Intel® Integrated Performance Primitives for Intel® Architecture

Reference Manual

Volume 1:
Signal Processing

Issued in U.S.A.
Document Number: A24968-4002

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

Version	Version Information	Date
-1001	Original issue.	09/00
-1002	Documents Intel IPP 1.0 final release. Functions NormDiff, AutoCorr, and ZeroMean have been added. Derivatives Functions section have been revised.	02/01
-1101	Documents Intel IPP 1.1 beta release.	04/01
-2001	Documents Intel IPP 2.0 beta release. General audio coding, MP3, and transcendental vector functions have been added. Speech recognition API have been revised.	08/01
-2002	Documents Intel IPP 2.0 Gold release. New Intel IPP common functions have been added. The set of arithmetic, vector initialization, statistical, and filtering functions have been expanded.	11/01
-3001	Documents Intel IPP 3.0 pre-beta. New speech codec functions have been added. The set of conversion and windowing functions have been expanded. Speech recognition API have been updated.	04/02
-3002	Documents Intel IPP 3.0 beta.	06/02
-3003	Documents Intel IPP 3.0 beta update.	09/02
-3004	Documents Intel IPP 3.0 gold release.	11/02
-4001	Documents Intel IPP 4.0 beta. New functions for cross-architecture development have been added.	05/03
-4002	Documents Intel IPP 4.0 gold release.	10/03

Contents

Overview 1

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

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

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

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

About This Software

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

Hardware and Software Requirements

The Intel IPP for Intel architecture software runs on personal computers that are based on IA-32 processors or Itanium® architecture-based processors and running Microsoft Windows* 2000, Windows ME, Windows XP, or Linux*. The Intel IPP integrates into the customer's application or library written in C or C++.

Platforms Supported

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

Cross-Architecture Alignment

Cross Architecture Overview

Intel IPP has been designed to support application development on various Intel® architectures. Previously Intel IPP was offered in two separate products, one for the Intel® Pentium®, Xeon™ and Itanium® processors and one for the Intel® PCA processors based on Intel Xscale® technology. While these separate packages included many similarities, prior to version 4.0, there were some differences in both functionality and interface.

With rising interest in cross architecture development, the Intel IPP development teams have worked to bridge these differences to allow for easy development of applications for multiple Intel® platforms. Intel IPP 4.0 represents the result of this effort and provides alignment of the two separate packages into a single offering that includes support for all of these architectures. This includes interface alignment and full API alignment for all functions that are relevant to both architectures.

With this release, the functions available for Intel Pentium, Xeon and Itanium processors represent a superset of the functions available for the Intel PCA processors. This means the API definition is common for all processors, while the underlying function implementation takes into account the variations in processor architectures.

By providing a single cross-architecture API, Intel IPP 4.0 allows software application repurposing and enables developers to port to unique features across Intel® processor-based desktop, server, mobile and handheld platforms. Developers can write their code once, to realize application performance over many processor generations.

For additional information on API additions and changes from version 3.0 to version 4.0 please refer to the product release notes and the document *Migration Guide for Intel® IPP Cross Architecture API Alignment* available at http://www.intel.com/software/products/ipp.

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

Table 1-1 Function Coverage in Intel IPP

Function Group	Intel® Pentium® 4 processors	Intel® Itanium® 2 processors	Intel® PCA processors
Signal Processing	available	available	available
Image Processing	available	available	available
JPEG	available	available	available
Speech Recognition	available	available	n/a
Speech Coding	available	available	available
Audio Codecs	available	available	available
Video Codecs	available	available	available
Matrix	available	available	n/a
Vector Math	available	available	n/a
Computer Vision	available	available	n/a
Cryptography	available	available	available

Legacy Code Support

While the alignment of functions for the Intel Pentium, Xeon and Itanium processors and the Intel PCA processors has resulted in some changes to the API, Intel IPP 4.0 also includes legacy support and full compatibility with the APIs used in the previous versions of Intel IPP. This support is available to enable developers to update their applications without requiring the interfaces to be changed. All developers, however, are encouraged to

adopt the new API as this will be the interface used in future versions of Intel IPP including optimizations for future Intel processors. A list of functions that have been changed is given at the end of each respective manual chapter.

Technical Support

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

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

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

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

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

About This Manual

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

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

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

Manual Organization

This manual contains the following chapters:

Chapter 1	"Overview." Introduces the Intel IPP software, provides information on manual organization, and explains notational conventions.
Chapter 2	"Intel Integrated Performance Primitives Concepts." Explains the basic concepts underlying signal processing in Intel IPP and describes the supported data formats and operation modes.
Chapter 3	"Support Functions." Describes functions used to manipulate the signals, such as `Copy` and `Set`. Also describes data conversion and memory allocation functions.
Chapter 4	"Vector Initialization Functions." Describes initialization and sample-generating functions.
Chapter 5	"Essential Vector Functions." Describes vector manipulation functions.
Chapter 6	"Filtering Functions." Detais filtering operations that use linear and non-linear filters.
Chapter 7	"Transform Functions." Describes domain transform functions (Fourier, Wavelet, Cosine).
Chapter 8	"Speech Recognition Functions." Describes functions used in the speech recognition applications.
Chapter 9	"Speech Coding Functions." Describes functions used as buiding blocks for implementing speech codecs.
Chapter 10	"Audio Coding Functions." Includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio codec, MP3 encoder and decoder, as well as for implementation of a portable optimized MPEG-4 AAC Main profile decoder and a portable optimized MPEG-1, 2 Layer III encoder.
Chapter 11	"String Functions." Describes functions that operate on strings of symbols.

