of `IppsBigNum *r` is less than the size of `IppsMont *m`

Pseudo-Random Number Generation

Pseudo-random number generation functions are the important primitives in many cryptographic systems design. The unpredictable nature of the cipher system inherited from a pseudo-random number generator is the security foundation for protecting the communication over open channels against potential adversaries.

This section describes the set of primitives that comprises a pseudo-random bit sequence generator implemented by a U.S. FIPS-approved method. The pseudo-random bit sequences are generated in a way that is based on the U.S. FIPS 186-2. The sampling of the random bit sequence is determined by the generation state, which is initiated by the user's choice on a real time random seed and this generator also takes a context bit string input for required entropy. Intel® IPP offers primitives for both initialized and direct versions of pseudo-random number generators. A direct generator requires only a single call to a primitive function `ippsPRNGGen()` in order to obtain a desired pseudo random bit sequence with desired top and bottom bit patterns.

On the other hand, the initialized version requires the user to call an initialization service function `ippsPRNGInit()` to initialize the structure `ippsPRNG` with the user-supplied buffer. The user then must call `ippsPRNGSetSeed()` to set up a 160-bit prime number and the initial state of the desired pseudo-random number generation process with the user-supplied new, secret value for the seed key (XKEY). The user can also use `ippsPRNGSetPrimeQ()` to set the probable prime q of choice. Once both the seed and prime q are set, the user can use the `ippsPRNGAdd()` function with the supplied context bit sequence serving as optional input for XSEED for the pseudo-random number generation.

Throughout this section, all the primitives employ a symbolic structure `IppsPRNG` to serve as an operational vehicle which carries the target bit length of the desired pseudo-random bit sequence, the current bit length of the generated pseudo-random bit sequence, the generation state specified as XKEY by FIPS 186-2, the primeq and the integer b, as well as the data container that holds the generated pseudo-random bit sequence.

The application code for generating a sequence of pseudo-random bits using the primitives described in this section should perform the following sequence of operations:

1. Get the buffer size required to configure the symbolic structure `ippsPRNG` by calling the primitive `ippsPRNGBuffersize()`.
2. Allocate a working buffer through an OS memory allocation function and configure the structure `ippsPRNG` by calling the primitive `ippsPRNGInit()` with allocated buffer and set the initial state by calling the primitive `ippsPRNGSetSeed()`.
3. Keep calling the primitive `ippsPRNGAdd()` to add entropy bits of desired context bit string c until `IppsPRNG *r` holds the generated pseudo random bit sequence with desired bit length.
4. Extract the generated pseudo-random bit sequence by calling the primitive `ippsPRNGGetRand()`.

5. Free the memory allocated to structure `IppsPRNG` by calling the OS memory-free service function if needed.

PRNGBufferSize

Prototype

```
IppStatus ippsPRNGBufferSize(int bitlength, int *size);
```

Description

`ippsPRNGBufferSize()` specifies the buffer size (in number of bytes) required for defining a structured working buffer of symbolic data type `IppsPRNG` for both generating and storing of a pseudo-random bit sequence of specified bit length (`int bitlength`).

Input Arguments

- `bitlength` – the target bit length of a pseudo-random bit sequence

Output Arguments

- `size` – the size of required buffer (in bytes) for initializing respective `IppsPRNG`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `bitlength < or = 0`

PRNGInit

Prototype

```
IppStatus ippsPRNGInit(int bitlength, IppsPRNG *r);
```

Description

ippsPRNGInit() defines the symbolic data type IppsPRNG *r, by organizing the user supplied buffer pointed by the input *r. ippsPRNGInit() partitions the given buffer in such a way that is able to generate and to store a pseudo-random bit sequence of specified bit length (int bitlength). Once the symbolic structure IppsPRNG *r is established, ippsPRNGInit() sets up a 160-bit default prime number.

Input Arguments

- bitlength – bit length of the target pseudo random bit sequence
- r – pointer to the user supplied buffer

Output Arguments

- r – the initialized symbolic structure IppsPRNG

Returns

- ippStsNoErr – no error
- ippStsNullPtrErr – NULL pointers
- ippStsLengthErr – if bitlength < or = 0

PRNGSetSeed

Prototype

```
IppStatus ippsPRNGSetSeed(Ipp32u *seed, IppsPRNG *r);
```

Description

ippsPRNGSetSeed() resets the FIPS 186-2 specified pseudo random number generation process by resetting the seed-key, XKEY with user supplied Ipp32u *seed.

Input Arguments

- seed – 160-bit seed value; places generator into a known state

Output Arguments

- r – the symbolic structure IppsPRNG reset its state with user supplied seed

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

PRNGSetPrimeQ

Prototype

```
IppStatus ippsPRNGSetPrimeQ(Ipp32u *q, IppsPRNG *r);
```

Description

`ippsPRNGSetPrimeQ()` sets the probable prime q for the pseudo-random number generation as specified in the FIPS 186-2 with user supplied probable prime `Ipp32u *q`, under the condition that the supplied probable prime passes a minimum prime test.

Input Arguments

- `q` – 160-bit probable prime number

Output Arguments

- `r` – the symbolic structure `IppsPRNG` reset its state with user supplied seed

Returns

- `ippStsNoErr` – no error/accept input prime q
- `ippStsNullPtrErr` – NULL pointers
- `ippStsBadArgErr` – either `q` is not full/exact 160-bit value, or `q` fails a minimum prime test

PRNGAdd

Prototype

```
IppStatus ippsPRNGAdd(IppsBigNum *c, int bitlength, IppsPRNG *r);
```

Description

Continuing the generation process with the state carried by the symbolic structure `IppsPRNG *r`, `ippsPRNGAdd()` generates additional pseudo random bits of specified bit length (`int bitlength`). This raises the current bit length of the pseudo-random bit sequence held by the `IppsPRNG *r`. To do this, `ippsPRNGAdd()` employs the process specified by FIPS 186-2, with the input context bit string held by `IppsBigNum *c`.

Input Arguments

- `c` – a `b`-bit content string, with $160 \le b \le 512$
- `bitlength` – additional bit length adding to the existing random bit sequence held by `r`
- `r` – an input `IppsPRNG` with existing pseudo random bit sequence

Output Arguments

- `r` – the result `IppsPRNG` with an expanded pseudo-random bit sequence
- `ippStsNoErr` – no error
- `ippStsBadArgErr` – if `b < 160`
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `bitlength < or = 0`

PRNGGen

Prototype

```
IppStatus ippsPRNGGen (Ipp32u *seed, IppsBigNum *c, int bitlength,
   Ipp32u top, Ipp32u bottom, IppsPRNG *r);
```

Description

`ippsPRNGGen()` sets up an initial state using the input 160-bit `*seed` for initializing the pseudo-random number generation process. It then generates a pseudo random bit sequence of specified bit length (`int bitlength`) using the symbolic structure `IppsPRNG *r`. To do this, `ippsPRNGGen()` employs the process defined by FIPS-186, with the input context bit string held by `IppsBigNum *c`. Once the bit sequence generated, it sets the top and bottom bit patterns as specified by `Ipp32u top` and `Ipp32u bottom` respectively.

However, `ippsPRNGGen()` does not set the top bits if the input `top==0`. In similar convention, it does not set its bottom bit pattern either, if the input `bottom==0`.

Input Arguments

- `seed` – 160-bit seed value; places generator into a known state
- `c` – a `b`-bit content string, with $160 \le b \le 512$
- `bitlength` – the target bit length of the result pseudo random bit sequence
- `top` – top bits pattern setting
- `bottom` – bottom bits pattern setting

Output Arguments

- `r` – the generated pseudo random bit sequence

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – if `b < 160`
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `bitlength < or = 0`

PRNGGetRand

Prototype

```
IppStatus ippsPRNGGetRand(int *curbitlength, Ipp32u *rseq, const IppsPRNG *r);
```

Description

`ippsPRNGGetRand()` extracts the current bit length (`int *curbitlength`) and the pseudo-random bit sequence (`Ipp32u *rseq`) from the input `IppsPRNG *r`.

Input Arguments

- `r` – the symbolic structure `IppsPRNG`

Output Arguments

- `curbitlength` – current bit length of the pseudo-random bit sequence generated in `IppsPRNG *r`
- `rseq` – the pseudo-random bit sequence

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers

Prime Number Generation

This section describes a set of primitives that are designed to generate a probable prime number of variable length and to validate a probable prime number through an industry recognized probabilistic primality test scheme for cryptographic use. Following the academic terminology, a probable prime number in this section is defined as an integer which passes the Miller-Rubin probabilistic primality test.

The scheme adopted for the probable prime number generation is based on a well-known prime number theorem. Study shows that the number of primes that are no greater than the given large integer x is closely approximated by the expression x/ln(x). Let π(x) denote the number of primes that are no greater than x, then the statement below is true:

Equation 1

$$\lim_{x \to \infty} \frac{\pi(x)}{x/(\ln x)} = 1$$

Further study shows that, let `X` represent the event that k-bit integer n under test is composite, and let Y_t denote the event that Miller-Rabin test with its test count `t` declares `n` to be a prime, the test error probability $p_{k,t} = P(X/Y_t)$ is upper bounded by

Equation 2

$$p_{k,t} \le k^{2/3} 2^t t^{-1/2} 4^{2-\sqrt{t}} \text{ for } t = 2, k \ge 88 \text{ or } 3 \le t \le k/9, k \ge 21$$

It suggests that a practical strategy for generating a random k-bit probable prime is to repeatedly pick k-bit random odd integers until one is found that could be declared as a probable prime by an industry-recognized probabilistic primality test scheme. This set of (probable) prime number generation primitives permits users to specify an appropriate security parameter `t` used in the Miller-Rabin primality test to meet the cryptographic requirements for their applications.

Throughout this section, all the primitives employ a symbolic structure `IppsPrime` to serve as an operational vehicle which carries the bit length of the target probable prime number, the state of the pseudo-random number generation, and the buffers used for Montgomery modular computation involved in the Miller-Rubin primality test, as well as for storing the generated probable prime number.

Furthermore, two new terms are introduced in this section. The bit length of the symbolic structure `IppsPrime` is defined as the bit length of the prime integer `p` carried by the structure. The size of the symbolic structure `IppsPrime` is therefore defined as the maximum bit length of such a prime integer `p` that could be carried by this operational vehicle.

The application code using the primitives described in this section to generate a probable prime number of a specified bit length should perform the following sequence of operations:

1. Get the buffer size required to configure the symbolic structure `ippsPrime` by calling the primitive `ippsPrimeBuffersize()`. For limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer size significantly while using the sliding window method enhances the performance.
2. Allocates a working buffer through an OS memory allocation function and configure the structure `ippsPrime` by calling the primitive `ippsPrimeInit()` with an allocated buffer and the choice made on the modular exponential method at the time of invoking `ippsPrimeBuffersize()`.
3. Call `ippsPrimeGen()` to generate the desired probable prime number of specified bit length using a 320-bit seed and an up to 1024-bit context bit string of users' choice. Check the returned `IppStatus` from this primitive, and if it is `ippStsInsufficientEntropy`, call `ippsPrimeGen()` again with a new set of randomized seed and context bit string.
4. Get the generated probable prime number by calling the primitive `ippsPrimeGet()`.
5. Free the memory allocated to structure `IppsPrime` by calling the OS memory-free service function if needed.

PrimeBufferSize

Prototype

```
IppStatus ippsPrimeBufferSize(IppsExpMethod method, int bitlength, int *size);
```

Description

`ippsPrimeBufferSize()` specifies the buffer size (in number of bytes) required for defining a structured working buffer of symbolic data type `IppsPrime` for both generating and storing a probable prime number of specified bit length (`int bitlength`).

`ippsPrimeBufferSize()` returns the required buffer size `int *size` also based on the exponential method of user's choice used in the scheme for probable prime number generation. Using binary method reduces buffer size significantly, while using sliding windows method enhances the performance.

Input Arguments

- `method` – specifies the exponential method of user's choice
- `bitlength` – the bit length for the desired probable prime number

Output Arguments

- `size` – the size of required buffer (in bytes) for initializing respective `IppsPrime`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `bitlength < or = 0`

PrimeInit

Prototype

```
IppStatus ippsPrimeInit(IppsExpMethod method, int bitlength, IppsPrime *p);
```

Description

`ippsPrimeInit()` initializes the symbolic data type `IppsPrime *p`, using the user supplied buffer pointed by the input `*p`. Based on the modular exponential method of users choice specified by the input `IppsExpMethod method`, `ippsPrimeInit()` partitions the given buffer in such a way that is able to generate, to test and to store the probable prime number up to the specified number of bits (`int bitlength`).

Input Arguments

- `method` – specifies the exponential method of user's choice
- `bitlength` – the target bit length for the desired probable prime number
- `p` – pointer to the user supplied buffer

Output Arguments

- `p` – the pointer to the initialized symbolic structure `IppsPrime`

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `bitlength < or = 0`

PrimeGen

Prototype

```
IppStatus ippsPrimeGen (IppsBigNum *c, Ipp32u *seed, int bitlength, int t,
    IppsPrime *p);
```

Description

`ippsPrimeGen()` uses a 320-bit input `seed` and up to 1024-bit context bit string held by `IppsBigNum *c` to generate a random probable prime number `IppsPrime *p` of the specified bit length (`int bitlength`). The generated probable prime number is further validated by the Miller-Rabin primality test scheme with the user specified security parameter `t`.

`ippsPrimeGen()` returns its `IppStatus` as `ippStsInsufficientEntropy`, if it detects that it needs more entropy seed during its probable prime generation. In this case, the user should update both `IppsBigNum *c` and `Ipp32u *seed`, and call the primitive again.

Input Arguments

- `c` – a `b`-bit content string, with $320 \le b \le 1024$
- `seed` – 320-bit seed value
- `bitlength` – the target bit length for the desired probable prime `p`
- `t` – test count specified for Miller-Rabin probable primality test

Output Arguments

- `p` – the generated probable prime number

Returns

- ippStsNoErr – No Error
- ippStsBadArgErr – If (b < 320) or (t < or = 0)
- ippStsNullPtrErr – Null pointers
- ippStsLengthErr – If bitlength < or = 0.
- ippStsOutOfRangeErr – If bitlength is larger than the size of the structure IppsPrime *p.
- ippStsInsufficientEntropy– Prime generation failure due to the poor choice of seed/context bit stream

PrimeTest

Prototype

```
IppStatus ippsPrimeTest (IppsPrime *p, IppsBigNum *c, Ipp32u *seed, int t,
    Ipp32u *result);
```

Description

`ippsPrimeTest()` uses the Miller-Rabin probabilistic primality test scheme with given security parameter `t` to test if the given integer `ippsPrime *p` is a probable prime. The pseudo-random number used in the Miller-Rabin test is generated by the pseudo-random number generator primitives described in this section, with the input context bit string of up to 512-bit held by `IppsBigNum *c` and 160-bit `Ipp32u *seed`.

`ippsPrimeTest()` sets its output `Ipp32u *result` to be either `IS_PRIME`, if the input integer passes the Miller-Rubin test; or `IS_COMPOSITE` if the input fails the test.

Input Arguments

- `p` – an input big number integer
- `c` – a `b`-bit content string, with $160 \le b \le 512$
- `seed` – 160-bit seed value
- `t` – test count

Output Arguments

- `result` – test result

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – if `(b < 320) or (t < or = 0)`
- `ippStsNullPtrErr` – NULL pointers

PrimeSet

Prototype

```
IppStatus ippsPrimeSet(const Ipp32u *prime, int bitlength, IppsPrime *p);
```

Description

`ippsPrimeSet()` sets a probable prime number (`const Ipp32u *prime`) and its bit length (`int bitlength`) for the symbolic structure `IppsPrime *p`.

Input Arguments

- `prime` – the input prime
- `bitlength` – the bit length of input prime

Output Arguments

- `p` – the symbolic structure `IppsPrime` loaded with the user supplied prime value

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – Bad Argument

PrimeGet

Prototype

```
IppStatus ippsPrimeGet(Ipp32u *prime, int *bitlength, const IppsPrime
   *p);
```

Description

`ippsPrimeGet()` extracts the bitlength (`int *bitlength`) and the probable prime number (`Ipp32u *prime`) from the input `IppsPrime *p`.

Input Arguments

- p– the symbolic structure IppsPrime

Output Arguments

- prime– the data field of the probable prime
- bitlength– the bitlength of the probable prime

Returns

- ippStsNoErr – No Error
- ippStsNullPtrErr – Null pointers

RSA Cryptographic Primitives

This section describes the set of primitives that completes the operations required for RSA cryptographic systems. This set of primitives offers a flexible user interface that enables the RSA cryptographic key size scalability (with up limit to 4k bits). Application code could use them to build a RSA cryptographic system with user supplied randomized seed and stimulus. The primitive ippsRSAKeyGen() generates the RSA system probable primes p and q, the system composite integer n, as well as the key pair - the RSA public key e and its respective private key d. The primitives ippsRSAEncrypt() and ippsRSADecrypt() are the RSA cryptographic functions.

Throughout this section, all the primitives employ a symbolic structure IppsRSA to serve as an operational vehicle which carries the RSA system composite integer, the pair of RSA probable primes, the RSA key pair, as well as a structured working buffer with the components computed in advance for RSA decryption using CRT. Inside this symbolic structure IppsRSA, it also includes an internal flag with the definition defined by the following IppsRSAType, to reflect the RSA system configuration status established for the structure.

```
typedef enum
typedef enum
{
IppsRSAPublic,
IppsRSAPrivate,
IppsRSAInvalid,
IppsRSAKeyN,
IppsRSAKeyP,
IppsRSAKeyQ,
IppsRSAKeyE,
IppsRSAKeyD,
```

```
IppsRSAKeyDP,
IppsRSAKeyDQ,
IppsRSAKeyQInv,
} IppsRSAType;
```

If `flag == IppsRSAPublic`, it represents that `IppsRSA` has been setup properly for RSA encryption, it indicates that the RSA system components for the composite integer `n` and the public key `e` have been setup. If `flag == IppsRSAPrivate`, it means that `IppsRSA` has been properly initialized for RSA decryption, and also that not only all components of RSA cryptographic system have been established, but also the key pair components for the RSA cryptographic operations have been validated. However, if `flag == IppsRSAInvalid`, it means that the RSA key structure is either set by the application to invalidate a pair of compromised key, or the key pair have not been properly setup, mostly it is due to the failure in the RSA key validation test devised by the primitive function `ippsRSAKeyCheck()`.

Other values of `IppsRSAType` permit application code to set each RSA system component with existing data protected in the system using the primitive `ippsRSAKeySet()` individually.

Furthermore, two new terms are introduced in this section. The bit length of the symbolic structure `IppsRSA` is defined as the bit length of the RSA system composite integer `n` carried by the structure. The size of the symbolic structure `IppsRSA` is therefore defined as the maximum bit length of such a composite integer `n` that could be carried by this operational vehicle.

Application code of this set of primitives could direct its desired RSA cryptographic operation operated exclusively in its designated memory segments by calling the primitive `ippsRSAInit()` to organize the supplied memory segments properly for the symbolic structure `IppsRSA`. This feature offers the user great manageability to protect key data during the procedure for key setup, private key generation, encryption and decryption.

To perform a RSA cryptographic operation, users need to generate two secret RSA-secure probable primes `p` and `q`, both should be selected in such a way that factoring `n = p*q` is computational infeasible. The primitive function `ippsRSAKeyGen()` offers a solution to generate such a pair of probable primes `p` and `q` based on the initial state of the public key `e` of the user's choice, that `n = p*q` is of specified `bitlength`, and `gcd(e, p-1)=1`, as well as `gcd(e, q-1)=1`. The difference `p-q` is not a small valued integer, and both `p-1` and `q-1` have large prime factors. This primitive function also computes the private key `d` with respect to the generated public key `e`, as well as several other components for applying the CRT.

The primitive `ippsRSAEncrypt()` computes the modular exponentiation $y=x^e \bmod n$ for encryption, if the internal `flag` of the input key equals `IppsRSAPublic`. The primitive function `ippsRSADecrypt()` computes $y=x^d \bmod n$ for decryption, if the internal `flag` of the input key equals `IppsRSAPrivate`. Both primitives return `ippStsInvalidCryptoKeyErr` to reflect that a bad key has been detected for any invalid internal `flag` value.

It should be noted that the primitive function ippsRSADecrypt() takes computational advantage to enhance its performance. It applies CRT to reduce the modulus size for computing the modular exponentiation and uses the Garner's formula to integrate the results. To balance the performance between encryption and decryption, it is a common practice to reduce the processing time for RSA encryption by selecting a public key e to be small/or with a small number of 1's in its binary representation.

The application code for conducting a typical RSA encryption/decryption must perform the following sequence of operations:

1. RSA decryption:
    a. Get the buffer sizes required to configure the symbolic structure `ippsRSA` by calling the primitive `ippsRSABuffersizes()`. For a limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer sizes significantly while using the sliding window method enhances the performance.
    b. Ensure that various memory segments pointed by `Ipp32u *buffer0`, `*buffer1`, `*buffer2`, and `*key` are appropriately allocated and their sizes are no less than the ones calculated from the primitive `ippsRSABuffersizes()`. Integrate these memory segments for configuring the symbolic structure `ippsRSA` by calling the primitive `ippsRSAInit()` using all the allocated memory segments with the input `flag = IppsRSAPrivate`, as well as the exponential method index.

c. Establish the RSA cryptographic system by means of either key generation primitive `ippsRSAKeyGen()` or key setup primitive `ippsRSAKeySet()`.

(1) RSA key generation:

To renew the RSA session, application code needs to specify the bit lengths for both the RSA system composite integer `n` and the probable prime `p`, and supplies the user's initial choice `*e` for searching a RSA public key to the primitive `ippsRSAKeyGen()` with random `seed` and stimulus (context bit string) `IppsBigNum *c`.

(2) RSA key setup:

To resume the RSA session, application code should perform the following sequence of operations as to properly setup the desired RSA key structure with all the existing RSA system components protected in the system memory:

— Supply the `size` and `value` of the composite integer `Ipps32u *n` to the primitive `ippsRSAKeySet()` with `flag` = `IppsRSAKeyN`.

— Supply the `size` and `value` of the probable prime `Ipps32u *p` to the primitive `ippsRSAKeySet()` with `flag` = `IppsRSAKeyP`.

— Supply the `size` and `value` of the probable prime `Ipps32u *q` to the primitive `ippsRSAKeySet()` with `flag` = `IppsRSAKeyQ`.

— Supply the `size` and `value` of the public key `Ipps32u *e` to the primitive `ippsRSAKeySet()` with `flag` = `IppsRSAKeyE`.

— Supply the `size` and `value` of the private key `Ipps32u *d` to the primitive `ippsRSAKeySet()` with `flag` = `IppsRSAKeyD`.

— Check the return value `IppStatus` to ensure the validity of the attained RSA system components as well as the RSA key pairs.

d. Invoke the primitive `ippsRSADecrypt()` with the newly established RSA key structure `IppsRSA *key` and the ciphertext `IppsBigNum *x`, the result of the operation `IppsBigNum *y` is the respective plaintext.

e. Take proper security measure to secure the established RSA key structure, including extracting the RSA system components by calling the primitive `ippsRSAKeyGet()`, moving the extracted sensitive components into a secure area, and further calling the OS memory-free service function to release buffers if needed.

2. RSA encryption:

a. Get the buffer sizes required to configure the symbolic structure `ippsRSA` by calling the primitive `ippsRSABuffersizes()`. For limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer sizes significantly while using sliding window method enhances the performance.

b. Ensure that various memory segments pointed by `Ipp32u *buffer0`, `*buffer1`, `*buffer2`, and `*key` are appropriately allocated and their sizes are no less than the ones calculated from the primitive `ippsRSABuffersizes()`. Integrate these memory

segments for configuring the symbolic structure `ippsRSA` by calling the primitive `ippsRSAInit()` using all the allocated memory segments with the input `flag = IppsRSAPublic`, as well as the exponential method index.

c. Request and obtain the partner's RSA system composite integer component `n` and the RSA public key component `d`, then setup the RSA cryptographic system with the following sequence of operations:
   — Supply the `size` and `value` of the composite integer `Ipp32u *n` to the primitive `ippsRSAKeySet()` with `flag = IppsRSAKeyN`.
   — Supply the `size` and `value` of the public key `Ipp32u *e` to the primitive `ippsRSAKeySet()` with `flag = IppsRSAKeyE`.

d. Invoke the primitive `ippsRSAEncrypt()` with the newly established RSA key structure `IppsRSA *key` and the value of the plaintext `IppsBigNum *x`, the result of the operation `IppsBigNum *y` is the respective ciphertext.

e. Call the OS memory-free service function to release buffers if needed.

RSABufferSizes

Prototype

```
IppStatus ippsRSABufferSizes (IppsExpMethod method, int k, int kp,IppsRSAType
    flag, int *buffer0_size, int *buffer1_size, int *buffer2_size, int *size);
```

Description

`ippsRSABufferSizes()` specifies various buffer sizes (in number of bytes) required for defining a structured working buffer of symbolic data type `IppsRSA` for the usage of RSA encryption and decryption. In case of RSA encryption (`flag == ippsRSAPublic`), this function calculates various buffer sizes for `IppsRSA` required to hold the `k`-bit RSA system composite integer `n`, the RSA public key `e`, as well as to permit the computation involved for executing RSA encryption.

In case of RSA decryption (`flag == ippsRSAPrivate)`, this function calculates various buffer sizes for `IppsRSA` required to hold the `k`-bit RSA system composite integer `n`, the `kp`-bit probable prime `p`, the `(k-kp)`-bit probable prime `q` and the RSA key pair (the public key `e` with its respective private key `d)`. The structured working buffer `IppsRSA` should also reserve sufficient working buffer to permit the computation involved for executing RSA decryption.

Any other flag value will be treated as a bad input argument and hence result in a return error code ippStsBadArgErr.

All the buffer sizes specified by ippsRSABufferSizes() are based on the exponential method selected by the user in the schemes for RSA key generation and cryptographic operation. Using the binary method reduces buffer size significantly, while using sliding windows method enhances the performance.

Input Arguments

- method – specifies the exponential method of user's choice
- k – the bit length of RSA system, or the bit length of the composite integer n
- kp – the bit length of RSA system prime p
- flag – IppsRSAPublic for RSA encryption and IppsRSAPrivate for RSA decryption

Output Arguments

- buffer0_size – the required buffer size for buffer0
- buffer1_size – the required buffer size for buffer1
- buffer2_size – the required buffer size for buffer2
- size – the required size for buffer pointed by key

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – if any of the following occurs: (1). (kp < k/2) (2). (kp > or = k) (3). bad flag value
- ippStsNullPtrErr – NULL pointers
- ippStsNotSupportedModeErr – if (k < 32) or (k > 4096)

RSAInit

Prototype

```
IppStatus ippsRSAInit (IppsExpMethod method, int k, int kp, IppsRSAType flag,
    Ipp32u *buffer0, Ipp32u *buffer1, Ipp32u *buffer2, IppsRSA *key);
```

Description

ippsRSAInit() initializes the symbolic data type IppsRSA *key, using the user supplied multiple buffers pointed by the pointers *buffer0, *buffer1, *buffer2, and *key. Based on the modular exponential method of users choice specified by the input IppsExpMethod method and the usage of the RSA system specified by the input IppsRSAType flag, ippsRSAInit() integrates the given buffers in such a way that meets the buffer requirements of all the RSA primitives in this section in the process to generate, to validate and to store the respective RSA system components, as well as to perform RSA cryptographic operations.

The symbolic structure initialization is different for the cases of RSA encryption and decryption. In case that (flag == IppsRSAPublic), this function sets up IppsRSA *key for RSA encryption; while in case that (flag == IppsRSAPrivate), this function sets up IppsRSA *key for RSA decryption. Any other flag value will be treated as a bad input argument and hence result in returning IppStatus to reflect that an invalid input has been detected.

All the primitives implemented in this release limits the RSA cryptographic operations with key size scalable up to 4096 bits.

Input Arguments

- method – specifies the exponential method of user's choice
- k – bit length of the composite integer n (32 < or = k < or = 4096)
- kp – bit length of the larger prime
- flag – RSA system usage type, IppsRSAPublic for RSA encryption and IppsRSAPrivate for RSA decryption
- buffer0 – pointer to the first buffer required for the establishment of IppsRSA
- buffer1 – pointer to the second buffer required for the establishment of IppsRSA
- buffer2 – pointer to the third buffer required for the establishment of IppsRSA
- key – pointer to the last buffer for the establishment of IppsRSA

Output Arguments

- key – the pointer to the initialized symbolic structure IppsRSA

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – if any of the following occurs: (1). (kp < k/2) (2). (kp > or = k) (3). bad flag value.
- ippStsNullPtrErr – NULL pointers
- ippStsNotSupportedModeErr – if (k < 32) or (k > 4096)

RSAKeySet

Prototype

```
IppStatus ippsRSAKeySet(const Ipp32u *value, int length, IppsRSAType flag,
    IppsRSA *key);
```

Description

`ippsRSAKeySet()` sets up the `value` to the `flag` designated component of the `IppsRSA *key`. If `(flag == IppsRSAKeyN)`, it sets up the RSA system composite integer `n` with the input value along with its length. If `(flag == IppsRSAKeyP)`, it sets up the RSA system probable prime `p` with the input `value` along with its `length`. If `(flag == IppsRSAKeyQ)`, it sets up the RSA system probable prime `q` with the input value along with its length. If `(flag == IppsRSAKeyE)`, this function sets up the RSA system public key `e` with the input value along with its `length`. If `(flag == IppsRSAKeyD)`, `ippsRSAKeySet()` sets up the RSA system private component `d` with the input value along with its `length`.

Once detected that all the RSA private key components have been established, this function will further validate the RSA key structure by calling `ippsRSAKeyCheck()`. For those valid RSA private key input, this function then further computes all other CRT related RSA components.

And if `(flag == IppsRSAInvalid)`, `ippsRSAKeySet()` invalidates the RSA key. Any other flag value will be treated as a bad input argument and hence result in returning the error code `ippStsBadArgErr`.

`ippsRSAKeySet()` returns `ippStsBadArgErr` in response to any attempt to set the RSA system primes or the RSA private key components, if the input RSA key structure `const IppsRSA *key` has been initialized for the usage of RSA encryption. It also returns `ippStsInvalidCryptoKeyErr` if it detects that the components of the RSA key fail to pass a RSA key checking scheme.

Input Arguments

- `value` – array of data (`Ipp32u` data type) for the designated component of `key`
- `length` – length (in number of 32-bit words) of the input array pointed by `*value`
- `flag` – specify the target component of `IppsRSA *key`
- `key` – symbolic structure for RSA cryptographic system

Output Arguments

- `key` – the result of RSA key structure

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – if either `(flag == IppsRSAPublic)` or `(flag == IppsRSAPrivate)`
- `ippStsNullPtrErr` – NULL pointers
- `ippStsLengthErr` – if `length < or = 0`
- `ippStsOutOfRangeErr` – if `length` is larger than the respective component size of the input structure `IppsRSA *key`
- `ippStsInvalidCryptoKeyErr` – the resulting key is found to be compromised

RSAKeyGet

Prototype

```
IppStatus ippsRSAKeyGet(Ipp32u *value, int *length, IppsRSAType flag,
   const IppsRSA *key);
```

Description

`ippsRSAKeyGet()` extracts the `flag` designated component (`Ipp32u *value`) from `IppsRSA *key`. It extracts the RSA system composite integer `n` with its `length`, if `(flag == IppsRSAKeyN)`. It extracts the RSA system probable prime `p` with its `length`, if `(flag == IppsRSAKeyP)`. It extracts the RSA system probable prime `q` with its `length`, if `(flag == IppsRSAKeyQ)`. It extracts the RSA public key `e` with its `length`, if `(flag == IppsRSAKeyE)`, and it extracts the RSA private key `d` with its `length`, if `(flag == IppsRSAKeyD)`.

`ippsRSAKeyGet()` returns `ippStsBadArgErr` in response to the attempt to get the RSA system primes or the RSA private key components, if the input RSA key structure `const IppsRSA *key` has been initialized for the usage of RSA encryption, or if the input `flag` value is either `IppsRSAPublic` or `IppsRSAPrivate`. It also returns `ippStsInvalidCryptoKeyErr`, if the RSA key is found to be compromised. In both cases, application should disregard both `*value` and its `length`.

Input Arguments

- `flag` – specify the target component of `IppsRSA *key`
- `key` – symbolic structure for RSA cryptographic system

Output Arguments

- `value` – data array (`Ipp32u` data type) of the `flag` designated component of `key`
- `length` – data length (in number of 32-bit words)

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – Either bad `flag` value or attempt to fetch RSA system primes or private key components, if the input structure `IppsRSA *key` has been configured to be for the usage of RSA encryption.
- `ippStsNullPtrErr` – NULL pointers
- `ippStsInvalidCryptoKeyErr` – the key is found to be compromised

RSAKeyCheck

Prototype

```
IppStatus ippsRSAKeyCheck(IppsBigNum *c, Ipp32u *seed, int t, IppsRSA *key,
    Ipp32u *result);
```

Description

`ippsRSAKeyCheck()` validates all of the components of the input `IppsRSA *key`, if the RSA key structure has been initialized for the usage of RSA decryption. It starts with using Intel® IPP primitive function `ippsPrimeTest()` with user inputs `Ipp32u *seed` and `IppsBigNum *c` to validate if both RSA system probable prime components `p` and `q` meet the requirements for the cryptographic use. It further validates two additional RSA system requirements:

1. The RSA system composite integer component n is the product of two RSA system probable prime components `p*q`, and
2. The RSA system key pair, the public key `e` and the private key `d`, have the multiplicative inversion relationship with respect to the modulus defined by the Euler phi function of the RSA system composite integer `n`, or simply described as `e*d = 1 mod ϕ(n)`, where `ϕ(n) = (p - 1)*(q -1)`.

`ippsRSAKeyCheck()` sets the output result to be `IS_VALID_KEY` if the input `IppsRSA *key` passes all the tests listed above. Or it sets the output result to be `IS_INVALID_KEY` and further disables the input `RSA` key structure `IppsRSA *key`, once it fails any test described above. Or it sets the output result to be `IS_INCOMPLETED_KEY`, once it detected that there is a missing component of the RSA key structure `IppsRSA *key`.

Input Arguments

- `c` – a `b`-bit content string, where $160 \le b \le 512$
- `seed` – 160-bit seed value
- `t` – test count
- `key` – the RSA key structure equipped with system parameters and the key pair

Output Arguments

- `result` – the check result

Returns

- `ippStsNoErr` – no error
- `ippStsBadArgErr` – if `(b < 160)` or `(t < or = 0)`
- `ippStsNullPtrErr` – NULL pointers

RSAKeyGen

Prototype

```
IppStatus ippsRSAKeyGen (IppsBigNum *e, IppsBigNum *c, Ipp32u *seed, int t,
    int k, int kp, IppsRSA *key);
```

Description

ippsRSAKeyGen() generates a RSA key structure for setting up the desired RSA cryptographic system based on the input IppsBigNum *e specifying the initial value for searching of RSA public key, the input parameters k and kp specifying the bit lengths for the RSA composite integer n and the larger RSA probable prime p respectively.

It employs a FIPS-approved probable prime generation scheme with the user supplied real time system parameters as both seed and stimulus (context bit string) IppsBigNum *c, they are used for generating the pseudo-random bit sequences during the prime generation as well as its primality test. The RSA system primes are generated in such a way that both probable primes p and q are randomly generated and their difference (p - q) is not a small integer.

ippsRSAKeyGen() treats the input IppsBigNum *e as an initial value in an entropy-preserving iteration scheme in searching of the RSA public key e which is co-prime to (p - 1)*(q -1).

It then computes the RSA composite integer n = (p*q), the RSA private key component d, and it further computes all other CRT related RSA components.

ippsRSAKeyGen() returns ippStsBadArgErr, if IppsRSA *key was initialized for RSA encryption use.

Input Arguments

- e – the initial value to the scheme for searching a RSA public key component
- c – a b-bit content string for pseudo random bit sequence generation, with $320 \le b \le 1024$
- seed – 320-bit seed value for pseudo random bit sequence generation
- t – test count for Miller-Rabin primality test
- k – bit length of the composite integer n
- kp – bit length for the probable prime p, where kp is no less than k/2

Output Arguments

- key – the result of RSA key structure
- e – a copy of the newly generated RSA public key

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – if any of the following occurs: (1) IppsRSA *key has been initialized for the usage of RSA encryption. (2) security parameter (t < or = 0) (3) random content bit string bitlength(b < 320) (4) RSA system prime bit length ((kp < k/2) or (kp > or = k))
- ippStsNullPtrErr – NULL pointers

- ippStsOutOfRangeErr – if either k is larger than the size of the input structure IppsRSA *key, or kp is larger than the RSA system prime component size of this given structure IppsRSA *key
- ippStsNotSupportedModeErr – if ((k < 32) or (k > 4096)).
- ippStsInsufficientEntropy – insufficient entropy in the random seed and stimulus bit string caused the RSA key generation to fail

RSAEncrypt

Prototype

```
IppStatus ippsRSAEncrypt(IppsBigNum *x, IppsBigNum *y, IppsRSA *key);
```

Description

ippsRSAEncrypt() performs the RSA encryption operation under the condition that the input RSA key structure has been setup for the usage of the RSA encryption. It encrypts by computing the modular exponentiation of the plaintext valued as IppsBigNum *x with the exponent specified by the RSA public key e with respect to the module valued as the RSA system composite integer n, or simply $(y = x^e \bmod n)$.

The application should disregard the output IppsBigNum *y, if the primitive ippsRSAEncrypt() returns its status as ippStsInvalidCryptoKeyErr.

Input Arguments

- x – plaintext value satisfying (0<x<n)
- key – RSA key structure

Output Arguments

- y – the ciphertext result

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – if IppsRSA *key has been initialized for the usage of RSA decryption
- ippStsNullPtrErr -Null pointers
- ippStsScaleRangeErr – if (x > n)

- ippStsOutOfRangeErr – if the size of IppsBigNum *y is less than the bit length of IppsRSA *key
- ippStsInvalidCryptoKeyErr – the compromised key causes the requested RSA encryption suspended

RSADecrypt

Prototype

```
IppStatus ippsRSADecrypt(IppsBigNum *x, IppsBigNum *y, IppsRSA *key);
```

Description

ippsRSADecrypt() performs the RSA decryption operation under the condition that the input RSA key structure has been setup for the usage of the RSA decryption. It decrypts by using Chinese Remainder Theorem to compute the modular exponentiation of the ciphertext valued as IppsBigNum *x with the exponent specified by the RSA private key d with respect to the module valued as the RSA system composite integer n, or simply $(y = x^d \bmod n)$.

The application should disregard the output IppsBigNum *y, if the primitive ippsRSAEncrypt() returns its status as ippStsInvalidCryptoKeyErr.

Input Arguments

- x – ciphertext value for RSA decryption, where (0<x<n)
- key – RSA key structure

Output Arguments

- y – the plaintext result

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – if IppsRSA *key has been initialized for the usage of RSA encryption
- ippStsNullPtrErr – NULL pointers
- ippStsScaleRangeErr – if (x > n)
- ippStsOutOfRangeErr – if the size of IppsBigNum *y is less than the bit length of IppsRSA *key

- `ippStsInvalidCryptoKeyErr` – the compromised key causes the requested RSA decryption suspended

DSA Digital Signature Primitives

This section describes the set of eight primitives that completes the operations required for DSA digital signature algorithm. The primitive `ippsDSAKeyGen(), ippsDSAKeySet()` and `ippsDSAKeyCheck()` are three primitives managing the DSA key structure, including creating, setting up or checking all the key components required for DSA digital signature generation and verification. The primitives `ippsDSASign()` and `ippsDSAVerify()` are designed for signing and verification of the DSA digital signature for the underlined message content represented by its respective SHA-1 message digest value. All the primitives are implemented in the way to fully comply with the FIPS 186-2.

Throughout this section, all the primitives employ a symbolic structure `IppsDSA` to serve as an operational vehicle which carries the pair of DSA system primes `p` and `q`, the DSA system component `g`, the DSA signing key `x` and the DSA verification key `y`. This symbolic structure also captures several auxiliary components, including both the seed and loop counts that were used in the prime generation, the component `J` defined as `(p-1)/q`, as well as a working buffer reserved for various DSA cryptographic operations. Inside this symbolic structure `IppsDSA`, it also includes an internal flag with the definition defined by the following `IppsDSAType`, to reflect the DSA system configuration status established for the structure.

```
typedef enum
{
IppsDSASign,
IppsDSAVerify,
IppsDSAInvalid,
IppsDSAKeyCount,
IppsDSAKeySeed,
IppsDSAKeyP,
IppsDSAKeyQ,
IppsDSAKeyG,
IppsDSAKeyX,
IppsDSAKeyY,
IppsDSAKeyJ,
} IppsDSAType;
```

If `flag == IppsDSASign`, it represents that `IppsDSA` has been setup properly for DSA digital signature generation. If `flag == IppsDSAVerify`, it means that `IppsDSA` has been properly initialized for DSA digital signature verification. However, if `flag == IppsDSAInvalid`, it means

that the DSA key structure is either set by the application to invalidate a pair of compromised key, or the key pair have not been properly setup, mostly it is due to the failure in the DSA key validation test devised by the primitive function `ippsDSAKeyCheck()`.

Other values of `IppsRSAType` permit application code to set each DSA system component with existing data protected in the system using the primitive `ippsDSAKeySet()` individually.

Furthermore, two new terms are introduced as well in this section. The bitlength of the symbolic structure `IppsDSA` is defined as the bitlength of the DSA prime modulus `p` carried by the structure; and the size of the symbolic structure `IppsDSA` is therefore defined as the maximum bitlength of such a prime modulus `p` that could be carried by this operational vehicle.

Application code of this set of primitives could direct its desired DSA cryptographic operation operated exclusively in its designated memory segments by calling the primitive `ippsDSAInit()` to organize the supplied memory segments properly for the symbolic structure `IppsDSA`. This feature offers its user great manageability to protect its key data during the procedure for key setup, key generation, signing and verifying the DSA digital signature.

The application code for conducting a typical DSA digital signature signing/verifying operation must perform the following sequence of operations:

1. DSA digital signature signing:
   a. Get the buffer sizes required to configure the symbolic structure `ippsDSA` by calling the primitive `ippsDSABuffersizes()`. For limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer sizes significantly while using sliding window method enhances the performance.
   b. Ensure that various memory segments pointed by `Ipp32u *buffer0, *buffer1, *buffer2,` and `*key` are appropriately allocated and their sizes are no less than the ones calculated from the primitive `ippsDSABuffersizes()`. Integrate these memory segments for configuring the symbolic structure `ippsDSA` by calling the primitive `ippsDSAInit()` using all the allocated memory segments with the input `flag = IppsDSASign`, as well as the exponential method index.
   c. Establish the DSA cryptographic system by means of either key generation primitive `ippsDSAKeyGen()` or key setup primitive `ippsDSAKeySet()`.
      - (1) DSA key generation: To renew the DSA session, application code needs to specify the bitlengths for the DSA system prime modulus `p` and supplies the user's entropy inputs `*seed` and stimulus bit-stream `*c` for searching a set of FIPS 186-2 qualified DSA key components with the primitive `ippsDSAKeyGen()`.
      - (2) DSA key setup: To resume the DSA session, application code should perform the following sequence of operations as to properly setup the desired DSA key structure with all the existing DSA system components protected in the system memory:
      - - Supply the `size` and `value` of the DSA loop count `Ipps32u count` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyCount`.

- - Supply the `size` and `value` of the DSA seed component `Ipps32u *seed` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeySeed`.
- - Supply the `size` and `value` of the probable prime `Ipps32u *p` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyP`.
- - Supply the `size` and `value` of the probable prime `Ipps32u *q` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyQ`.
- - Supply the `size` and `value` of the DSA system component `Ipps32u *g` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyG`.
- - Supply the `size` and `value` of the DSA digital signature signing key `Ipps32u *x` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyX`.
- - Supply the `size` and `value` of the DSA digital signature verification key `Ipps32u *y` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyY`.
- - Supply the `size` and `value` of the DSA system J-component `Ipps32u *J` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyJ`.
- - Check the return value `IppStatus` to ensure the validity of the attained DSA system components as well as the DSA key pairs.

d. Invoke the primitive `ippsDSASign()` with the newly established DSA key structure `IppsDSA *pCtx` and the SHA-1 message digest value of the underlined message content, the result of the operation `(*r, *s)` is the respective DSA digital signature.

e. Take proper security measure to secure the established DSA key structure, including extracting the DSA system components by calling the primitive `ippsDSAKeyGet()`, moving the extracted sensitive components into a secure area, and further calling the OS memory-free service function to release buffers if needed.

2. DSA digital signature verification:

a. Get the buffer sizes required to configure the symbolic structure `ippsDSA` by calling the primitive `ippsDSABuffersizes()`. For limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer sizes significantly while using sliding window method enhances the performance.

b. Ensure that various memory segments pointed by `Ipp32u *buffer0, *buffer1, *buffer2,` and `*key` are appropriately allocated and their sizes are no less than the ones calculated from the primitive `ippsDSABuffersizes()`. Integrate these memory segments for configuring the symbolic structure `ippsDSA` by calling the primitive `ippsDSAInit()` using all the allocated memory segments with the input `flag = IppsDSAVerify`, as well as the exponential method index.

c. Request and obtain the partner's DSA system prime pair `p` and `q`, the DSA system component `g`, and the DSA digital signature verification key `y`, then setup the DSA cryptographic system with the following sequence of operations:

- - Supply the `size` and `value` of the probable prime `Ipps32u *p` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyP`.
- - Supply the `size` and `value` of the probable prime `Ipps32u *q` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyQ`.
- - Supply the `size` and `value` of the DSA system component `Ipps32u *g` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyG`.
- - Supply the `size` and `value` of the DSA digital signature verification key `Ipps32u *y` to the primitive `ippsDSAKeySet()` with `flag = IppsDSAKeyY`.

d. Invoke the primitive `ippsDSAVerify()` with the newly established DSA key structure `IppsDSA *pCtx`, the DSA digital signature (`*r,*s`), as well as the SHA-1 message digest value of the underlined message content `Ipp8u *mdsha1`, the operation validates the presented DSA digital signature.

e. Call the OS memory-free service function to release buffers if needed.

DSABufferSizes

Prototype

```
IppStatus ippsDSABufferSizes(IppsExpMethod method, int L, IppsDSAType
   flag, int *buffer0_size, int *buffer1_size, int *buffer2_size, int
   *size);
```

Description

`ippsDSABufferSizes()` specifies various buffer sizes (in number of bytes) required for defining a structured working buffer of symbolic data type `IppsDSA` for the usage of DSA digital signature signing and verification. In case of DSA digital signature signing (`flag == ippsDSASign`), this function calculates various buffer sizes for`IppsDSA` required to hold the `L`-bit DSA prime modulus `p`, the `160`-bit DSA prime modulus `q`, the DSA system component `g`, the DSA signing key `x`, as well as other DSA system components and working buffer contributing to the DSA digital signature signing.

In case of DSA digital signature verification (`flag == ippsDSAVerify`), this function calculates various buffer sizes for `IppsRSA` required to hold the `L`-bit DSA prime modulus `p`, the `160`-bit DSA prime modulus `q`, the DSA system component `g`, the DSA verification key `y`. The structured working buffer `IppsDSA` should also reserve sufficient working buffer to permit the computation involved for verifying the DSA digital signature.

Any other `flag` value will be treated as a bad input argument and hence result in a return error code ippStsBadArgErr.

It should be also noted that all the buffer sizes specified by `ippsDSABufferSizes()` are also based on the exponential method of user's choice used in the schemes for DSA key generation and operations for digital signature signing and verification. Using binary method reduces buffer size significantly, while using sliding windows method enhances the performance.

Input Arguments

- `method` – specifies the exponential method of user's choice
- `L`– the bitlength of the DSA prime modulus `p`
- `flag- IppsDSASign` for DSA digital signature signing and `IppsDSAVerify` for DSA digital signature verification

Output Arguments

- `buffer0_size`– the required buffer size for `buffer0`.
- `buffer1_size`– the required buffer size for `buffer1`.
- `buffer2_size`– the required buffer size for `buffer2`.
- `size`– the required size for buffer pointed by `key`.

Returns

- `ippStsNoErr` – No Error
- `ippStsBadArgErr` – If any of the following occurs: (1). $L \neq 512 + 64 \times m; (0 \leq m \leq 8)$ (2). bad `flag` value.
- `ippStsNullPtrErr` – Null pointers

DSAInit

Prototype

```
IppStatus ippsDSAInit(IppsExpMethod method, int L, IppsDSAType flag,
   Ipp32u *buffer0, Ipp32u *buffer1, Ipp32u *buffer2, IppsDSA *pCtx);
```

Description

`ippsDSAInit()` initializes the symbolic data type `IppsDSA *pCtx,` using the user supplied multiple buffers pointed by the pointers `*buffer0`, `*buffer1`, `*buffer2`, and `*pCtx`. Based on the modular exponential method of users choice specified by the input `IppsExpMethod method` and the usage of the DSA system specified by the input `IppsDSAType flag, ippsDSAInit()` integrates the given buffers in such a way that meets the buffer requirements of all the DSA primitives in this section in the process to generate, to validate and to store the respective DSA system components, as well as to perform DSA digital signature operations.

It should be noted that the symbolic structure initialization is different for the cases of DSA digital signature signing and verification. In case that `(flag == IppsDSASign)`, this function sets up `IppsRSA *pCtx` for DSA digital signature signing; while in case that `(flag == IppsDSAVerify)`, this function sets up `IppsRSA *pCtx` for DSA digital signature verification. Any other `flag` value will be treated as a bad input argument and hence result in returning `IppStatus` to reflect that an invalid input has been detected.

Input Arguments

- `method` – specifies the exponential method of user's choice
- `L`– the bitlength of the DSA prime modulus `p`
- `flag- IppsDSASign` for DSA digital signature signing and `IppsDSAVerify` for DSA digital signature verification
- `buffer0` – pointer to the first buffer required for the establishment of `IppsDSA`
- `buffer1` – pointer to the second buffer required for the establishment of `IppsDSA`
- `buffer2` – pointer to the third buffer required for the establishment of `IppsDSA`
- `pCtx`– pointer to the last buffer for the establishment of `IppsRSA`

Output Arguments

- `pCtx`– the pointer to the initialized symbolic structure `IppsRSA`.

Returns

- ippStsNoErr – No Error
- ippStsBadArgErr – If any of the following occurs: (1). $L \neq 512 + 64 \times m; (0 \leq m \leq 8)$ (2). bad flag value.
- ippStsNullPtrErr – Null pointers

DSAKeySet

Prototype

```
IppStatus ippsDSAKeySet(const Ipp32u *value, int length, IppsDSAType
   flag, IppsDSA *pCtx);
```

Description

`ippsDSAKeySet()` sets up the `value` to the `flag` designated component of the `IppsDSA *pCtx`. If `(flag == IppsDSAKeyCount)`, it sets up counter for generating DSA prime modulus `p`, with the input `value` along with its `length`. If `(flag == IppsDSAKeySeed)`, it sets up the seed value for generating DSA prime modulus `p` with the input `value` along with its `length`. If `(flag == IppsDSAKeyP)`, it sets up the DSA prime modulus `p` with the input `value` along with its `length`.If `(flag == IppsDSAKeyQ)`, it sets up the DSA prime `q` with the input `value` along with its `length`. If `(flag == IppsRSAKeyG)`, this function sets up the DSA system component `g`with the input `value` along with its `length`.If `(flag == IppsDSAKeyX)`, `ippsDSAKeySet()` sets up the DSA digital signature signing key `x` with the input `value` along with its `length`.If `(flag == IppsDSAKeyY)`, `ippsDSAKeySet()` sets up the DSA digital signature verification key `y` with the input `value` along with its `length`.If `(flag == IppsDSAKeyJ)`, `ippsDSAKeySet()` sets up the DSA J-component `J` with the input `value` along with its `length`.

Once detected that all the DSA digital signature signing key components have been established, this function will further validates the DSA key structure by calling `ippsDSAKeyCheck()`. For those valid DSA digital signature signing key input, this function then further computes `K`'s multiplicative inverse with respect to prime `q`.

And if `(flag == IppsDSAInvalid)`, `ippsDSAKeySet()` invalidates the DSA key. Any other `flag` value will be treated as a bad input argument and hence result in returning a error code `ippStsBadArgErr`.

It should be noted that `ippsDSAKeySet()` returns `ippStsBadArgErr` in response to any attempt to set the DSA digital signature signing key components `K` and `x`, if the input DSA key structure `const IppsDSA *pCtx` has been initialized for the usage of DSA digital signature verification. And it also returns `ippStsInvalidCryptoKeyErr` if it detects that the components of the DSA key fails to pass a DSA key checking scheme.

Input Arguments

- `value` – array of data (`Ipp32u` data type) for the designated component of `pCtx`.
- `length`– length (in number of 32-bit words) of the input array pointed by `*value`.

- `flag` – specify the target component of `IppsDSA *pCtx`
- `pCtx` – symbolic structure for supporting DSA digital signature operations

Output Arguments

- `pCtx` – the result of DSA key structure

Returns

- `ippStsNoErr` – No Error
- `ippStsBadArgErr` – If either `(flag == IppsDSASign)` or `(flag == IppsDSAVerify)`
- `ippStsNullPtrErr` – Null pointers
- `ippStsLengthErr` – If `length < or = 0.`
- `ippStsOutOfRangeErr` – If `length` is larger than the respective component size of the input structure `IppsDSA *pCtx.`
- `ippStsInvalidCryptoKeyErr` – the resulting key is found to be compromised

DSAKeyGet

Prototype

```
IppStatus ippsDSAKeyGet(Ipp32u *value, int *length, IppsDSAType flag,
   const IppsDSA *pCtx);
```

Description

`ippsDSAKeyGet()` extracts the `flag` designated component (`Ipp32u *value`) from input DSA key structure `IppsDSA *pCtx.` It extracts the count number tried in the process of generating DSA prime modulus `p` with its `length,` if `(flag == IppsDSAKeyCount)`. It extracts the seed that generates the DSA prime modulus `p` with its `length,` if `(flag == IppsDSAKeySeed)`. It extracts the DSA prime modulus `p` with its `length,` if `(flag == IppsDSAKeyP)`. It extracts the DSA system probable prime `q` with its `length,` if `(flag == IppsDSAKeyQ)`. It extracts the DSA system component `g` with its `length,` if `(flag == IppsDSAKeyG)`. It extracts the DSA digital signature signing key `x` with its `length,` if `(flag == IppsDSAKeyX).` It extracts the DSA digital signature verification key `y` with its `length,` if `(flag == IppsDSAKeyY)`.It also extracts the J-component of DSA system with its `length,` if `(flag == IppsDSAKeyJ)`.

It should be noted that `ippsRSAKeyGet()` returns ippStsBadArgErr in response to the attempt to get the DSA digital signature signing key component `x`, if the input DSA key structure `const IppsDSA *pCtx` has been initialized for the usage of DSA digital signature verification, or if the input `flag` value is either `IppsDSASign` or `IppsDSAVerify`. It also returns ippStsInvalidCryptoKeyErr, if the DSA key is found to be compromised. In both cases, application should disregard both `*value` and its `length`.

Input Arguments

- `flag` – specify the target component of `IppsDSA *pCtx`
- `pCtx` – symbolic structure for DSA cryptographic system

Output Arguments

- `value` – data array (`Ipp32u` data type) of the `flag` designated component of `pCtx`.
- `length`– data length (in number of 32-bit words).

Returns

- `ippStsNoErr` – No Error
- ippStsBadArgErr – Either bad `flag` value or attempt to fetch DSA digital signature signing key component `x`, if the input structure `IppsDSA *pCtx` has been configured to be for the usage of DSA digital signature verification.
- `ippStsNullPtrErr` – Null pointers
- `ippStsInvalidCryptoKeyErr` – the key is found to be compromised

DSAKeyCheck

Prototype

```
IppStatus ippsDSAKeyCheck(IppsBigNum *c, Ipp32u *seed, int t, IppsDSA
   *pCtx, Ipp32u *result);
```

Description

`ippsDSAKeyCheck()` validates all the components of the input `IppsDSA *pCtx`, if the DSA key structure has been initialized for the usage of DSA digital signature signing. It starts with using Intel® IPP primitive function `ippsPrimeTest()` with user inputs `Ipp32u *seed` and `IppsBigNum *c` to execute t-round (t > 50) Miller-Rabin primarity test to validate if both DSA system prime

components p and q comply to the FIPS-186-2 specified DSA requirements. It further validates two additional DSA system requirements: (1) the pair of the DSA system primes p and q have the property as q|(p-1), and (2) the DSA system key pair, the signing key x and the verification key y, has the relationship as

$$y = g^x \bmod p$$

ippsRSAKeyCheck() sets the output result to be IS_VALID_KEY if the input IppsDSA *pCtx passes all the tests listed above. Or it sets the output result to be IS_INVALID_KEY and further disables the input DSA key structure IppsDSA *pCtx, once it fails any test described above. Or it sets the output result to be IS_INCOMPLETED_KEY, once it detected that there is a missing component of the DSA key structure IppsDSA *pCtx.

Input Arguments

- c – a b-bit content string, where $160 \le b \le 512$
- seed – 160-bit seed value
- t – test count
- pCtx – the DSA key structure equipped with system parameters and the key pair

Output Arguments

- result– the check result

Returns

- ippStsNoErr – No Error
- ippStsBadArgErr – If (b < 160) or (t < 50)
- ippStsNullPtrErr – Null pointers

DSAKeyGen

Prototype

```
IppStatus ippsDSAKeyGen(IppsBigNum *c, Ipp32u *seed, int t, int L,
   IppsDSA *pCtx);
```

Description

ippsDSAKeyGen() generates a DSA key structure for setting up the desired DSA cryptographic system based on the input IppsBigNum *c and Ipp32u *seed.

It employs a FIPS 186-2 specified procedure to generate both an 160-bit randomized prime q and an L-bit prime p based on the input *seed. Both generated primes q and p are further validated through a 50-round Miller-Rabin primarity test with the user supplied real time system parameters as both seed and stimulus (context bit string) IppsBigNum *c. The internal loop count for generating the DSA prime p is captured in the DSA key structure IppsDSA *pCtx. If the loop count runs over 4096 for generating prime p, or if the loop count runs over 100 for generating prime q, ippsDSAKeyGen() returns ippStsInSufficientEntropy.

ippsDSAKeyGen() also fully complies FIPS 186-2 for the generation of DSA key components g, signing key x, and verification key y. A loop process is also adopted in searching of the DSA key pair x and y until they both fully meet the FIPS 186-2 specified DSA requirement. A loop count over 100 for generating DSA key pair will terminate its execution with the return status code ippStsInSufficientEntropy.

It further computes the DSA system J-component and keep it in the DSA key structure IppsDSA *pCtx for future verification.

It should be noted that ippsDSAKeyGen() returns ippStsBadArgErr, if IppsDSA *pCtx was initialized for DSA digital signature verification.

Input Arguments

- c– a b-bit content string for pseudo random bit sequence generation, with
- seed – 320-bit seed value for pseudo random bit sequence generation
- t – test count for Miller-Rabin probableprimality test
- L – bitlength of the DSA prime modulus p
- pCtx – DSA key structure

Output Arguments

- pCtx – the DSA key structure with generated DSA key components.

Returns

- ippStsNoErr – No Error
- ippStsBadArgErr – If any of the following occurs: (1) IppsDSA *pCtx has been initialized for the usage of DSA digital signature verification. (2) random content bit string bitlength(b < 320)
- ippStsNullPtrErr – NULL pointers
- ippStsOutOfRangeErr – If L is larger than the size of the input structure IppsDSA *pCtx.

- ippStsInsufficientEntropy - Insufficient entropy in the random seed and stimulus bit string caused the DSA key generation to fail.

DSASign

Prototype

```
IppStatus ippsDSASign(IppsBigNum *c, Ipp32u *seed, const Ipp8u *mdsha1,
   Ipp8u *r,                        Ipp8u *s, IppsDSA *pCtx);
```

Description

`ippsDSASign()` performs the DSA digital signature signing operation under the condition that the input DSA key structure `IppsDSA *pCtx` has been setup with `IppsDSASign`. It complies FIPS 186-2 to generate a DSA digital signature of the input message as represented by `Ipp8u *mdsha1` using the input signing key embedded in the DSA key structure `IppsDSA *pCtx`.

To generate a DSA digital signature, `ippsDSASign()` conducts a loop process in search of the signing random sequence `K` based on the entropy input `*seed` and the stimulus context bit-stream `*c`, till it generates a DSA digital signature pair (`Ipp8u *r, Ipp8u *s`). If such a loop count runs over 100, `ippsDSASign()` suspands its execution and returns the status code as `IppStsInSufficientEntropy`.

Input Arguments

- `c` – a `b`-bit content string for pseudo random bit sequence generation, with
- `seed` – 160-bit seed value for pseudo random bit sequence generation
- `mdsha1` – 160-bit SHA-1 message digest value of the underline message content
- `pCtx` – DSA key structure equipped with DSA signing key

Output Arguments

- `r, s` – the resulting DSA digital signature.

Returns

- `ippStsNoErr` – No Error.
- `ippStsBadArgErr` – If `b < 160`.
- `ippStsNullPtrErr` – NULL pointers

- ippStsInvalidCryptoKeyErr – If the input DSA key *pCtx has not setup with IppsDSASign

DSAVerify

Prototype

```
IppStatus ippsDSAVerify(const Ipp8u *mdsha1, const Ipp8u *r, const Ipp8u
   *s, IppsDSA *pCtx, Ipp32u *result);
```

Description

ippsDSAVerify() verifies the input DSA digital signature (Ipp8u *r, Ipp8u *s) with user supplied DSA verification key IppsDSA *pCtx and the SHA-1 message digest value Ipp8u *mdsha1. It sets the *result to be IS_VALID_DS, if it validates the input DSA digital signature; else it sets the *result to be IS_INVALID_DS, in case it fails the DSA digital signature verification.

Input Arguments

- mdsha1 – 160-bit SHA-1 message digest value of the underline message content.
- r, s – the DSA digital signature
- pCtx – DSA key structure

Output Arguments

- result– the verification result

Returns

- ippStsNoErr – No Error.
- ippStsBadArgErr – If either *r, or *s equals to zero.
- ippStsNullPtrErr – NULL pointers
- ippStsInvalidCryptoKeyErr – If the input DSA key IppsDSA *pCtx is found to be compromised.

Audio Toolkit API 20

This chapter describes the Audio Toolkit Application Programming Interface (API) for the Intel® Integrated Performance Primitives (Intel® IPP).

This chapter contains the following sections:

"Noise Reduction" describes the architecture, data structures, and primitives for the Intel® IPP Noise Reduction (NR) implementation.

"Acoustic Echo Canceller" describes architecture, data structures, and primitives for the Intel® IPP Acoustic Echo Canceller (AEC).

"Voice Activity Detector" describes the architecture, data structures, and primitives for the Intel® IPP Voice Activity Detector (VAD).

"Speech Recognition" describes the architecture, data structures, and primitives for the Intel® IPP speech recognition feature extractor (FE) and feature encoder (source/channel coder).

Noise Reduction

This section describes the Intel® Integrated Performance Primitives (Intel® IPP) that can be used to construct an implementation of Ephraim-Malah Noise Suppressor (EMNS), as originally described by Ephraim and Malah. See [Eph84] for more information. 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.

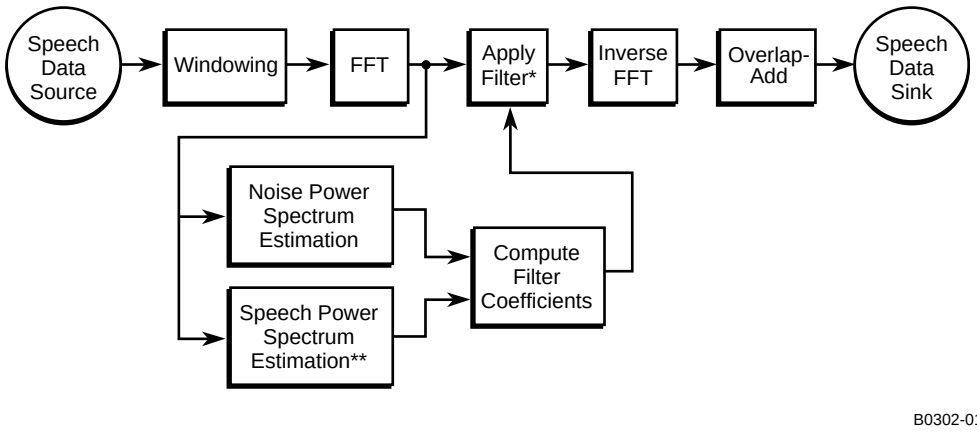
Intel® IPP NR primitives support the following features:

- filter update
- noise floor estimation

Noise Suppressor Architecture

The major building blocks of the Ephraim-Malah Noise Suppressor are shown in Figure 20-1, where we show the steps to apply noise reduction to a speech signal stream.

Figure 20-1 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

Noise Reduction

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 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

$$w(m) = \begin{cases} \dfrac{m+0.5}{N/2} & \text{for } m = 0,...,N/2-1 \\ 1-w(m-N/2) & \text{for } m = N/2,...,N-1 \end{cases}$$

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, o denotes point-wise multiplication and $\sqrt{w}$ denotes a vector containing the square root of the entries of w. F is the Fourier transform matrix with entries, f(m,n)=exp(-j2πmn/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:

$$\beta_n = \beta_{\min} + \rho_{n-1}^y\left(\beta_{\max} - \beta_{\min}\right)$$

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 coarse 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:

$$\alpha_n = \alpha_{\min} + (1 - \rho_{n-1}^{y})(\alpha_{\max} - \alpha_{\min})$$

where $\alpha_{\min} = 0.91$, $\alpha_{\max} = 0.95$, and ρ_{n-1}^{y} is the likelihood of speech presence in bin *k*. Note that 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 description, a dependency is assumed.

5. **Update Wiener filter weights**. 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 [Cappe94]. We recommend the following lower limit for a priori SNR:

$$\Re_n^{prio}(k) = \frac{\mathbf{P}_n^{y}(k)}{\mathbf{P}_{n-1}^{v}(k)}$$

of $\Re_{\min dB}^{prio} = -15.0\,\text{dB}$ be imposed to avoid musical noise. This corresponds to:

$$W_{\min} = \frac{1}{1 + 10^{-\frac{\Re_{\min dB}^{prio}}{10}}}$$

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 is an advantage 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

$$M(\theta) = \frac{1}{2} \cdot \sqrt{\pi\theta} \cdot e^{-\frac{\theta}{2}} \left[(1+\theta) \cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right) \right]$$

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 See [Mar01] for more information.

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 $\mathbf{H}_n^v(k)$ can 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 (below 60 Hz) can be set to 0.

11. **Calculate filter output**. The filter output is defined as follows:

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n^y(k) \cdot \mathbf{Z}_n(k)$$

Inverse DFT and overlap-add. The time-domain filter output is defined as follows:

$$\hat{\mathbf{y}}_{n-1} = \left[\mathbf{0}_{\frac{N}{2}\times\frac{N}{2}} \quad \mathbf{I}_{\frac{N}{2}\times\frac{N}{2}} \right] \cdot \sqrt{\mathbf{w}} \bullet \mathbf{F}^{-1}\hat{\mathbf{Y}}_{n-1} + \left[\mathbf{I}_{\frac{N}{2}\times\frac{N}{2}} \quad \mathbf{0}_{\frac{N}{2}\times\frac{N}{2}} \right] \cdot \sqrt{\mathbf{w}} \bullet \mathbf{F}^{-1}\hat{\mathbf{Y}}_n$$

NOTE. *The algorithm introduces a delay of N/2 samples in the output.*

Header Files

The header files <ippdefs.h> and <ippSR.h> must be included in order to link against any of the EMNS primitives, as shown in the following example code:

```
#include "ippdefs.h"
#include "ippSR.h"

int main()
{
        ...
        /* call EMNS Intel® IPP functions */
        ippsFilterUpdateWiener_32s(pSrcPriorSNRQ15,
        pDSTFilterCoefsQ31, len);
}
```

Definitions and Data Structures

The following sections describe the header files and data structures for the Intel® IPP Ephraim-Malah Noise Suppressor.

Data Structures

There is one structure associated with the OSMS noise floor estimator.

`IppsOSMSParam`

This structure is used internally and should not be modified by the programmer.

Noise Suppressor Filter Update Primitives

The Intel® IPP Noise Suppressor Filter Update primitives are described in the following sections.

FilterUpdateEMNS_32s

Prototype

```
IppStatus ippsFilterUpdateEMNS_32s(const Ipp32s *pSrcWienerCoefsQ31,
   const Ipp32s *pSrcPostSNRQ15,Ipp32s *pDstFilterCoefsQ31, int len);
```

Description

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 0 and 1) 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_32s32s()`, computes the filter coefficients $\mathbf{H}_n(k)$ according to the above equation. Power spectral estimates are typically computed using the `ippsWeightedAdd` primitive.

Input Arguments

- `pSrcWienerCoefsQ31` – pointer to a real-valued vector containing the Q31 format Wiener filter coefficients ($0.0_{Q31} \le$ `pSrcWienerCoefsQ31[k]` $< 1.0_{Q31}$).
- `pSrcPostSNRQ15` – pointer to a real-valued vector containing an estimate of the *a posteriori* signal to noise ratio in Q15 format ($0_{Q15} <$ `pSrcPostSNRQ15` $< 32768.0_{Q15}$)
- `len` – number of elements contained in input and output vectors ($0 <$ `len` < 65536)

Output Arguments

`pDstFilterCoefsQ31` – pointer to a real-valued vector containing the Q31 format filter coefficients ($0.0_{Q31} \le$ `pDstFilterCoefsQ31[k]` $< 1.0_{Q31}$)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrcWienerCoefsQ31`, `pSrcPostSNRQ15`, or `pDstFilterCoefsQ31` is NULL
- `ippStsLengthErr` – illegal value for `len`

FilterUpdateWiener_32s

Prototype

```
IppStatus ippsFilterUpdateWiener_32s (const Ipp32s *pSrcPriorSNRQ15,
   Ipp32s *pDstFilterCoefsQ31, int len);
```

Description

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 $\mathbf{Z}_n(k)$ coefficients are the gains (scalar values between 0 and 1) 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.

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:

$$\mathbf{W}_n^y(k)=\frac{1}{1+\frac{1}{\Re_n^{prio}(k)}}$$

,

$$\Re_n^{prio}(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^v(k)}$$

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`.

Input Arguments

- `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]` $< 32768.0_{Q15}$)
- `len` – number of elements contained in input and output vectors ($0 <$ `len` < 65536)

Output Arguments

`pDstFilterCoefsQ31` – pointer to a real-valued vector containing the Q31 format filter coefficients ($0.0_{Q31} \leq$ `pDstFilterCoefsQ31[k]` $< 1.0_{Q31}$)

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrcPriorSNRQ15` or `pDstFilterCoefsQ31` is NULL
- `ippStsLengthErr` – illegal value for `len`

Noise Floor Estimation Primitives

The Intel® IPP noise floor estimation primitives are described in the following sections.

GetSizeMCRA_32s

Prototype

```
IppStatus ippsGetSizeMCRA_32s (int nFFTSize, int *pDstSize);
```

Description

This function calculates the size in bytes that should be allocated for the `IppOSMSState` state structure required by `ippsUpdateNoisePSDMCRA_32s_I`. Call this function before allocating memory and before calling `ippsInitMCRA`.

Input Arguments

`nFFTSize` – size of the FFT used for noise PSD estimations (8 ≤ `nFFTSize` ≤ 8192)

Output Arguments

`pDstSize` – pointer to the variable to contain the size in bytes

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pDst` is NULL
- `ippStsSizeErr` – illegal value for `nFFTSize`

InitMCRA_32s

Prototype

```
IppStatus ippsInitMCRA_32s_I (int nSamplesPerSec,
   int nFFTSize, IppMCRAState *pDst);
```

Description

This function initializes the state structure for `ippsUpdateNoisePSDMCRA_32s_I`.

Input Arguments

- `nSamplesPerSec` – input sample rate (0 < `nSamplesPerSec` ≤ 48000)
- `nFFTSize` – size of the FFT used to for noise PSD estimation (8 ≤ `nFFTSize` ≤ 8192)

Output Arguments

`pDst` – pointer to the state structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pDst` is NULL
- `ippStsSizeErr` – illegal value for `nFFTSize`
- `ippStsRangeErr` – `nSamplesPerSec` is out of range.

NOTE. *State memory address stored in pDst must be aligned to 32-bit word boundary.*

AltInitMCRA_32s

Prototype

```
IppStatus ippsAltInitMCRA_32s(int nSamplesPerSec, int nFFTSize, int
nUpdateSamples, IppMCRAState *pDst);
```

Description

This function initializes the state structure for `ippsUpdateNoisePSDMCRA_32s_I`.

This function is a candidate to replace `ippsInitMCRA_32s`.

Input Arguments

- `nSamplesPerSec` – input sample rate (8000 < nSamplesPerSec = 48000)
- `nFFTSize` – size of the FFT used to for noise PSD estimation (8 = nFFTSize = 8192)
- `nUpdateSamples` – number of new samples per frame

Output Arguments

- `pDst` – pointer to the state structure

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – pDst is NULL
- `ippStsSizeErr` – illegal value for nFFTSize

- `ippStsRangeErr` – nSamplesPerSec is out of range

NOTE. *This function definition is provided for reference only. As a rule, primitives for the Intel XScale® platform do not allocate memory. This function may exist for other platforms.*

UpdateNoisePSDMCRA_32s_I

Prototype

```
IppStatus ippsUpdateNoisePSDMCRA_32s_I (const Ipp32s
   *pSrcNoisySpeech, IppMCRAState *pSrcDstState, Ipp32s
   *pSrcDstNoisePSD);
```

Description

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].

Input Arguments

- `pSrcNoisySpeech` – pointer to a real-valued vector containing the magnitude squared of the FFT of the noisy speech ($0 \le$ `pSrcNoisySpeech[k]` $< 2^{31}$)
- `pSrcDstState` – pointer to the state structure
- `pSrcDstNoisePSD` – pointer to the output noise power spectrum vector

Output Arguments

- `pSrcDstState` – pointer to the updated state structure
- `pSrcDstNoisePSD` – pointer to the output noise power spectrum vector

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrcNoisySpeech`, `pSrcDstState`, or `pSrcDstNoisePSD` is NULL

Acoustic Echo Canceller

This section describes the Intel® Integrated Performance Primitives that are used to construct an Acoustic Echo Canceller (AEC). The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The AEC consists of a frequency-domain adaptive filter algorithm and a controller.

Intel® IPP AEC primitives support the following features:

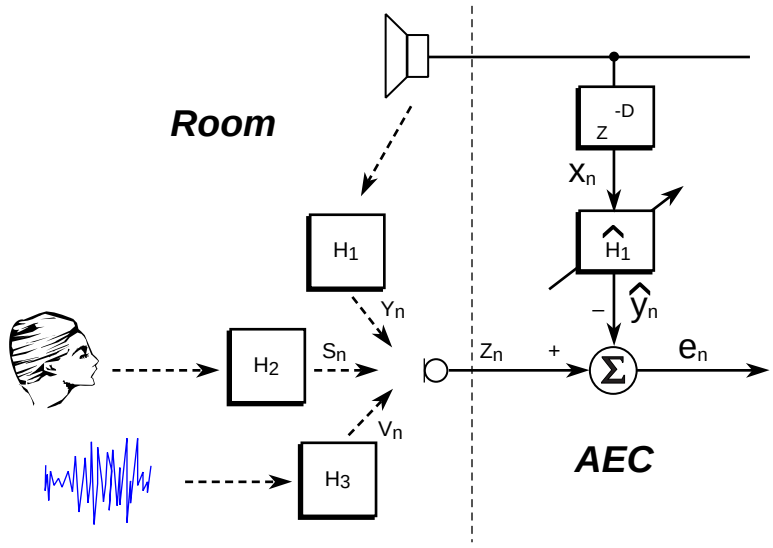
- filtering
- filter coefficient
- step size update
- AEC controller

Acoustic Echo Canceller Architecture

Figure 20-2 shows a typical application of acoustic echo cancellation, in this case, a monophonic speakerphone. Here, audio is played into a room through a loudspeaker and captured at a microphone. The acoustic transfer function between the loudspeaker and the microphone is denoted $\mathbf{H}_1$ and the resulting signal at the microphone is denoted $\mathbf{y}_n$ where n is a time index. At the same time, a person is shown speaking into the microphone. The acoustic transfer function between the person's mouth and the microphone is denoted $\mathbf{H}_2$ and the resulting signal at the microphone is denoted $\mathbf{s}_n$. A noise source (for example, a PC cooling fan) is also active in the room. The acoustic transfer function between the noise source and the microphone is denoted $\mathbf{H}_3$ and the resulting signal at the microphone is denoted $\mathbf{v}_n$. The signals are assumed to combine

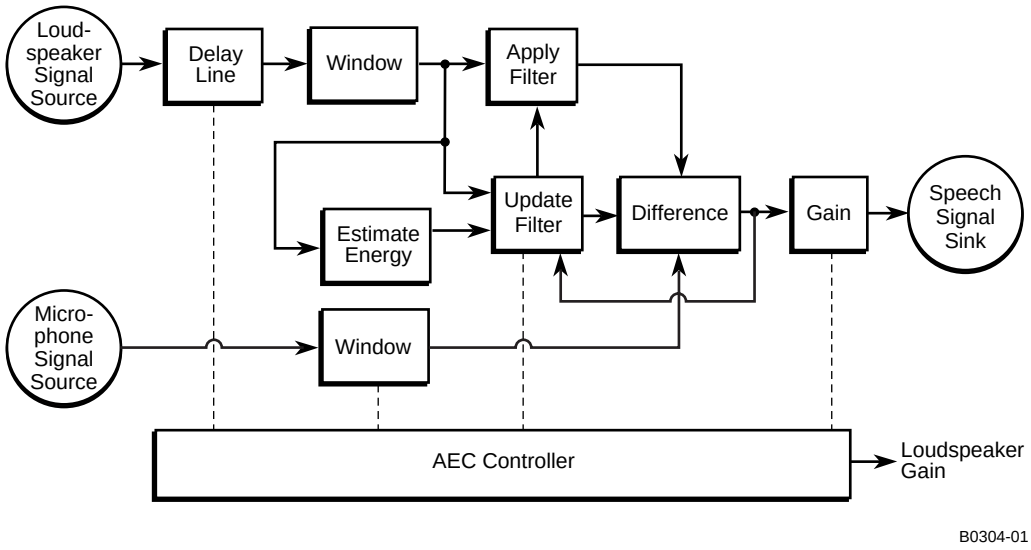
additively at the microphone. The job of the AEC is to accurately estimate the transfer function $\mathbf{H}_1$ and delay **D** so that a filtered copy of the loudspeaker signal can be subtracted from the microphone signal, canceling the echo.

Figure 20-2 Typical Acoustic Echo Cancellation Scenario



The major building blocks of the Acoustic Echo Canceller are shown in Figure 20-3, which shows the steps to apply AEC to a speech signal stream.

Figure 20-3 Major Blocks in an Acoustic Echo Cancellation System



Frequency Domain Block NLMS Adaptive Filter

The following sections describe the algorithm steps for the normalized least mean square (NLMS) adaptive filter.

Algorithm Steps

The frequency domain block normalized least mean square (NLMS) adaptive filter has a relatively simple computational structure and modest complexity, See [Ash94]. The algorithm consists of the following steps:

1. **New input block**.
   The most recently acquired N/2 input samples $\boldsymbol{\chi}_n$ and the previous N/2 input samples $\boldsymbol{\chi}_{n-1}$ make up the new input block $\mathbf{x}_n$.

$$\mathbf{x}_n = \begin{bmatrix} \boldsymbol{\chi}_{n-1} \\ \boldsymbol{\chi}_n \end{bmatrix}$$

2. **Input DFT**. The discrete Fourier transform of the input is calculated.

$$\mathbf{X}_n = \mathbf{F}\mathbf{x}_n$$

   F is the Fourier transform matrix with entries, f(m,n)=exp(-j2πmn/N), where N is the size of the transform

3. **Update input history**. The most recent L frequency-domain input blocks are retained.

$$\mathrm{X}_n = \left[\mathbf{X}_n \mathbf{X}_{n-1} \mathrm{L}\ \mathbf{X}_{n-L+1}\right]$$

4. **Update input energy estimate**.
   The energy at each bin $P_{xx}(k)$ is estimated using a leaky average with $0 < \beta < 1$.

$$\hat{P}_{xx}(k) = (1-\beta)\cdot\hat{P}_{xx}(k) + \beta\cdot\left|\mathbf{X}_n(k)\right|^2 \quad \text{for } k = 0,\ldots,N/2$$

5. **Compute adaptive step sizes**.
   The adaptive step sizes are computed using the classical normalized LMS approach with

$$0 < \mu << 1 \text{ and } P_{x\min} > 0.$$

$$\mathrm{M}(k) = \begin{cases} \dfrac{\mu}{\hat{P}_{xx}(k)}, & \hat{P}_{xx}(k) > P_{x\min} \\ \dfrac{\mu}{P_{x\min}}, & \hat{P}_{xx}(k) \le P_{x\min} \end{cases} \quad \text{for } k = 0,\ldots,N/2$$

6. **Compute the filter output.**
   The adaptive filter uses the low-latency structure described in [Ash94]. The filter impulse response is evenly divided into non-overlapping segments. In general, this corresponds to rewriting the convolution:

$$y(m) = \sum_{i=0}^{I-1} h(i) \cdot x(m-i)$$

   as:

$$y(m) = \sum_{j=0}^{J-1} \sum_{k=0}^{I/J-1} h\left(k + j\left(I/J\right)\right) \cdot x\left(m - j\left(I/J\right) - k\right) = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} h_j\left(k\right) \cdot x_j\left(m-k\right)$$

   The filter output is computed in the frequency domain as:

$$\hat{\mathbf{Y}}_n(k) = \sum_{i=0}^{L-1} \mathbf{X}_{n-i}(k) \cdot \mathbf{H}_i(k) \text{ for } k = 0,\ldots,N/2$$

   where $\mathbf{H}_i$ has been properly constrained (see step 9) such that the underlying convolution is linear.

7. **Constrain the error**. First the time-domain filter output is computed via an inverse discrete Fourier transform.

$$\hat{\mathbf{y}}_n = \mathbf{F}^{-1}\hat{\mathbf{Y}}_n$$

   In the acoustic echo cancellation application, the error $\mathbf{e}_n$ is the difference between the microphone input $\mathbf{y}_n$ and the adaptive filter output.

$$\mathbf{e}_n = \mathbf{y}_n - \hat{\mathbf{y}}_n$$

   To properly implement the adaptation step (see step 8) in the frequency-domain, the time-domain error must be constrained as follows:

$$\mathbf{E}_n = \mathbf{F}\begin{bmatrix} \mathbf{0} \\ \boldsymbol{\varepsilon}_n \end{bmatrix}$$

   The n^{th} block of the echo canceller output is defined here as:

$$\boldsymbol{\varepsilon}_n = \left[\mathbf{e}_n(N/2) \text{L } \mathbf{e}_n(N-1)\right]^T$$

8. **Update adaptive filter**.
   The adaptive filter update is carried out in the frequency-domain. Segments of the adaptive filter response are updated independently by

$$\mathbf{H}_i = \mathbf{H}_i + \mathbf{M}_n \bullet \mathbf{X}^*_{n-i} \bullet \mathbf{E}_n \quad \text{for } i = 0,...,L-1$$

   where $\bullet$ denotes pointwise multiplication. Here $\mathbf{X}^*_{n-i} \bullet \mathbf{E}_n$ is recognized as the i^{th} segment of the gradient.
9. **Constrain the adaptive filter response**.
   To exactly implement the linear convolution (filtering) operation in the frequency domain, it is necessary to apply a constraint. The filter response is constrained by

$$\mathbf{h}_i = \mathbf{F}^{-1}\mathbf{H}_i$$

$$\mathbf{H}_i = \mathbf{F}\begin{bmatrix}\boldsymbol{\eta}_i \\ \mathbf{0}\end{bmatrix}$$

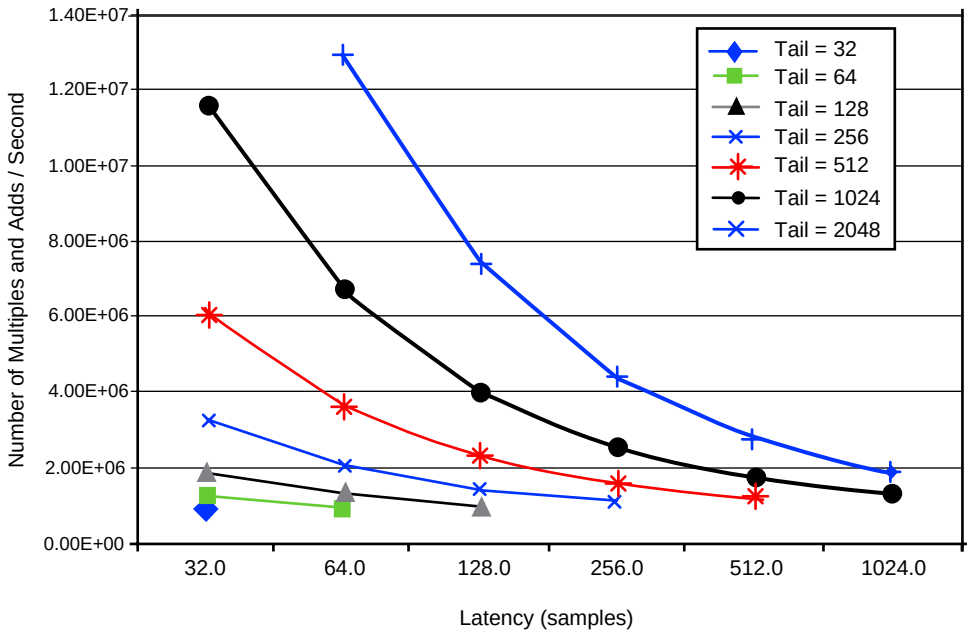
   for $i = 0,...,L-1$. Here, the first half of each segment of the impulse response $\boldsymbol{\eta}_i = [\mathbf{h}_i(0) \text{L} \ \mathbf{h}_i(N/2-1)]^T$ is retained and the rest set to 0.

Computational Complexity

The computational complexity of the adaptive filter is illustrated in Figure 20-4. This illustrates the trade-off between low latency and filter tail length. For example, when latency is fixed at 8 milliseconds (64 samples per block at 8000 Hz sample rate), a tail length of 128 milliseconds

requires about 7 MOPS (million operations per second). Reducing the latency to 4 milliseconds increases computations to 11.5 MOPS. Increasing tail length to 256 milliseconds increases computations to 13 MOPS.

Figure 20-4 Computational Complexity of FD-NLMS Adaptive Filter



AEC Controller

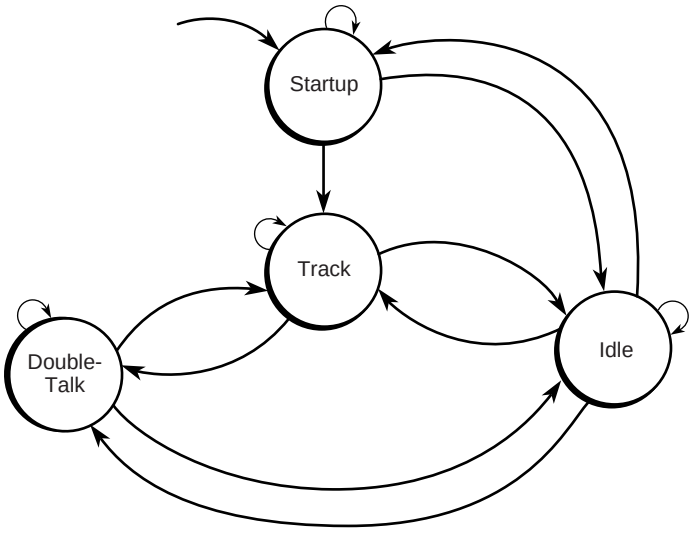
To understand the need for an AEC controller, consider Figure 20-2. The adaptive filter $\hat{H}_1$ modifies its coefficients such that the filtered loudspeaker signal $\hat{y}_n$ best matches the microphone signal z_n in the least squares sense. When the loudspeaker signal is active, s_n and v_n are weak, and the loudspeaker-to-microphone transfer function H_1 is nearly linear, $\hat{H}_1$ will approach the true transfer function, H_1. If the loudspeaker signal is active and s_n or v_n are also active (a situation known as "double-talk") then $\hat{H}_1$ will not converge to the proper solution. Even when s_n or v_n are weak, they may still cause the adaptive filter to diverge when the loudspeaker signal is also weak or silent.

The AEC controller addresses these problems by adjusting the adaptive filter step size (μ in Step 5 of "Algorithm Steps") such that the filter converges rapidly when the loudspeaker signal is present at the microphone but does not diverge quickly during double-talk or insufficient excitation of the microphone by the loudspeaker. The controller also manages playback and output gain to block echo when the adaptive filter has diverged.

AEC Algorithm Description

The AEC controller update primitive ippsControllerUpdateAEC_32s is based on the simple state machine shown in Figure 20-5.

Figure 20-5 State Diagram of AEC Controller



State transitions are governed by simple full-band energy measurements:

$$E_n^e = \sum_{k=0}^{N/2-1} e_n^2(k)$$

error energy

$$E_n^z = \sum_{k=0}^{N/2-1} z_n^2(k)$$

microphone energy

$$E_n^x = \sum_{k=0}^{N/2-1} x_n^2(k)$$ loudspeaker energy

$$\overline{E}_n^z = (1-\gamma)\cdot \overline{E}_{n-1}^z + \gamma \cdot E_n^z$$ smoothed microphone energy

$$\overline{E}_n^e = (1-\gamma)\cdot \overline{E}_{n-1}^e + \gamma \cdot E_n^e$$ smoothed error energy

$$ERLE_n = \overline{E}_n^z / \overline{E}_n^e$$ echo return loss enhancement

The smoothing constant is $\gamma = 0.005$ when the block update rate is 8 milliseconds (for example, 64 samples at 8000 Hz sample rate) and is adjusted to achieve the same smoothing rate at other block update rates. Four conditions are defined using the energy measurements:

Converged (C) $ERLE_n > T^{ERLE}$

Receive Active (R) $E_n^x > T_n^x$

Mic Active (M) $E_n^z > T_n^z$

No Double-talk (N) $E_n^x \cdot G_{xz} > E_n^z$

The thresholds are set as follows: the ERLE threshold T^{ERLE} is 2.0 (3 dB), the loudspeaker activity threshold T_n^x is $6.0 \cdot \min\{E_n^x, E_{n-1}^x, ..., E_{n-windowsize}^x\}$ (8 dB above the local minimum), the microphone activity threshold T_n^z is $6.0 \cdot \min\{E_n^z, E_{n-1}^z, ..., E_{n-windowsize}^z\}$ (8 dB above the local minimum), and the total echo return loss between the loudspeaker and microphone G_{xz} is 0.25 (-6 dB). The minimum-tracking window size corresponds to 2 seconds.

The "No Double-talk" condition defined here is crude but effective. With respect to Figure 20-2, the total echo return loss G_{xz} includes attenuation of the signal x_n by the output amplifier, loudspeaker, room transfer function, microphone, and microphone pre-amplifier circuitry. G_{xz} is set to -6 dB since directional microphones achieve an ERL of 6dB in many cases. If the signal x_n coming out the loudspeaker drops 6 dB or more before being picked up at the microphone then it is assumed to be uncorrupted by the local talker. It may be necessary to carefully design the hardware platform to ensure that this assumption is true.

State transitions are triggered by the four conditions according to Table 20-1.

Table 20-1 State Transition Conditions

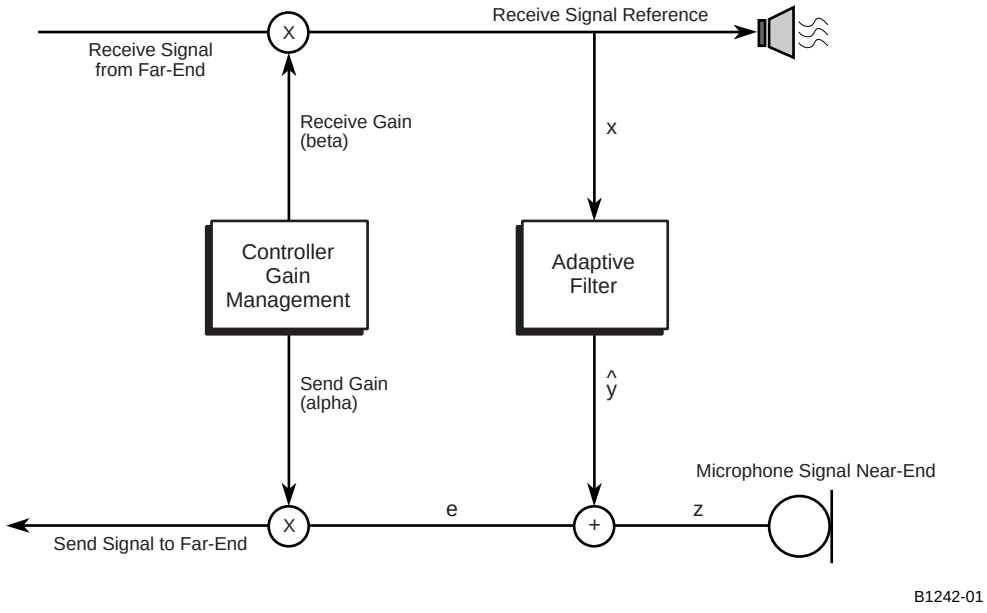
C	R	M	N	Current State	Next State	μ	α	β
F	T	T	T	Startup	Startup	$4\mu_c$	0.0	1.0
F	T	F	X	Startup	Startup	μ_c	0.0	1.0
F	T	T	F	Startup	Startup	0.0	1.0	0.0
X	F	T	X	Startup	Startup	0.0	1.0	0.0
X	F	F	X	Startup	Startup	0.0	0.5	0.5
T	T	T	T	Startup	Track	$4\mu_c$	1.0	1.0
T	T	T	F	Startup	Track	μ_c	1.0	1.0
T	T	F	T	Startup	Track	μ_c	0.0	1.0
T	T	F	F	Startup	Idle	0.0	0.5	0.5
T	T	T	X	Track	Track	$2\mu_c$	1.0	1.0
X	T	F	X	Track	Track	μ_c	1.0	1.0
F	T	T	X	Track	Double-talk	0.0	1.0	1.0
X	F	X	X	Track	Idle	0.0	0.5	0.5
F	T	T	T	Idle	Startup	$2\mu_c$	0.0	1.0
X	F	T	X	Idle	Idle	0.0	1.0	1.0
X	F	F	X	Idle	Idle	0.0	0.5	0.5
T	T	T	T	Idle	Track	$2\mu_c$	1.0	1.0
T	T	T	F	Idle	Track	$\mu 2_c$	1.0	1.0
X	T	F	X	Idle	Track	$2\mu_c$	1.0	1.0
X	T	T	F	Idle	Double-talk*	0.0	1.0	1.0
F	T	T	X	Double-talk	Double-talk	0.0	1.0	1.0
T	T	T	T	Double-talk	Track	$2\mu_c$	1.0	1.0
T	T	T	F	Double-talk	Track	μ_c	1.0	1.0
X	T	F	X	Double-talk	Track	μ_c	1.0	1.0
T	T	T	F	Double-talk	Track*	0.0	1.0	1.0
X	F	T	X	Double-talk	Idle	0.0	1.0	1.0
X	F	F	X	Double-talk	Idle	0.0	0.5	0.5

The step size (μ), loudspeaker gain (β), and output gain (α) are adjusted as shown. The value μ_c refers to the constant step size that is set during controller initialization (`muQ31` in the `IppAECNLMSParam` structure — see "ControllerInitAEC_32s". Transitions marked with an "*" are redundant but take precedence when the ERLE exceeds the threshold for the "converged" (C) condition but is still below 40% of the peak ERLE in dB. That is,

$$\log\left(T^{ERLE}\right) < \log\left(ERLE_n\right) < 0.4 \cdot \log\left(\max\left\{ERLE_n, ERLE_{n-1}, \ldots, ERLE_{n-windowsize}\right\}\right)$$

The values, α and β, in the table correspond to the send (output) and receive (loudspeaker) gain targets, respectively. The instantaneous gains are modified gradually to avoid audible flutter. When one of the gain targets changes, the instantaneous gain is increased or decreased linearly to meet the target in exactly 100 milliseconds. Figure 20-6 illustrates that application of the send and receive gain in an AEC system.

Figure 20-6 Application of AEC Send and Receive Gains by the AEC Controller



Definitions and Data Structures

The following sections describe the header files and data structures for the Intel® IPP Acoustic Echo Canceller.

Header Files

The header files <ippdefs.h> and <ippSR.h> must be included in order to link against any of the AEC primitives, as shown in the following example code:

```
#include "ippdefs.h"
#include "ippSR.h"

int main()
{
        ...
        /* call AEC Intel® IPP functions */
        ippsStepSizeUpdateAECNLMS_32s(pInputPSD, pStepSize, maxStepSize,
minInputPSD, len);
}
```

Data Structures

The AEC Controller requires access to many parameters. These parameters are stored in the following structure:

```
typedef struct {
Ipp16s *pMicrophone;        /* pointer to mic samples */
Ipp16s *pLoudspeaker;       /* pointer to speaker samples */
Ipp16s *pError;             /* pointer to error samples */
Ipp32s *pAFInputPSD;        /* pointer to filter input PSD */
Ipp32sc **ppAFCoefs;        /* pointer to filter segment array */
Ipp32s muQ31;               /* fixed step size (Q31 value in (0,1)) */
Ipp32s AECOutGainQ30;       /* AEC output gain (Q30 value in (0,1)) */
Ipp32s speakerGainQ30;      /* loudspeaker gain (Q30 value in (0,1)) */
int numSegments;            /* number of segments of filter tail */
int numFFTBins;             /* number of FFT bins (FFTSize / 2 + 1) */
int numSamples;             /* mic, error, loudspeaker frame size */
int sampleRate;             /* sample rate (Hertz) */

} IppAECNLMSParam;
```

numSamples represents the size of the non-overlapping part of the frames. The valid ranges for the elements of the IppAECNLMSParam structure are shown in Table 20-2.

Table 20-2 Valid Ranges for Elements of IppAECNLMSParam

Member	Range
pMicrophone	Samples are in $[-2^{15}, +2^{15})$.
pLoudspeaker	Samples are in $[-2^{15}, +2^{15})$.
pError	Samples are in $[-2^{15}, +2^{15})$.
pAFInputPSD	PSD coefficients are in range $[0, 2^{31})$.
ppAFCoefs	Real and imaginary parts of coefficients are in $[-2^{31}, +2^{31})$.
muQ31	Q31 scalar in range [0 , 1).
AECOutGainQ30	Q30 scalar in range [0 , 1).
speakerGainQ30	Q30 scalar in range [0 , 1).
numSegments	In range [1 , 255].
numFFTBins	Is one of {17, 33, 65, 129, 257, 513, 1025,2049, 4097}.
numSamples	In range [1 , 4096]
sampleRate	In range [1 , 48000]

The pointers in the structure should always point to the most recent block or coefficient set.

The controller keeps its internal state in an additional structure called:

IppAECCtrlState.

This structure is used internally and should not be modified by the programmer. Filter tail lengths of up to 2 seconds are supported.

The structure IppAECScaled32s provides a representation for scaled 32-bit signed integers.

```
typedef struct {
     Ipp32sval;
     Ipp32ssf;
} IppAECScaled32s;
```

A variable x of type IppAECScaled32s represents the value x.val * $2^{x.sf}$. The structure IppAECScaled32s provides for efficient computations on the Intel XScale® microarchitecture processors compared to floating point, at the cost of increased storage. Typically, the val field is "left justified", meaning that it has been left shifted such that:

$2^{30} = x.val < 2^{31}$

or

$-2^{30} < x.val = -2^{31}$

AEC Filter Primitives

The Intel® IPP AEC Filter primitive is described in the following section.

FilterAECNLMS_CCS_32sc_Sfs

Prototype

```
IppStatus ippsFilterAECNLMS_CCS_32sc_Sfs (const Ipp32sc
   **ppSrcSignalIn, const Ipp32sc **ppSrcCoefs, Ipp32sc *pDstSignalOut,
   int numSegments, int len, int scaleFactor);
```

Description

This function implements Step 6 in "Algorithm Steps" to compute the frequency-domain adaptive filter output.

Input Arguments

- `ppSrcSignalIn` – pointer to an array of pointers to the most recent input blocks (for example, $\mathbf{X}_n$, $\mathbf{X}_{n-1}$, …, $\mathbf{X}_{n-L+1}$). These are the complex-valued vectors that contain the FFT of the input signal stored in Intel® IPP CCS format.
- `ppSrcCoefs` – pointer to an array of pointers to the filter coefficients vectors. These are the complex-valued vectors containing the filter coefficients stored in Intel® IPP CCS format.
- `numSegments` – the number of filter segments (L) (0 < `numSegments` < 256)
- `len` – number of elements contained in the input and output vectors and each filter segment (0 < `len` ≤ 4097)
- `scaleFactor` – saturation fixed scale factor (-32 < `scaleFactor` < 32)

Output Arguments

`pDstSignalOut` – pointer to the complex-valued filter output vector stored in Intel® IPP CCS format

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – pDst is NULL

- `ppSrcSignalIn`, `ppSrcCoefs`, or `pDstSignalOut` is NULL
- `ippStsLengthErr` – illegal value for `len`
- `ippStsRangeErr` – `numSegments` or `scaleFactor` is out of range

AEC Filter Update Primitives

The Intel® IPP AEC Filter Update primitive is described in the following section.

CoefUpdateAECNLMS_CCS_32sc_ISfs

Prototype

```
IppStatus ippsCoefUpdateAECNLMS_CCS_32sc_ISfs (const
   IppAECScaled32s*Ipp32s pSrcStepSize, const Ipp32sc
   **ppSrcFilterInput, const Ipp32sc *pSrcError, Ipp32sc
   **ppSrcDstCoefs, int numSegments, int len, int scaleFactorCoef);
```

Description

This function implements Step 8 in "Algorithm Steps" to update the adaptive filter coefficients.

NOTE. *It is assumed that the filter coefficients have been multiplied by* $2^{\text{scaleFactorCoef}}$ *prior to calling this function. If overflow occurs, the result of the update is saturated.*

Input Arguments

- `pSrcStepSize` – pointer to the scaled integer step size vector
  It is recommended that the `val` field be left justified. That is,
  ($0.0_{Q31} \leq$ `pSrcStepSize[k]` $< 2^{31}$).
  and the `sf` field be restricted to the range,
  (-64 = `pSrcStepSize[k].sf` < 64).

- `ppSrcFilterInput` – pointer to an array of pointers to the most recent input blocks. (for example, $\mathbf{X}_n$, $\mathbf{X}_{n-1}$, …, $\mathbf{X}_{n-L+1}$). These are the complex-valued vectors that contain the FFT of the input signal stored in Intel® IPP CCS format.
- `ppSrcDstCoef` – pointer to an array of pointers to the filter coefficient vectors. These are the complex-valued vectors containing the filter coefficients.
- `pSrcError` – pointer to the complex-valued vector containing the filter error stored in Intel IPP CCS format
- `numSegments` – the number of filter segments (L) (0 < `numSegments` < 256)
- `len` – number of elements contained in each input and output vector (0 < `len` ≤ 4097)
- `scaleFactorCoef` – fixed scale factor for filter coefficients (0 ≤ scaleFactor < 32). Typically, the same scale factor is used in both `ippsCoefUpdateAECNLMS_32sc_I` and `ippsFilterAECNLMS_32sc_Sfs`.

Output Arguments

`ppSrcDstCoefs` – pointer to an array of pointers to the filter coefficient vectors after the update. These are the complex-valued vectors containing the filter coefficients stored in Intel® IPP CCS format.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `ppSrcStepSize`, `ppSrcFilterInput`, `pSrcError`, or `ppSrcDstCoefs` is NULL
- `ippStsLengthErr` – illegal value for `len`.
- `ippStsRangeErr` – `pSrcStepSize[i].val<0`, `numSegments` out of range, or `scaleFactorCoef` out of range

NOTE. The filter coefficients are assumed to have been multiplied by $2^{\text{scaleFactorCoef}}$ prior to the call to this function. If overflow occurs, the result of the update is saturated.

AEC Step Size Update Primitives

The Intel® IPP AEC Step Size Update primitive is described in the following section.

StepSizeUpdateAECNLMS_32s

Prototype

```
IppStatus ippsStepSizeUpdateAECNLMS_32s (const Ipp32s *pSrcInputPSD,
   Ipp32s muQ31, Ipp32s IppAECScaled32s maxStepSize, Ipp32s minInputPSD,
   IppAECScaled32s *pDstStepSize, int len);
```

Description

This function implements Step 5 in "Algorithm Steps" to compute the adaptive step size.

Input Arguments

- `pSrcInputPSD` – pointer to real-valued vector containing the input power spectrum estimate
- `muQ31` – real-valued Q31 scalar ($0.0_{Q31} \le$ `muQ31` $< 1.0_{Q31}$).
- `maxStepSize` – left justified scaled integer representing the maximum step size allowed (usually = `mu` / `minInputPSD` > 0).
- `minInputPSD` – minimum value of input PSD for which step size update should occur (0 < `minInputPSD`).
- `len` – number of elements contained in input and output vectors (0 < `len` ≤ 4097).

Output Arguments

`pDstStepSize` – pointer to the left justified scaled integer output vector

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrcInputPSD` or `pDstStepSize` is NULL
- `ippStsLengthErr` – illegal value for `len`
- `ippStsRangeErr` – `pSrcInputPSD[i]<0`, `muQ31 <0`, `minInputPSD <0`, or `maxStepSize.val <0`

AEC Controller Primitives

The Intel® IPP AEC Controller primitives are described in the following sections.

ControllerGetSizeAEC_32s

Prototype

```
IppStatus ippsControllerGetSizeAEC_32s (int *pDstSize);
```

Description

This function returns the size (in bytes) of the AEC controller state structure. This information enables you to allocate the correct amount of memory.

Input Arguments

none

Output Arguments

`pDstSize` – pointer to variable that holds the size in bytes of the state

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pDstSize` is NULL

ControllerInitAEC_32s

Prototype

```
IppStatus ippsControllerInitAEC_32s (const IppAECNLMSParam
   *pSrcParams, IppAECCtrlState *pDstState);
```

This function initializes the AEC controller state structure.

Input Arguments

`pSrcParams` – pointer to AEC parameters structure

Output Arguments

`pDstState` – pointer to the AEC controller state structure. State memory address stored in `pDstState` must be aligned to 32-bit word boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrcParams` or `pSrcDstState` is NULL
- `ippStsRangeErr` – `pSrcParams->numSamples` or `pSrcParams->sampleRate` is out of range. See "Data Structures" for description of range.

ControllerUpdateAEC_32s

Prototype

```
IppStatus ippsControllerUpdateAEC_32s (const IppAECNLMSParam
   *pSrcParams, IppAECCtrlState *pSrcDstState, Ipp32s *pDstMuQ31, Ipp32s
   *pDstAECOutGainQ30, Ipp32s *pDstSpeakerGainQ30);
```

Description

This function implements the energy-based AEC controller described in "AEC Controller". It sets the fixed adaptation step size (mu), the AEC output gain, and the loudspeaker gain based on various system parameters. For example, during "double-talk" (the condition where a person is speaking into the microphone while audio is playing through the loudspeaker) the adaptation step size is reduced. This algorithm is Intel-proprietary.

The `ippsControllerUpdateAEC_32s` primitive provides basic AEC control functionality applicable to a variety of Intel XScale® technology-based devices. However, best performance will always be obtained by designing the controller for the specific hardware and physical device geometry in any given application.

The purpose of the basic AEC controller is to provide baseline AEC performance during the prototyping stage. It is expected that this basic controller will be replaced with a custom controller in a final product. Limitations of the basic AEC controller include the following:

- does not maintain adaptive filter convergence for double-talk durations of more than about two seconds. This case rarely occurs during normal conversation.
- does not adapt when the loudspeaker energy is constant for more than a few hundred milliseconds. This case rarely occurs in practice. However, it may have the surprising side-effect that playing wideband stationary noise through the loudspeaker does not cause the adaptive filter to converge rapidly (as would be expected when no controller is present).
- assumes that there is roughly a 6 dB drop between the loudspeaker and microphone. This is highly dependent on the physical layout and directionality of the loudspeaker and microphone as well as the analog playback gain. Consequently, an AEC which uses this controller may not perform well with an external amplifier at high volume levels.

Input Arguments

- `pSrcParams` – pointer to AEC parameters structure
- `pSrcDstState` – pointer to AEC controller state structure

Output Arguments

- `pSrcDstState` – pointer to AEC controller state structure
- `pDstMuQ31` – pointer to real-valued Q31 scalar ($0.0\ Q_{31} =$ `pDstMuQ31` $< 1.0\ Q_{31}$)
- `pAECOutGainQ30` – pointer to real-valued Q30 scalar ($0.0\ Q_{30} =$ `pAECOutGainQ30` $= 1.0\ Q_{30}$).
- `pDstSpeakerGainQ30` – pointer to real-valued Q31 scalar ($0.0\ _{Q30} \leq$ `pDstSpeakerGainQ30` $< 1.0\ _{Q30}$).

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrcParams`, `pSrcDstState`, `pDstMuQ31`, `pDstAECOutGainQ30`, or `pDstSpeakerGainQ30` is NULL

Voice Activity Detector

This section describes the Intel® Integrated Performance Primitives that are used 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.

Intel® IPP VAD primitives support the following features:

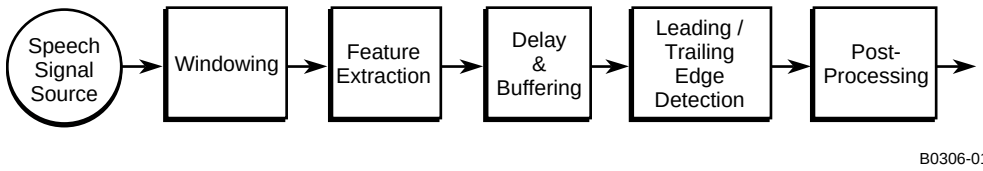
- Peak picking
- Periodicity

- Zero crossing rate

Voice Activity Detector Architecture

A typical voice activity detector is illustrated in Figure 20-7. 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 subband 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. In low-delay implementations, you can eliminate false detections by applying post-processing (for example, median filtering).

Figure 20-7 Major Blocks in a Voice Activity Detector



Definitions and Data Structures

The following sections describe the header files and data structures for the Intel® IPP Voice Activity Detector.

Header Files

The header files <ippdefs.h> and <ippSR.h> must be included in order to link against any of the VAD primitives, as shown in the following example code:

```
#include "ippdefs.h"
#include "ippSR.h"

int main()
{
        ...
        /* call VAD Intel® IPP functions */
```

```
        ippsPeriodicityLSPE_16s(pSignal, len, pPeriodicityQ15,
        pPeriod, maxPeriod, minPeriod);
}
```

Data Structures

There are no special VAD data structures.

Voice Activity Detection Primitives

The Intel® IPP VAD primitives are described in the following sections.

FindPeaks_32s8u

Prototype

```
IppStatus ippsFindPeaks_32s8u (const Ipp32s *pSrc, Ipp8u *pDstPeaks,
   int len, int searchSize, int movingAvgSize);
```

Description

This function identifies peaks in the input vector, places a value of 1 in the output vector at the locations of the peaks, and places a 0 (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.

Input Arguments

- `pSrc` – pointer to an input vector
- `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)

Output Arguments

`pDstPeaks` – pointer to the output vector containing a one in positions corresponding to a peak in the input vector and zeros elsewhere

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – `pSrc` or `pDstPeaks` is NULL
- `ippStsLengthErr` – `len` is out of range
- `ippStsSizeErr` – `searchSize` or `movingAvgSize` are out of range

PeriodicityLSPE_16s

Prototype

```
IppStatus ippsPeriodicityLSPE_16s (const Ipp16s *pSrc, int len,
   Ipp16s *pPeriodicityQ15, int *period, int maxPeriod, int minPeriod);
```

Description

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.

Input Arguments

- `pSrc` – pointer to an input speech vector
- `len` – number of elements contained in the input vector subject to $6 < \texttt{len} \le \texttt{min}(16*\texttt{minPeriod}, 1024)$
- `maxPeriod` – maximum period to search ($0 < \texttt{maxPeriod} < \texttt{len}$)
- `minPeriod` – Minimum period to search ($6 \le \texttt{minPeriod} < \texttt{maxPeriod}$)

Output Arguments

- `pPeriodicityQ15` – pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ($0.0_{Q15} \le$ `pPeriodicityQ15` $\le 1.0_{Q15}$)
- `period` – the period (in samples) that minimizes the LSPE cost function

Returns

- `ippStsNoErr` – no error
- `ippStsLengthErr` – `len` is out of range
- `ippStsRangeErr` – `maxPeriod` or `minPeriod` is out of range
- `ippStsNullPtrErr` – `pSrc` or `pPeriodicityQ15` is NULL

Periodicity_32s16s

Prototype

```
IppStatus ippsPeriodicity_32s16s (const Ipp32s *pSrc, int len,
   Ipp16s *pPeriodicityQ15, int *period, int maxPeriod, int minPeriod);
```

Description

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.

Input Arguments

- `pSrc` – pointer to an input vector containing non-negative entries
- `len` – number of elements contained in the input vector ($0 <$ `len` ≤ 4096)
- `maxPeriod` – maximum period to search (`minPeriod` $<$ `maxPeriod` $<$ `len`)
- `minPeriod` – minimum period to search ($0 <$ `minPeriod` $<$ `maxPeriod`)

Output Arguments

- `pPeriodicityQ15` – pointer to the Q15 format value corresponding to the normalized sum of the largest harmonic sampling ($0.0_{Q15} \le$ `pPeriodicityQ15` $\le 1.0_{Q15}$)
- `period` – pointer to the period (in samples) that provided the maximum-energy harmonic sampling

Returns

- `ippStsNoErr` – no error
- ippStsLengthErr – `len` is out of range.
- `ippStsRangeErr` – `pSrc[k]`, `maxPeriod`, or `minPeriod` is out of range
- `ippStsNullPtrErr` – `pSrc`, `period`, or `pPeriodicityQ15` is NULL

Speech Recognition

This section describes the Intel® Integrated Performance Primitives that are used to construct a speech recognition feature extractor (FE) and feature encoder (source/channel coder). The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The FE consists of Mel-frequency cepstral coefficient feature extraction and compression by split-vector quantization. It is possible to construct an ETSI Aurora I compliant feature extractor using these primitives. See [ETSI00].

Intel® IPP FE primitives support the following features:

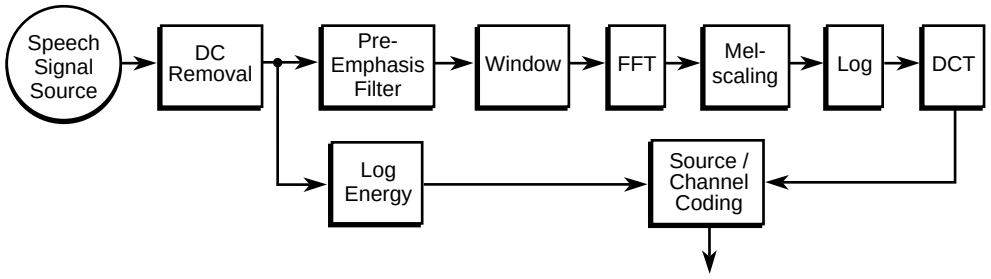
- Offset compensation
- Mel-filterbanks
- DCT and liftering
- Split-vector quantization

Speech Recognition Feature Extractor Architecture

A typical feature extractor for Mel-frequency cepstral coefficients (MFCCs) is illustrated in Figure 20-8. The speech signal is processed by removing its DC component and pre-emphasis filtering. It is then windowed and separated into (possibly overlapping) frames. The discrete Fourier transform of each frame is averaged into sub-bands according to the Mel scale. A discrete

cosine transform is then applied to the log of the Mel-scale filter output. Feature extraction is then performed. Source and channel coding may then be applied to reduce the data rate and increase robustness against channel errors.

Figure 20-8 Major Blocks in a Mel-frequency Cepstral Coefficient Feature Extractor



Definitions and Data Structures

The following sections describe the header files and data structures for the Intel® IPP Speech Recognition Feature Extractor.

Header Files

The header files <ippdefs.h> and <ippSR.h> must be included in order to link against any of the FE primitives, as shown in the following example code:

```
#include "ippdefs.h"
#include "ippSR.h"

int main()
{
        ...
        /* call SR Intel® IPP functions */
ippsEvalFBank_32s_Sfs(pSignalPSD, pMelFiltered, pFBank, nScaleFactor);
   }
```

Data Structures

Intel® IPP Speech recognition primitives use the following feature extraction data structures:

`IppsFBankState_32s`

`IppsDCTLifterState_16s`

`IppsCdbkState_16s`

These structures are used internally and should not be modified by the programmer.

Speech Recognition Feature Extraction Primitives

The Intel® IPP Speech Recognition Feature Extraction primitives are described in the following sections.

CompensateOffsetQ15_16s

Prototype

```
IppStatus ippsCompensateOffset_16s (const Ipp16s* pSrc, Ipp16s* pDst,
   int len, Ipp16s* pSrcDstPrevInputSample, Ipp16s prevOutputSample,
   Ipp16s valQ15);
IppStatus ippsCompensateOffset_16s_I (Ipp16s* pSrcDst, int len, Ipp16s*
   pSrcDstPrevInputSample, Ipp16s prevOutputSample, Ipp16s valQ15);
```

Description

Vector offset compensation – This function removes the offset of the input signals. The destination vector is calculated as follows:

$$pDst[0] = pSrc[0] - pSrcDstPrevInputSample[0] + val * prevOutputSample$$

$$pDst[k] = pSrc[k] - pSrc[k-1] + val * pDst[k-1], \quad k = 1, \ldots, len-1$$

$$pSrcDstPrevInputSample[0] = pSrc[len-1]$$

Input Arguments

- `pSrc`, `pSrcDst` – pointer to the input vector
- `len` – number of elements contained in both the input and output vectors (0 < `len` < 65536)
- `pSrcDstPrevInputSample` – pointer to the previous input sample (`pSrc[-1]`)

- `prevOutputSample` – pointer to the previous output sample (`pDst[-1]`)
- `valQ15` – Q15 format filter coefficient (0.0_{Q15} val ≤ Q15 < 1.0_{Q15})

Output Arguments

- `Dst`, `pSrcDst` – pointer to the output vector
- `pSrcDstPrevInputSample` – pointer to the previous input sample (That is: pSrc[-1])

Returns

- `ippStsNoErr` – no error
- `ippStsSizeErr` – indicates an error when `valQ15` is out of range or when `len` is less than or equal to 0
- `ippStsNullPtrErr` – indicates an error when `pSrc`, `pDst`, `pSrcDst`, or `pSrcDst0` pointer is NULL

DCTLifter_32s16s_Sfs

Prototype

```
IppStatus ippsDCTLifter_32s16s_Sfs (const Ipp32s* pSrc, Ipp16s* pDst,
   const IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
```

Description

DCT and liftering – This function first performs the discrete cosine transform then lifters the DCT coefficients. The DCT coefficients are calculated according to the formula:

$$pDst[k] = pLifter[k] \cdot \sum_{j=1}^{lenDCT} pSrc[j] \cdot \cos\left(\frac{\pi k(j-0.5)}{lenDCT}\right), \quad k = 0,\ldots,lenCeps-1$$

$$pDst[lenCeps] = pLifter[lenCeps] \cdot \sum_{j=1}^{lenDCT} pSrc[j]$$

NOTE. C0 is stored as the last element of the output vector. The values, `lenCeps`, `lenDCT`, and `pLifter`, are as defined in "ippsDCTLifterInit_MulC0_16s".

Input Arguments

- `pSrc` – pointer to the non-negative log magnitude spectrum coefficients (`pSrc[k]` ≥ 0)
- `pDCTLifter` – pointer to the DCT / lifter structure
- `scaleFactor` – saturation fixed scale factor (scaled primitives only) (-16 < `scaleFactor` < 16)

Output Arguments

`pDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsSizeErr` – indicates an error if `pSrc[k]` < 0 or `scaleFactor` is out of range
- `ippStsNullPtrErr` – indicates an error when `pSrc`, `pDst`, or `pDCTLifter` pointers are NULL

DCTLifterGetSize_MulC0_16s

Prototype

```
IppStatus ippsDCTLifterGetSize_MulC0_16s (int lenDCT, int lenCeps,
   int *pSize);
```

Description

DCT structure size calculation – This function computes the size in bytes required for the DCT structure and associated storage.

Input Arguments

- `lenDCT` – length of the `DCT` ($0 < $ `lenDCT` ≤ 8192)
- `lenCeps` – number of the output coefficients not including C0 ($0 <$ `lenCeps` $\leq$ `lenDCT`)

Output Arguments

`pSize` – pointer to the variable to contain the size in bytes

Returns

- `ippStsNoErr` – no error
- `ippStsSizeErr` – indicates an error if `lenDCT` or `lenCeps` are out of bounds
- `ippStsNullPtrErr` – indicates an error when `pSize` pointer is NULL

ippsDCTLifterInit_MulC0_16s

Prototype

```
IppStatus ippsDCTLifterInit_MulC0_16s (IppsDCTLifterState_16s*
   pDCTLifter, int lenDCT, const Ipp32s* pLifterQ15, int lenCeps);
```

Description

- DCT Initialization – This function initializes the structure for the DCT calculation and liftering. C0 is stored as the last element of the output vector.

Input Arguments

- `lenDCT` – length of the DCT ($0 <$ `lenDCT` ≤ 8192)
- `lenCeps` – number of the output coefficients not including C0 ($0 <$ `lenCeps` $\leq$ `lenDCT`)
- `pLifterQ15` – pointer to the Q15 format liftering coefficients vector ($0.0_{Q15} <$ `pLifterQ15[k]` $< 512.0_{Q15}$)

Output Arguments

`pDCTLifter` – pointer to the structure to be initialized for the DCT calculation and liftering. State memory address stored in `pDCTLifter` must be aligned to 32-bit word boundary.

Returns

- `ippStsNoErr` – no error
- `ippStsSizeErr` – indicates an error if `pLifterQ15[k]`, `lenDCT`, or `lenCeps` are out of range
- `ippStsNullPtrErr` – indicates an error when `pLifterQ15` or `pDCTLifter` pointer is NULL

ippsEvalFBank_32s_Sfs

Prototype

```
IppStatus ippsEvalFBank_32s_Sfs (const Ipp32s* pSrc,
   Ipp32s* pDst, const IppsFBankState_32s* pFBank, int scaleFactor);
```

Description

Mel-frequency filter bank analysis – This function performs the filter bank analysis for the input vector pSrc. See description of the analysis under "ippsMelFBankInit_32s".

Input Arguments

- `pSrc` – pointer to the non-negative magnitude spectrum coefficients (`pSrc[k]` ≥ 0)
- `pFBank` – pointer to the filter bank structure
- `scaleFactor` – saturation fixed scale factor (scaled primitives only) (-32 < `scaleFactor` < 32)

Output Arguments

`pDst` – pointer to the output vector

Returns

- `ippStsNoErr` – no error
- `ippStsSizeErr` – indicates an error if `pSrc[i]` < 0 or `scaleFactor` are out of range
- `ippStsNullPtrErr` – indicates an error when the `pSrc` or `pDst` or `pFBank` pointer is NULL
- `ippStsFBankErr` – indicates an error when `pFBank` structure is not ready for calculation

MelFBankGetSize_32s

Prototype

```
IppStatus ippsMelFBankGetSize_32s (int winSize, int nFilter,
   IppMelMode mode, int *pSize);
```

Description

Mel-frequency filter bank structure size – This function determines the size required for the Mel-frequency filter bank structure and associated storage. Call this function before memory allocation and before `ippsMelFBankInit_32s`.

Input Arguments

- `winSize` – frame length in samples ($32 \le$ `winSize` ≤ 8192)
- `nFilter` – number of Mel-scale filter banks K ($0 <$ `nFilter` $\le$ `winSize`)
- `mode` – flag that determines the execution mode. Currently only IPP_FBANK_FREQWGT is supported

Output Arguments

`pSize` – pointer to the variable to contain the size of the filter bank structure

Returns

- `ippStsNoErr` – 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

ippsMelFBankInit_32s

Prototype

```
IppStatus ippsMelFBankInit_32s (IppsFBankState_32s* pFBank,
   int *pFFTLen, int winSize, Ipp32s sampFreq, Ipp32s lowFreq, Ipp32s
   highFreq, int nFilter, Ipp32s melMulQ15, Ipp32s melDivQ15, IppMelMode
   mode);
```

Description

Mel-frequency filter bank initialization – 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 cepstral coefficients (MFCC) feature calculation. The Mel-frequency translation is accomplished according to the following equation:

$$mel(f) = melMul * \ln(1 + f / melDiv)$$

The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

$$c_k = mel(f_{low}) + k * [mel(f_{high}) - mel(f_{low})] / (nFilter + 1), \quad k = 0, ..., nFilter+1$$

$$y_k = \text{round}[2^{pFFTOrder[0]} * mel^{-1}(c_k) / sampFreq], \quad k = 0, ..., nFilter+1$$

where round(x) rounds the value of x to the nearest integer. Calculate the filter outputs using the `ippsEvalFBank` function according to the following formula:

$$fBank(k) = \sum_{f=y_k}^{y_{k+1}} \frac{f - y_k + 1}{y_{k+1} - y_k + 1} X(f) + \sum_{f=y_{k+1}+1}^{y_{k+2}} \frac{y_{k+2} - f + 1}{y_{k+2} - y_{k+1} + 1} X(f), \quad k = 0,...,nFilter - 1$$

where X(f) is the filter bank input (for example, magnitude of the signal DFT).

Input Arguments

- `winSize` – frame length in samples ($32 \le$ `winSize` ≤ 8192)
- `sampFreq` – input signal sampling frequency Fs in Hz ($0 <$ `sampFreq` ≤ 48000)
- `lowFreq` – start frequency f_{low} of the first band-pass filter in Hz ($0 \le$ `lowFreq` $<$ `highFreq`)
- `highFreq` – end frequency f_{high} of the last band-pass filter in Hz (`lowFreq` $<$ `highFreq` $\le$ `sampFreq`/2)
- `nFilter` – number of Mel-scale filter banks K ($0 <$ `nFilter` $\le 2^{\lceil \log_2 winSize \rceil - 1} + 1$)

- melMulQ15 – Q15 Mel-scale formula multiply factor (16.0 $_{Q15}$ < melMulQ15 < 32768.0 $_{Q15}$). Use 2595 for standard Mel-warping.
- melDivQ15 – Q15 Mel-scale formula divisor (16.0$_{Q15}$ < melDivQ15 < 8192.0 $_{Q15}$).
- mode – Flag that determines the execution mode. Currently only IPP_FBANK_FREQWGT is supported.

Output Arguments

- pFBank – pointer to the Mel-scale filter bank structure to be initialized. State memory address stored in pFBank must be aligned to 32-bit word boundary.
- pFFTLen – FFT length ($2^{\lceil \log_2 winSize \rceil}$)

Returns

- ippStsNoErr – no error
- ippStsBadArgErr – indicates an error when melDivQ15 & melMulQ15 are out of bounds
- ippStsFBankFlagErr – indicates an error when the mode value is incorrect
- ippStsNullPtrErr – indicates an error when pFBank or pFFTLen pointers are NULL
- ippStsSizeErr – indicates an error when winSize, nFilter, sampFreq, lowFreq, or highFreq is less than or equal to 0; or when melDivQ15 or melMulQ15 are out of range
- ippStsFBankFreqErr – indicates an error when highFreq is less than lowFreq or greater than sampFreq/2

ippsSignChangeRate_16s

Prototype

```
IppStatus ippsSignChangeRate_16s (const Ipp16s* pSrc,
   int len, Ipp32s* pDstResult);
```

Description

This function counts the number of sign changes in the input signal. A sign change is considered to have occurred whenever pSrc[i] and pSrc[i+1] are not equal to 0 and differ in sign.

Input Arguments

- pSrc – pointer to input signal

- `len` – number of elements contained in the input and output vectors (0 < `len` < 65536)

Output Arguments

`pDstResult` – pointer to the result variable

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – indicates an error when the pointers to data arrays are NULL
- `ippStsSizeErr` – indicates an error when len is less or equal to 0

Speech Recognition Feature Compression Primitives

The Intel® IPP Speech Recognition Feature Compression primitives are described in the following sections.

CdbkGetSize_16s

Prototype

```
IppStatus ippsCdbkGetSize_16s (int width, int step, int height,
   int cdbkSize, IppCdbk_Hint hint, int *pSize);
```

Description

Vector quantization codebook size calculation – This function calculates the size in bytes of the codebook and additional information to be used for fast search.

Input Arguments

- `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. The only hint value that is currently supported on the Intel XScale® microarchitecture is:

— 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. The height parameter is not needed in this mode and its value will be ignored.

Output Arguments

`pSize` – pointer to the variable to contain the size in bytes of the codebook structure and associated storage

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – indicates an error when `pSize` pointer is NULL
- `ippStsSizeErr` – indicates an error when `cdbkSize` is less than or equal to 0; `cdbkSize` is greater than 8192; `cdbkSize` is not equal to `height`; `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 on the Intel XScale® microarchitecture is IPP_CDBK_FULL.The height parameter is not needed in this mode and its value will be ignored.

CdbkInit_L2_16s

Prototype

```
IppStatus ippsCdbkInit_L2_16s (IppsCdbkState_16s* pCdbk,
   const Ipp16s* pSrc, int width, int step, int height, int cdbkSize,
   Ipp_Cdbk_Hint hint);
```

Description

Vector quantization codebook initialization – 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.

Input Arguments

- `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 logarithmic 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 logarithmically search.

Output Arguments

`pCdbk` – pointer to the codebook structure to be initialized. State memory address stored in `pCdbk` must be aligned to 32-bit word boundary.

Returns

- `ippStsNoErr` – 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 on the Intel XScale® microarchitecture is IPP_CDBK_FULL.

NOTE. State memory address stored in pCdbk must be aligned to 32-bit word boundary.

SplitVQ_16s16s

Prototype

```
IppStatus ippsSplitVQ_16s16s (const Ipp16s* pSrc, int srcStep,
   Ipp16s* pDst, int dstStep, int height, const IppsCdbkState_16s**
   pCdbk, int nStream)
```

Description

Split vector quantization – This function quantizes the multiple-stream vectors `pSrc` against given codebooks `pCdbks`. The length of each stream is assumed to equal to that of the corresponding codebook vectors.

Input Arguments

- `pCdbk` – pointer to the codebook structure containing nStream streams
- `pSrc` – pointer to the source vector containing height*`srcStep` elements
- `srcStep` – row step in the source vector pSrc (0 < `srcStep` < 512.
- `dstStep` – row step in the destination vector `pDst` (0 < `dstStep` < 512.
- `height` – number of rows in the source and destination vectors (0 < `height` < 65536)
- `nStream` – number of streams in the source vectors (0 < `nStream` < 512)

Output Arguments

`pDst` – pointer to the destination indexing vector containing height*dstStep entries

Returns

- `ippStsNoErr` – no error
- `ippStsNullPtrErr` – indicates an error when `pCdbk`, `pCdbk[k]`, `pSrc`, or `pDst` pointers are 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 the number of bits sufficient to represent the indexes is greater than `dstStep`

FormVectorVQ_16s16s

Prototype

```
IppStatus ippsFormVectorVQ_16s16s (const Ipp16s* pSrc, int srcStep,
   Ipp16s* pDst, int dstStep, int height, const IppsCdbkState_16s**
   pCdbk, int nStream)
```

Description

Split vector dequantization – 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.

Input Arguments

- `pCdbk` – pointer to the codebook structure containing `nStream` streams
- `pSrc` – pointer to the source vector containing `height*srcStep` elements
- `srcStep` – row step in the source vector `pSrc` (0 < `srcStep` < 512)
- `dstStep` – row step in the destination vector pDst (0 < `dstStep` < 512)
- `height` – number of rows in the source and destination vectors (0 < `height` < 65536)
- `nStream` – number of streams in the source vectors (0 < `nStream` < 512)

Output Arguments

`pDst` – pointer to the constructed vectors containing `height*dstStep` entries

Returns

- `ippStsNoErr` – no error

- `ippStsNullPtrErr` – indicates an error when `pCdbk`, `pCdbk[k]`, `pSrc`, or `pDst` pointers are 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 the number of bits sufficient to represent the indexes is greater than `srcStep`

H.264 Video Decoder 21

This chapter describes the Intel® Integrated Performance Primitives that support the ITU-T Rec. H.264 / ISO/IEC 14496-10 AVC video decoder. H.264 is a popular industry standard for compressed video presentation with substantially increased coding efficiency and enhanced robustness to network environments in various applications such as video conferencing, digital storage media, television broadcasting, internet streaming, and communication. To benefit application developers, the design philosophy of the primitives emphasizes maximum flexibility and performance. On one hand, developers have the option of building a complete H.264 decoder solution using the compact set of performance optimized H.264 primitives described in this chapter in conjunction with the user's administrative and memory management functions customized for the application environment.The H.264 primitives have been tuned carefully for minimum cycle count, minimum memory footprint, and maximum quality. On the other hand, developers have the option of building a custom H.264 decoder using only a subset of the Intel® IPP H.264 primitives. This development option is facilitated in the API by providing access to the intermediate computational results generated by each of the H.264 routines. Finally, the API allows the user to fully exploit performance properties of a particular target operating system (OS) by allowing user management of administrative functions such as the high-level bit stream manipulation, memory allocation/deallocation, and control of the various buffers.

The primitives cover the following aspects of H.264 decoder:

- Quarter-sample-accurate motion compensation
- Directional spatial prediction for intra coding
- CAVLC entropy decoding
- Hierarchical 4x4 block integer transform and dequantisation
- In-loop deblock filtering

Table 21-1 Video Coding

API Name	Description
ippiInterpolateLuma_H264_8u_C1R	H.264 video decoder 1/4 pixel luma interpolation
ippiInterpolateChroma_H264_8u_C1R	H.264 video decoder 1/8 pixel chroma interpolation
ippiPredictIntra_4x4_H264_8u_C1R	H.264 video decoder luma 4x4 intra prediction
ippiPredictIntra_16x16_H264_8u_C1R	H.264 video decoder luma16x16 intra prediction
ippiPredictIntraChroma8x8_H264_8u_C1R	H.264 video decoder chroma 8x8 intra prediction
ippiDecodeCoeffsToPairCAVLC_H264_1u8u	H.264 video decoder CAVLC decoding
ippiDecodeChromaDcCoeffsToPairCAVLC_H264_1u8u	H.264 video decoder chroma DC CAVLC decoding
ippiTransformDequantLumaDCFromPair_H264_8u16s_C1	H.264 video decoder lumaDC transform/dequantization
ippiTransformDequantChromaDCFromPair_H264_8u16s_C1	H.264 video decoder chromaDC transform/ dequantization
ippiDequantTransformResidualFromPairAndAdd_H264_8u_C1	H.264 video decoder residual transform/dequantization and add to prediction
ippiFilterDeblockingLuma_VerEdge_H264_8u_C1IR	H.264 video decoder luma MB vertical edge deblock filtering
ippiFilterDeblockingLuma_HorEdge_H264_8u_C1IR	H.264 video decoder luma MB horizontal edge deblock filtering
ippiFilterDeblockingChroma_VerEdge_H264_8u_C1IR	H.264 video decoder chroma MB vertical edge deblock filtering
ippiFilterDeblockingChroma_HorEdge_H264_8u_C1IR	H.264 video decoder chroma MB horizontal edge deblock filtering

Data Types and Structures

Intra_16x16 Macroblock Prediction Mode

Intra_16x16 macroblock prediction modes are defined as follows:

```
typedef enum
{
   IPP_16X16_VERT   = 0,/* Intra_16x16_Vertical (prediction mode) */
   IPP_16X16_HOR    = 1,/* Intra_16x16_Horizontal (prediction mode) */
   IPP_16X16_DC     = 2,/* Intra_16x16_DC (prediction mode) */
   IPP_16X16_PLANE  = 3,/* Intra_16x16_Plane (prediction mode) */
} IppIntra16x16PredMode_H264;
```

Intra_4x4 Macroblock Prediction Mode

Intra_4x4 macroblock prediction modes are defined as follows:

```
typedef enum
{
    IPP_4x4_VERT     = 0,  /* Intra_4x4_Vertical (prediction mode) */
    IPP_4x4_HOR      = 1,/* Intra_4x4_Horizontal (prediction mode) */
    IPP_4x4_DC       = 2,/* Intra_4x4_DC (prediction mode) */
    IPP_4x4_DIAG_DL  = 3, /* Intra_4x4_Diagonal_Down_Left (prediction
   mode) */
    IPP_4x4_DIAG_DR  = 4,/* Intra_4x4_Diagonal_Down_Right (prediction
   mode) */
    IPP_4x4_VR       = 5,/* Intra_4x4_Vertical_Right (prediction mode)
   */
```

```
    IPP_4x4_HD       = 6,/* Intra_4x4_Horizontal_Down (prediction mode)
   */
    IPP_4x4_VL       = 7,/* Intra_4x4_Vertical_Left (prediction mode) */
    IPP_4x4_HU       = 8/* Intra_4x4_Horizontal_Up (prediction mode) */
} IppIntra4x4PredMode_H264;
```

Intra Chroma Macroblock Prediction Mode

Intra chroma macroblock prediction modes are defined as follows:

```
typedef enum
{
  IPP_CHROMA_DC    = 0,/* Intra_Chroma_DC (prediction mode) */
  IPP_CHROMA_HOR   = 1,/* Intra_Chroma_Horizontal (prediction mode) */
  IPP_CHROMA_VERT  = 2,/* Intra_Chroma_Vertical (prediction mode) */
  IPP_CHROMA_PLANE = 3/* Intra_Chroma_Plane (prediction mode) */
} IppIntraChromaPredMode_H264;
```

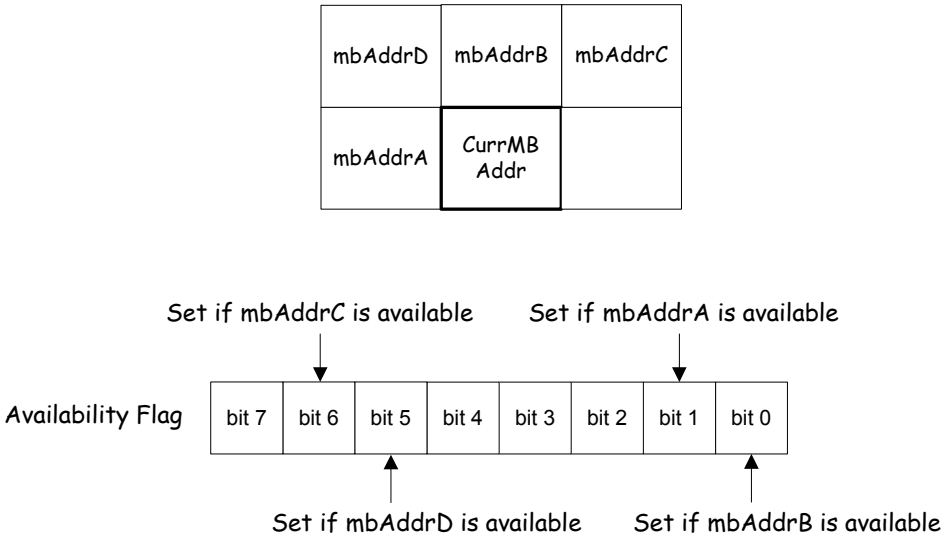
Neighboring Macroblock Availability

Neighboring macroblock availability flags are defined as follows:

```
enum {
     IPP_UPPER        = 1,/* set if the above macroblock is available */
     IPP_LEFT         = 2,/* set if the left macroblock is available */
     IPP_CENTER       = 4,
     IPP_RIGHT        = 8,
     IPP_LOWER        = 16,
     IPP_UPPER_LEFT   = 32,/* set if the above-left macroblock is
   available */
     IPP_UPPER_RIGHT  = 64,/* set if the above-right macroblock is
   available */
```

```
        IPP_LOWER_LEFT    = 128,
        IPP_LOWER_RIGHT   = 256
};
```

Figure 21-1 Neighboring Macroblock Availability Definition

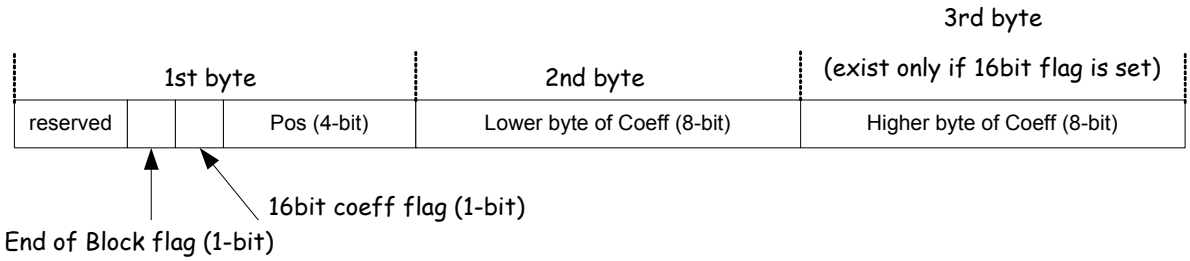


Coefficient-Position Pair Buffer

The interface between CAVLC decoding output and Transform/Dequantisation input is formed as an efficient and compact buffer storage structure called Coefficient-Position Pair Buffer. It stores all non-zero coefficients in 4x4 block units (or 2x2 block units for chroma DC), along with their position in the block. This type of storage format is designed for both optimal performance and minimal storage on the Intel XScale® microarchitecture. The Coefficient-Position Pair Buffer Definition is shown in Figure 21-2.

This figure shows how each non-zero coefficient occupies two to three bytes in the pair buffer.

Figure 21-2 Coefficient-Position Pair Buffer Definition



IPP API Details

In the Return section of all of the functions described in this chapter, the return code `IPP_STATUS_BAD_ARG` only applies to the debug version of the code. Since parameter checking decreases performance, `IPP_STATUS_BAD_ARG` is not implemented in the release version of the code. As a result, in the release version of the code, preprimitive behavior is unpredictable when an invalid parameter is encountered.

Intra Prediction

ippiPredictIntra_4x4_H264_8u_C1R

Prototype

```
IppStatus ippiPredictIntra_4x4_H264_8u_C1R (Ipp8u* pSrcLeft, Ipp8u
   *pSrcAbove, Ipp8u *pSrcAboveLeft, Ipp8u* pDst, int leftStep, int
   dstStep, IppIntra4x4PredMode_H264 predMode, Ipp32s availability);
```

Description

Perform Intra_4x4 prediction for luma samples.

Input Arguments

- pSrcLeft — Pointer to the buffer of 4 left coefficients: p[x, y] (x = -1, y = 0..3)

- pSrcAbove — Pointer to the buffer of 8 above coefficients: p[x,y] (x = 0..7, y = -1)
- pSrcAboveLeft — Pointer to the above left coefficient: p[x,y] (x = -1, y = -1)
- leftStep — Step of left coefficient buffer
- dstStep — Step of the destination buffer
- predMode — Intra_4x4 prediction mode, please refer to section 3.4.2.
- availability — Neighboring 4x4 block availability flag, please refer to "Neighboring Macroblock Availability".

Output Arguments

- pDst — Pointer to the destination buffer

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- pDst is NULL.
- dstStep < 4.
- predMode is not in the valid range of enumeration IppIntra4x4PredMode_H264.
- predMode is IPP_4x4_VERT, but availability doesn't set IPP_UPPER indicating p[x,-1] (x = 0..3) is not available.
- predMode is IPP_4x4_HOR, but availability doesn't set IPP_LEFT indicating p[-1,y] (y = 0..3) is not available.
- predMode is IPP_4x4_DIAG_DL, but availability doesn't set IPP_UPPER indicating p[x,-1] (x = 0..3) is not available.
- predMode is IPP_4x4_DIAG_DR, but availability doesn't set IPP_UPPER_LEFT or IPP_UPPER or IPP_LEFT indicating p[x,-1] (x = 0..3), or p[-1,y] (y = 0..3) or p[-1,-1] is not available.
- predMode is IPP_4x4_VR, but availability doesn't set IPP_UPPER_LEFT or IPP_UPPER or IPP_LEFT indicating p[x,-1] (x = 0..3), or p[-1,y] (y = 0..3) or p[-1,-1] is not available.
- predMode is IPP_4x4_HD, but availability doesn't set IPP_UPPER_LEFT or IPP_UPPER or IPP_LEFT indicating p[x,-1] (x = 0..3), or p[-1,y] (y = 0..3) or p[-1,-1] is not available.
- predMode is IPP_4x4_VL, but availability doesn't set IPP_UPPER indicating p[x,-1] (x = 0..3) is not available.
- predMode is IPP_4x4_HU, but availability doesn't set IPP_LEFT indicating p[-1,y] (y = 0..3) is not available.
- availability sets IPP_UPPER, but pSrcAbove is NULL.
- availability sets IPP_LEFT, but pSrcLeft is NULL.

- availability sets IPP_UPPER_LEFT, but pSrcAboveLeft is NULL.

NOTE. *pSrcAbove, pSrcAbove, pSrcAboveLeft may be invalid pointers if they are not used by intra prediction implied in predMode.*

ippiPredictIntra_16x16_H264_8u_C1R

Prototype

```
IppStatus ippiPredictIntra_16x16_H264_8u_C1R (Ipp8u* pSrcLeft, Ipp8u
   *pSrcAbove, Ipp8u *pSrcAboveLeft, Ipp8u* pDst, int leftStep, int
   dstStep, IppIntra16x16PredMode_H264 predMode, Ipp32s availability);
```

Description

Perform Intra_16x16 prediction for luma samples.

Input Arguments

- pSrcLeft — Pointer to the buffer of 16 left coefficients: p[x, y] (x = -1, y = 0..15)
- pSrcAbove — Pointer to the buffer of 16 above coefficients: p[x,y] (x = 0..15, y = -1)
- pSrcAboveLeft — Pointer to the above left coefficient: p[x,y] (x = -1, y = -1)
- leftStep — Step of left coefficient buffer
- dstStep — Step of the destination buffer
- predMode — Intra_16x16 prediction mode, please refer to section 3.4.1.
- avilability — Neighboring 16x16 MB availability flag, please refer to section 3.4.4.

Output Arguments

- pDst — Pointer to the destination buffer

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- pDst is NULL.
- dstStep < 16.
- predMode is not in the valid range of enumeration IppIntra16x16PredMode_H264
- predMode is IPP_16X16_VERT, but availability doesn't set IPP_UPPER indicating p[x,-1] (x = 0..15) is not available.
- predMode is IPP_16X16_HOR, but availability doesn't set IPP_LEFT indicating p[-1,y] (y = 0..15) is not available.
- predMode is IPP_16X16_PLANE, but availability doesn't set IPP_UPPER_LEFT or IPP_UPPER or IPP_LEFT indicating p[x,-1](x = 0..15), or p[-1,y] (y = 0..15), or p[-1,-1] is not available.
- availability sets IPP_UPPER, but pSrcAbove is NULL.
- availability sets IPP_LEFT, but pSrcLeft is NULL.
- availability sets IPP_UPPER_LEFT, but pSrcAboveLeft is NULL.

NOTE. *pSrcAbove, pSrcAbove, pSrcAboveLeft may be invalid pointer if they are not used by intra prediction implied in predMode.*

NOTE. *IPP_UPPER_RIGHT is not used in intra_16x16 luma prediction.*

ippiPredictIntraChroma8x8_H264_8u_C1R

Prototype

```
IppStatus ippiPredictIntraChroma8x8_H264_8u_C1R (Ipp8u* pSrcLeft, Ipp8u
   *pSrcAbove, Ipp8u *pSrcAboveLeft, Ipp8u* pDst, int leftStep, int
   dstStep, IppIntraChromaPredMode_H264 predMode, Ipp32s availability);
```

Description

Perform Intra prediction for chroma samples.

Input Arguments

- pSrcLeft — Pointer to the buffer of 8 left coefficients: p[x, y] (x = -1, y = 0..7)
- pSrcAbove — Pointer to the buffer of 8 above coefficients: p[x,y] (x = 0..7, y = -1)
- pSrcAboveLeft — Pointer to the above left coefficient: p[x,y] (x = -1, y = -1)
- leftStep — Step of left coefficient buffer
- dstStep — Step of the destination buffer
- predMode — Intra chroma prediction mode, please refer to section 3.4.3.
- avilability — Neighboring chroma block availability flag, please refer to "Neighboring Macroblock Availability".

Output Arguments

- pDst — Pointer to the destination buffer

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- pDst is NULL.
- dstStep < 8.
- predMode is not in the valid range of enumeration IppIntraChromaPredMode_H264.
- predMode is IPP_CHROMA_VERT, but availability doesn't set IPP_UPPER indicating p[x,-1] (x = 0..7) is not available.
- predMode is IPP_CHROMA_HOR, but availability doesn't set IPP_LEFT indicating p[-1,y] (y = 0..7) is not available.
- predMode is IPP_CHROMA_PLANE, but availability doesn't set IPP_UPPER_LEFT or IPP_UPPER or IPP_LEFT indicating p[x,-1](x = 0..7), or p[-1,y] (y = 0..7), or p[-1,-1] is not available.
- availability sets IPP_UPPER, but pSrcAbove is NULL.
- availability sets IPP_LEFT, but pSrcLeft is NULL.

- availability sets IPP_UPPER_LEFT, but pSrcAboveLeft is NULL.

NOTE. *pSrcAbove, pSrcAbove, pSrcAboveLeft may be invalid pointer if they are not used by intra prediction implied in predMode.*

NOTE. *IPP_UPPER_RIGHT is not used in intra chroma prediction.*

Interpolation

ippiInterpolateLuma_H264_8u_C1R

Prototype

```
IPPAPI(IppStatus, ippiInterpolateLuma_H264_8u_C1R, (const Ipp8u* pSrc,
   Ipp32s srcStep, Ipp8u* pDst, Ipp32s dstStep, Ipp32s dx, Ipp32s dy,
   IppiSize roi));
```

Description

Performs quarter-pixel interpolation for inter luma MB.

Input Arguments

- pSrc — Pointer to the source reference frame buffer
- srcStep — Reference frame step in byte
- dstStep — Destination frame step in byte
- dx — Fractional part of horizontal motion vector component in 1/4 pixel unit
- dy — Fractional part of vertical motion vector y component in 1/4 pixel unit
- roi — Dimension of the interpolation region

Output Arguments

- pDst — Pointer to the destination frame buffer

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- pSrc or pDst is NULL.
- srcStep or dstStep < roi.width.
- dx or dy is out of range [0-3].
- roi.width or roi.height is out of range {4, 8, 16}.
- roi.width is equal to 4, but pDst is not 4 byte aligned.
- roi.width is equal to 8 or 16, but pDst is not 8 byte aligned.
- srcStep or dstStep is not a multiple of 8.

ippiInterpolateChroma_H264_8u_C1R

Prototype

```
IPPAPI(IppStatus, ippiInterpolateChroma_H264_8u_C1R, (const Ipp8u* pSrc,
   Ipp32s srcStep, Ipp8u* pDst, Ipp32s dstStep, Ipp32s dx, Ipp32s dy,
   IppiSize roi));
```

Description

Performs 1/8-pixel interpolation for inter chroma MB.

Input Arguments

- pSrc — Pointer to the source reference frame buffer
- srcStep — Reference frame step in byte
- dstStep — Destination frame step in byte
- dx — Fractional part of horizontal motion vector component in 1/8 pixel unit
- dy — Fractional part of vertical motion vector component in 1/8 pixel unit

- roi Dimension of the interpolation region

Output Arguments

- pDst Pointer to the destination frame buffer

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- pSrc or pDst is NULL.
- srcStep or dstStep < 8.
- dx or dy is out of range [0-7].
- roi.width or roi.height is out of range {2,4,8}.
- roi.width is equal to 2, but pDst is not 2 byte aligned.
- roi.width is equal to 4, but pDst is not 4 byte aligned.
- roi.width is equal to 8, but pDst is not 8 byte aligned.
- srcStep or dstStep is not a multiple of 8.

CAVLC Decoding

ippiDecodeChromaDcCoeffsToPairCAVLC_H264_1u8u

Prototype

```
IppStatus ippiDecodeChromaDcCoeffsToPairCAVLC_H264_1u8u(Ipp8u**
   ppBitStream, Ipp32s* pOffset, Ipp8u* pNumCoeff, Ipp8u**
   ppPosCoefbuf);
```

Description

Performs CAVLC decoding for 2x2 block of ChromaDCLevel.

Input Arguments

- ppBitStream Double pointer to current byte in bit stream buffer

- pOffset *ppBitStream — Pointer to current bit position in the byte pointed to by

Output Arguments

- ppBitStream — *ppBitStream is updated after each block is decoded
- pOffset — *pOffset is updated after each block is decoded
- pNumCoeff — Pointer to the number of nonzero coefficients in this block
- ppPosCoefBuf — Double pointer to destination residual coefficient-position pair buffer

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- ppBitStream or pOffset is NULL.
- ppPosCoefBuf or pNumCoeff is NULL.

NOTE. **pOffset is valid within [0-7].*
*Bit Position in one byte: *pOffset*
|Most Least|
| 1 2 3 4 5 6 7|
ppPosCoefBuf will not be updated if there are no non-zero coefficients in currently decoded block. In this case, users are expected to keep this information in order to correctly bypass the transform/dequantisation of these empty blocks later.

ippiDecodeCoeffsToPairCAVLC_H264_1u8u

Prototype

```
IppStatus ippiDecodeCoeffsToPairCAVLC_H264_1u8u(Ipp8u** ppBitStream,
   Ipp32s* pOffset, Ipp8u* pNumCoeff, Ipp8u**ppPosCoefbuf, int
   sVLCSelect, int sMaxNumCoeff);
```

Description

Performs CAVLC decoding for 4x4 block of Intra16x16DCLevel, Intra16x16ACLevel, LumaLevel, and ChromaACLevel.

Input arguments

- ppBitstream — Double pointer to current byte in bit stream buffer
- pOffset — Pointer to current bit position in the byte pointed to by *ppBitStream
- sMaxNumCoeff — Maximum number of non-zero coefficients in current block
- sVLCSelect — VLC table selector, obtained from number of non-zero AC coefficients of above and left 4x4 blocks

Output Arguments

- ppBitStream — *ppBitStream is updated after each block is decoded
- pOffset — *pOffset is updated after each block is decoded
- pNumCoeff — Pointer to the number of nonzero coefficients in this block
- ppPosCoefBuf — Double pointer to destination residual coefficient-position pair buffer

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- ppBitStream or pOffset is NULL.
- ppPosCoefBuf or pNumCoeff is NULL.
- sMaxNumCoeff is neither equal to 16 nor equal to 15.
- sVLCSelect is less than 0.

Dequantization/Transform/Add Residual

ippiTransformDequantLumaDCFromPair_H264_8u16s_C1

Prototype

```
IppStatus ippiTransformDequantLumaDCFromPair_H264_8u16s_C1(Ipp8u **
   ppSrc, Ipp16s* pDst, int QP);
```

Description

Reconstruct the 4x4 LumaDC block from coefficient-position pair buffer, perform integer inverse, and dequantization for 4x4 LumaDC coefficients, and update the pair buffer pointer to next non-empty block.

Input Arguments

- ppSrc — Double pointer to residual coefficient-position pair buffer output by CALVC decoding
- QP — Quantization parameter

Output Arguments

- ppSrc — *ppSrc is updated to the start of next non empty block
- pDst — Pointer to the reconstructed 4x4 LumaDC coefficients buffer

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- ppSrc or pDst is NULL.
- pDst is not 8 byte aligned.
- QP is not in the range of [1-51].

ippiTransformDequantChromaDCFromPair_H264_8u16s_C1

Prototype

```
IppStatus ippiTransformDequantChromaDCFromPair_H264_8u16s_C1(Ipp8u **
   ppSrc, Ipp16s* pDst, int QP);
```

Description

Reconstruct the 2x2 ChromaDC block from coefficient-position pair buffer, perform integer inverse transformation, and dequantization for 2x2 chroma DC coefficients, and update the pair buffer pointer to next non-empty block.

Input arguments

- ppSrc — Double pointer to residual coefficient-position pair buffer output by CALVC decoding
- QP — Quantization parameter

Output Arguments

- ppSrc — *ppSrc is updated to the start of next non empty block
- pDst buffer — Pointer to the reconstructed 2x2 ChromaDC coefficients

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- ppSrc or pDst is NULL.
- pDst is not 8 byte aligned.
- QP is not in the range of [1-51].

ippiDequantTransformResidualFromPairAndAdd_H264_8u_C1

Prototype

```
IppStatus ippiDequantTransformResidualFromPairAndAdd_H264_8u_C1(Ipp8u
   **ppSrc, const Ipp8u* pPred, Ipp16s* pDC, Ipp8u* pDst, int predStep,
   int dstStep, int QP, int AC);
```

Description

Reconstruct the 4x4 residual block from coefficient-position pair buffer, perform dequantisation and integer inverse transformation for 4x4 block of residuals with previous intra prediction or motion compensation data, and update the pair buffer pointer to next non-empty block.

Input Arguments

- ppSrc — Double pointer to residual coefficient-position pair buffer output by CALVC decoding
- pPred — Pointer to the reference 4x4 block
- predStep — Reference frame step in byte
- dstStep — Destination frame step in byte
- pDC — Pointer to the DC coefficient of this block, NULL if it doesn't exist
- QP — Quantization parameter
- AC — Flag indicating if at least one non-zero coefficient exists

Output Arguments

- pDst — pointer to the reconstructed 4x4 block data

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- pPred or pDst is NULL.
- pPred or pDst is not 4 byte aligned.
- predStep or dstStep is not a multiple of 4.

- AC is not zero, but Qp is not in the range of [1-51] or ppSrc is not NULL.
- AC is zero, but pDC is NULL.

Deblocking Filter

ippiFilterDeblockingLuma_VerEdge_H264_8u_C1IR

Prototype

```
IppStatus ippiFilterDeblockingLuma_VerEdge_H264_8u_C1IR, (Ipp8u*
   pSrcDst, Ipp32s srcdstStep, Ipp8u* pAlpha, Ipp8u* pBeta, Ipp8u*
   pThresholds, Ipp8u *pBS);
```

Description

Performs deblocking filtering on four vertical edges of the luma macroblock(16x16).

Input Arguments

- pSrcDst — Pointer to the initial and resultant coefficients
- srcdstStep — Step of the arrays
- pAlpha — Array of size 2 of Alpha Thresholds (the first item is alpha threshold for external vertical edge, and the second item is for internal vertical edge)
- pBeta — Array of size 2 of Beta Thresholds (the first item is alpha threshold for external vertical edge, and the second item is for internal vertical edge)
- pTresholds — Array of size 16 of Thresholds (TC0) (values for the left edge of each 4x4 block, arranged in vertical block order)
- pBS — Array of size 16 of BS parameters (arranged in vertical block order)

Output Arguments

- pSrcDst — Pointer to the initial and resultant coefficients

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- Either of the pointers in pSrcDst, pAlpha, pBeta, pTresholds or pBS is NULL.
- pSrcDst is not 8-byte aligned.
- srcdstStep is not a multiple of 8.

ippiFilterDeblockingLuma_HorEdge_H264_8u_C1IR

Prototype

```
IppStatus ippiFilterDeblockingLuma_HorEdge_H264_8u_C1IR, (Ipp8u*
   pSrcDst, Ipp32s srcdstStep, Ipp8u* pAlpha, Ipp8u* pBeta, Ipp8u*
   pThresholds, Ipp8u *pBS);
```

Description

Performs deblocking filtering on four horizontal edges of the luma macroblock (16x16).

Input Arguments

- pSrcDst — Pointer to the initial and resultant coefficients
- srcdstStep — Step of the arrays
- pAlpha — Array of size 2 of Alpha Thresholds (the first item is alpha threshold for external vertical edge, and the second item is for internal horizontal edge)
- pBeta — Array of size 2 of Beta Thresholds (the first item is alpha threshold for external horizontal edge, and the second item is for internal horizontal edge)
- pTresholds — Array of size 16 of Thresholds (TC0) (values for the left edge of each 4x4 block, arranged in horizontal block order)
- pBS — Array of size 16 of BS parameters (arranged in horizontal block order)

Output Arguments

- pSrcDst — Pointer to the initial and resultant coefficients

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- Either of the pointers in pSrcDst, pAlpha, pBeta, pTresholds, or pBS is NULL.
- pSrcDst is not 8-byte aligned.
- srcdstStep is not a multiple of 8.

ippiFilterDeblockingChroma_VerEdge_H264_8u_C1IR

Prototype

```
IppStatus ippiFilterDeblockingChroma_VerEdge_H264_8u_C1IR, (Ipp8u*
   pSrcDst, Ipp32s srcdstStep, Ipp8u* pAlpha, Ipp8u* pBeta, Ipp8u*
   pThresholds, Ipp8u *pBS);
```

Description

Performs deblocking filtering on four vertical edges of the chroma macroblock(8x8).

Input Arguments

- pSrcDst — Pointer to the initial and resultant coefficients
- srcdstStep — Step of the arrays
- pAlpha — Array of size 2 of Alpha Thresholds (the first item is alpha threshold for external vertical edge, and the second item is for internal vertical edge)
- pBeta — Array of size 2 of Beta Thresholds (the first item is alpha threshold for external vertical edge, and the second item is for internal vertical edge)
- pTresholds — Array of size 8 of Thresholds (TC0) (values for the left edge of each 4x4 block, arranged in vertical block order)
- pBS — Array of size 16 of BS parameters (values for each 2x4 chroma block, arranged in vertical block order)

Output Arguments

- pSrcDst — Pointer to the initial and resultant coefficients

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- Either of the pointers in pSrcDst, pAlpha, pBeta, pTresholds, or pBS is NULL.
- pSrcDst is not 8-byte aligned.
- srcdstStep is not a multiple of 8.
- pTresholds is not 4-byte aligned.
- pBS is not 4-byte aligned.

ippiFilterDeblockingChroma_HorEdge_H264_8u_C1IR

Prototype:

```
IppStatus ippiFilterDeblockingChroma_HorEdge_H264_8u_C1IR, (Ipp8u*
   pSrcDst, Ipp32s srcdstStep, Ipp8u* pAlpha, Ipp8u* pBeta, Ipp8u*
   pThresholds, Ipp8u *pBS);
```

Description

Performs deblocking filtering on the horizontal edges of the chroma macroblock (8x8).

Input Arguments

- pSrcDst — Pointer to the initial and resultant coefficients
- srcdstStep — Step of the arrays
- pAlpha — Array of size 2 of Alpha Thresholds (the first item is alpha threshold for external horizontal edge, and the second item is for internal horizontal edge)
- pBeta — Array of size 2 of Beta Thresholds (the first item is alpha threshold for external horizontal edge, and the second item is for internal horizontal edge)
- pTresholds — Array of size 8 of Thresholds (TC0) (values for the left edge of each 4x4 block, arranged in horizontal block order)
- pBS — Array of size 16 of BS parameters (values for each 2x4 chroma block, arranged in horizontal block order)

Output Arguments

- pSrcDst Pointer to the initial and resultant coefficients

Return

If the function runs without error, it returns IPP_STATUS_OK.

If one of the following cases occurs, the function returns IPP_STATUS_BAD_ARG:

- Either of the pointers in pSrcDst, pAlpha, pBeta, pTresholds or pBS is NULL.
- pSrcDst is not 8-byte aligned.
- srcdstStep is not a multiple of 8.
- pTresholds is not 4-byte aligned.
- pBS is not 4-byte aligned.

Acronyms

1D	one-dimensional
2D	two-dimensional
AAC	Advanced Audio Coding
ACELP	Algebraic Code Excited Linear Prediction
ADIF	Audio Data Interchange Format
ADTS	Audio Data Transport Stream
AES	Advanced Encryption Standard
ANSI	American National Standards Institute
API	Application Programming Interface
ARM	Asynchronous Response Mode
BAB	Binary Alpha Block
BC	Backwards Compatibility
BFI	Bad Frame Indication
BMA	Block Match Algorithm
BSI_Abis	Bad Sub-block Indication obtained from A-bis CRC checks
CAE	Context-based arithmetic encoding
CALVC	Contex-Based Adaptive Variable Length Coding
CB	Coefficient Buffer
CBC	Cipher Block Chain
CC	Color Conversion function set. Default, it was merged with level shift
CCS	Complex Conjugate Symmetric

CCU	Channel Coding Unit
CELP	Code Excited Linear Prediction
CFB	Cipher Feed Back
CNG	Comfortable Noise Generation
CODEC	Coder/decoder
COPAD	Copy and Padding function set
CRC	Cyclic Redundancy Check
CRT	Chinese Remainder Theorem
DAA	Data Authentication Algorithm
DCT	Discrete Cosine Transform
DCTQ	DCT&Quantization function set
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
DSP	Digital Signal Processing
DSS	Digital Signature Standard
DTX	Discontinuous Transmission
DWT	Discrete Wavelet Transform
ECB	Electronic Code Book
ETSI	European Telecommunications Standards Institute
FDCT	forward DCT
FFT	fast Fourier transform
FIPS	Federal Information Processing Standard
FIR	finite impulse response
FM	Frame Muting
FDWT	Forward Discrete Wavelet Transform
GCD	Greatest Common Divisor
GOB	Group of Blocks
HMAC	Key-Hash Message Authentication Code
HUFFC	Huffman coding function set
IDCT	inverse DCT

IDCT	Inverse Discrete Cosine Transform
IEC	International Electrotechnical Commission
IIR	infinite impulse response
IMDCT	Inverse Modified Discrete Cosine Transform
IPP	Intel® Integrated Performance Primitives
IS	Intensity Stereo
ISO	International Standards Organization
ITU	International Telecommunication Union
IDW T	Inverse Discrete Wavelet Transform
LC	Low Complexity profile
LDS	Large Diamond Search
LFE	Low Frequency Enhancement
LMS	Least mean square
LP	Linear Prediction
LPC	Linear Predictive Coder
LS	Level Shift function set
LSF	Line Spectral Frequency
LSP	Line Spectral Pair
LSW	Least Significant Word
MB	Macroblock
MC	Motion Compensation
MCU	Minimum Coded Unit. The smallest group of data units that are coded
MD5	Message Digest Algorithm-5
MDCT	Modified Discrete Cosine Transform
MP	Main Profile
MP3	MPEG-1 Audio Layer 3
MPEG	Moving Pictures Experts Group
MPMLQ	Multipulse Maximum Likelihood Quantization (ITU-T G.723.1 Voice Compression)
MS	Middle/Side Stereo

MSW	Most Significant Word
MV	Motion Vector
MVFAST	Motion Vector Field Adaptive Search Technique
NIST	National Institute of Standards and Technology
OBMC	Overlapped Block Motion Compensation
OS	Operating system
PCM	Pulse-Code Modulation
PDFI	Potentially Degraded Frame Indication from RSS
PQMF	Pseudo Quadrature Mirror Filter
PSD	Power Spectral Density
PSVQ	Predictive Split Vector Quantizer
RAM	Random access memory
ROI	Rectangle Of Interest
ROM	Read only memory
RSA	A well-known public key cryptographic system
RSS	Radio Sub System
RX	Receive
SA	StrongARM*; StrongARM* platform
SAD	Sum of Absolute Differences
SCE	Single Channel Element
SDS	Small Diamond Search
SEA	Successive Elimination Algorithm
Sfs	saturated fixed scale
SID	Silence Descriptor
SFB	Scale Factor Band
SHA	Secure Hash Algorithm
SSR	Scaleable Sampling Rate profile
TDES	Triple Data Encryption Standard
TNS	Temporal Noise Shaping
TRAU	Transcoding Rate Adaptation Unit

TX	Transmit
VAD	Voice Activity Detection
VLC	Variable Length Coding
VLD	variable length decoding
VOL	Video Object Layer
VOP	Video Object Plane
SCE	Single Channel Element

Bibliography B

This bibliography provides a list of reference materials that are useful to an application programmer. Although the list is not complete or exhaustive, it serves as a starting point.

- [Mit93] – This is useful to readers who already have a basic understanding of signal processing. This reference collects the work of 27 experts in the field and has great breadth and depth.
  - — Sanjit K. Mitra and James F. Kaiser editors, *Handbook for Digital Signal Processing*, John Wiley & Sons, Inc., New York, 1993.

    These materials describe signal processing. Many of these works are referenced in the chapters of this manual.
- [Ash94] – M.R. Asharif and F. Amano, *Acoustic Echo Canceler Using the FBAF Algorithm*, IEEE Trans. Comm., Vol. 42, No. 12, Dec. 1994, pp. 3090-3094.
- [Cap78] – V. Cappellini, A. G. Constantinides, and P. Emilani. *Digital Filters and Their Applications*, Academic Press, London, 1978.
- [Cappe94] – O. Cappe, *Elimination of the musical noise phenomenon with the Ephraim and Malah noise suppressor*" IEEE Trans. Speech and Audio, Vol. 2, No. 2, Apr. 1994, pp. 345-349.
- [Coh02] – I. Cohen and B. Berdugo, *Noise Estimation by Minima Controlled Recursive Averaging for Robust Speech Enhancement*, IEEE Signal Proc. Letters, Vol. 9, No. 1, Jan. 2002, pp. 12-15.
- [Dig98] – Gives introduction discussion on wide range of compressing algorithm and industrial standards.
  - — J. Gibson, T. Berger, T. Lookabaugh, D Linbergh, and R. Baker, Digital Compression for Multimedia.
- [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.

- [ETSI] – *ETSI ES 201 108 V1.1.2 (2000-04). ETSI Standard. Speech processing, Transmission and Quality aspects (STD): Distributed Speech recognition; Front-end feature extraction algorithms; Compression algorithms*.
- [Fei92] – E. Feig and S. Winograd, Fast algorithms for DCT, *IEEE*
- [H.263] – The ITU-T standard is helpful for understanding the content of Chapter 11.
  - [H.263] ITU-T, Recommendation H.263 Version 2 (H.263+): Video Coding for Low Bitrate Communication, 27 January, 1998.
- [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.
- [IEE90] – IEEE Std 1180-1990: IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform.
- [ISO98-2] – ISO/IEC 14496-2: Information Technology – Generic Coding of Audio-Visual Objects – Part 2: Visual (FD, October 1998).
- [ISO93] and [ISO98] – The ISO standards that are called MPEG – 1 audio and MPEG-2 audio respectively. Both are helpful for understanding the content of Chapter 8.
  - [ISO93]ISO/IEC 11172-3:1993, Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5Mbit/s-----Part 3: Audio
  - [ISO98]ISO/IEC 13818-3:1998, Information technology – Generic coding of moving pictures and associated audio information-----Part 3: Audio
- [ISO98-2] – The ISO standard that are called MPEG-4 video. [They are helpful for understanding the content of Chapter 12.
- ISO/IEC 13818-7:1997, Information technology—Generic coding of moving pictures and associated audio information-----Part 7: Advanced Audio Coding (AAC)
- ISO/IEC 15444-1. Information technology – JPEG 2000 image coding system -- Part 1: Core coding system.
- [Jac89] – This document is the most concise of all the texts in this list.
  - Leland B. Jackson, *Digital Filters and Signal Processing*, Kluwer Academic Publishers, second edition,1989.
- [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.
- [Men97] – A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, Handbook of Applied Cryptography, CRC press, Inc., 1997.

- [NIC91] – Nam Ik Cho and Sang Uk Lee, Fast Algoritm and Implementation of 2D DCT, *IEEE Transactions on Circuits and Systems*, vol. 31, No.3, 1991.
- [Opp75] – Alan V. Oppenheim and Ronald W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
- [Opp89] – This is a revised edition of the classic [Opp75]
  - — Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [Riv83] R.L. Rivest, A. Shamir, and L.M. Adleman, Cryptographic Communications System and Method, U.S. Patent # 4,405,829, September, 1983.
- [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.
- [Sed78] – R. Sedgewick, *Implementing Quicksort Programs*, Communications of the ACM, Vol. 21, No. 10, pp. 847-857, Oct. 1978.
- [Tan00] S. Gokhun Tanyer and Hamzo Ozer, *Voice Activity Detection in Nonstatioinary Noise*, IEEE Trans. Speech & Audio Proc., Vol. 8, No. 4, July 2000, pp. 478-482.
- [Tau02] – David S. Taubman and Michael W. Marcellin, JPEG 2000 Image Compression Fundamentals, Standards, and Practice, Kluwer Academic Publishers 2002.
- US DOC/NIST: FIPS PUB 46-3 Data Encryption Standard (DES), October, 1999.
- US DOC/NIST: FIPS PUB 180-1 Secure Hash Standard, April, 1995.
- US DOC/NIST: FIPS PUB 186-2 Digital Signature Standard, January, 2000.
- [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] – Covers continuous-time systems.
  - — Rodger E. Ziemer, William H. Tranter, and D. Ronald Fannin, *Signals and Systems: Continuous and Discrete*, Macmillan Publishing Co., New York, 1983.
- US DOC/NIST: FIPS PUB 81 DES Modes Of Operations, December, 1980
- US DOC/NIST: FIPS PUB 197: Advanced Encryption Standard (AES), November, 2001
- J. Daemen and V. Rijmen: *The Rijndael Block Cipher*, November, 1998
- B. Schneier: *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, December, 1993
- B. Schneier etc.: *Twofish: A 128-Bit Block Cipher*, June, 1998
- US DOC/NIST: FIPS PUB 113: Computing Data Authentication, May, 1985
- US DOC/NIST: FIPS PUB 180-1 Secure Hash Standard, April, 1995

- US DOC/NIST: FIPS (draft) Secure Hash Standard, October, 2000
- RFC 1321: The MD5 Message-Digest Algorithm, April, 1992
- US DOC/NIST FIPS PUB 198: *Keyed-Hash Message Authentication Code (HMAC)*, March, 2002
- US DOC/NIST: FIPS PUB 186-2 Digital Signature Standard, January, 2000

extools.h and extrool.c files

This appendix includes two example files that are used in the examples throughout this manual.

The extools.h file is the header file that helps you to compile and execute the examples in Chapter 16 with different compilers.

```
/* ////////////////////////////////////////////////////////////////////////
// Name:       extools.h
// Purpose:    Manual's ippIP Example Tools Header file
//
*/
#if !defined( __EXAMPLE_TOOL_H__ )
#define __EXAMPLE_TOOL_H__

#include "ipp.h"


// main() function depend on target environment
#if defined( linux ) || defined( __GNUC__ )
#  include <stdio.h>
#  define _T(x) x
#  define _tprintf printf
#  define MAIN() main(void)
typedef char _TCHAR;

#elif defined( _WIN32_WCE )
#  include <windows.h>
#  define MAIN() WINAPI WinMain( HINSTANCE, HINSTANCE, LPWSTR, int )

#elif defined( _WIN32 )
#  include <windows.h>
#  include <tchar.h>
#  include <stdio.h>
#  define MAIN() main(void)
```

```
#else
#  error _WIN32_WCE, linux or __GNUC__ should be defined

#endif

// use Intel Performance Primitives
#include "ippIP.h"

// Returns address of ROI removed at a Location position relatively pImg
Ipp8u* AddressOf(Ipp8u* pImg, int sLine, IppiPoint aLocation);

// Print ROI content of one-channel image
void PrintROI_C1(const _TCHAR* pTitle, Ipp8u*  pROI, int sLine, IppiSize aSize);
void PrintROI_C1(const _TCHAR* pTitle, Ipp16s* pROI, int sLine, IppiSize aSize);
void PrintROI_C1(const _TCHAR* pTitle, Ipp32s* pROI, int sLine, IppiSize aSize);


#endif // __EXAMPLE_TOOL_H__
```

The following code is from the extrool.c files and can be used to show or display results:

```
void PrintROI_C1(const _TCHAR* pTitle, Ipp16s* pROI, int sLine, IppiSize aSize)
{
   _tprintf(_T("%s\n"), pTitle);

   for(int y=0; y<aSize.height; y++) {
      for(int x=0; x<aSize.width; x++) {
         _tprintf(_T("% -6d "), pROI[x]);
      }
      _tprintf(_T("\n"));
      pROI = (Ipp16s*)( (Ipp8u*)pROI +sLine );
   }
}
```