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

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

Function Descriptions

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

Arguments	Specifies all the function arguments.
Discussion	Defines the function and describes the operation performed by the function. This section may also include the code examples and descriptive equations.
Application Notes	If present, describes any special information which application programmers or other users of the function need to know.
Return Value	Explains the value returned by the function. Most commonly, it lists error codes that the function returns.
See Also	If present, lists the names of functions which perform related tasks.

Audience for This Manual

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

Online Version

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

Related Publications

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

Notational Conventions

In this manual, notational conventions include:

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

Font Conventions

The following font conventions are used throughout the manual:

`THIS TYPE STYLE`	Used in the text for Intel IPP constant identifiers; for example, `IPP_MAX_64S`.
`This type style`	Mixed with the uppercase in structure names as in `IppLibraryVersion`; also used in function names, code examples and call statements; for example, `void ippsFree()`.
`This type style`	Variables in arguments discussion; for example, *`val`*, *`srcLen`*.

Signal Name Conventions

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

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

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

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

Naming Conventions

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

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

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

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

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

Intel® Integrated Performance Primitives Concepts

2

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

Basic Features

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

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

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

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

Intel IPP is well-suited for cross-platform applications. For example, the functions developed for IA-32 platform can be readily ported to Itanium® -based platforms and systems with Intel® StrongARM* technology or Intel XScale® technology. For more information on platform compatibility, see Cross-Architecture Alignment section in chapter 1. In addition, each Intel IPP function has its reference code written in ANSI C, which clearly presents the algorithm used and provides for compatibility with different operating systems.

Function Naming

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

Function names in Intel IPP have the following general format:

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

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

Data-Domain

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

`s`	for signals (expected data type is a 1D signal);
`i`	for images and video (expected data type is a 2D image);
`m`	for matrices (expected data type is a matrix).

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

Name

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

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

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

Data Types

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

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

where

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

and

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

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

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

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

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

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

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

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

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

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

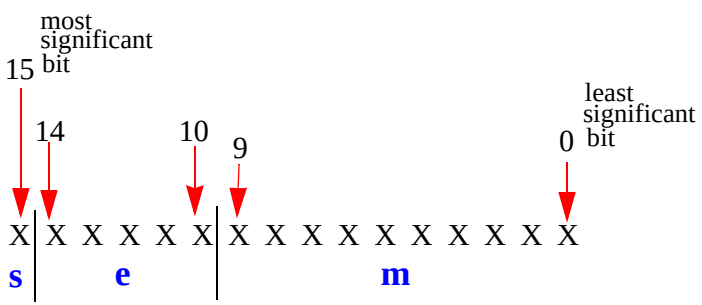
There are several data types, namely 24u, 24s and 16f, that are not supported by the Intel IPP but can be readily converted to supported data types for further processing by the library functions.

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

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

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

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



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

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

Descriptor

The *descriptor* field further describes the data associated with the operation. It may contain implied parameters and/or indicate additional required parameters. To minimize the number of code branches in the function and thus reduce potentially unnecessary execution overhead, most of the general functions were split into separate primitive functions, with some of their parameters entering the primitive function name as descriptors.

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

The following descriptors are used in signal processing functions:

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

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

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

Arguments

The *arguments* field specifies the function arguments.

The order of arguments is as follows:

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

The argument name have the following conventions:

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

Structures and Enumerators

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

Library Version Structure

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

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

Complex Data Structures

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

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

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

Function Context Structures

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

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

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

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

Enumerators

The `IppStatus` constant enumerates the status values returned by the Intel IPP functions, indicating whether the operation was error-free or not. See “Error Reporting” on page 2-11 for more information on the set of valid status values and corresponding error messages for signal processing functions.

The `IppCmpOp` enumeration defines the type of relational operator to be used in threshold functions on page 5-76:

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

The `IppRoundMode` enumeration defines the rounding mode to be used in conversion functions on page 5-60:

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

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

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

The IppCpuType enumerates processor types returned by the ippGetCpuType function, see page 3-7 .

```
typedef enum {
 /* Enumeration:         Processor:                          */
    ippCpuUnknown = 0x0,
    ippCpuPP,          /* Pentium(R)                         */
    ippCpuPMX,         /* Pentium(R) with MMX(TM)            */
    ippCpuPPR,         /* Pentium(R) Pro                     */
    ippCpuPII,         /* Pentium(R) II                      */
    ippCpuPIII,        /* Pentium(R) III                     */
    ippCpuP4,          /* Pentium(R) 4                       */
    ippCpuP4HT,        /* Pentium(R) 4 with HT enabled       */
    ippCpuITP = 0x10, /* Itanium(R)                          */
    ippCpuITP2         /* Itanium(R) 2                       */
} IppCpuType;
```

The IppWinType enumeration defines the type of window to be used in functions that compute FIR filter coefficients on page 6-54:

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

Data Ranges

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

Table 2-2 Data Types and Ranges

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

† The range for absolute values

Data Alignment

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

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

Integer Scaling

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

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

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

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

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

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

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

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

Error Reporting

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

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

Table 2-3 Error Status Values and Messages

Status	Message
ippStsCpuNotSupportedErr	The target cpu is not supported
ippStsMP3FrameHeaderErr	Error in fields IppMP3FrameHeader structure
ippStsMP3SideInfoErr	Error in fields IppMP3SideInfo structure
ippStsAacPrgNumErr	AAC: Invalid number of elements for one program
ippStsAacSectCbErr	AAC: Invalid section codebook
ippStsAacSfValErr	AAC: Invalid scalefactor value
ippStsAacCoefValErr	AAC: Invalid quantized coefficient value
ippStsAacMaxSfbErr	AAC: Invalid coefficient index
ippStsAacPredSfbErr	AAC: Invalid predicted coefficient index
ippStsAacPlsDataErr	AAC: Invalid pulse data attributes
ippStsAacGainCtrErr	AAC: Gain control not supported
ippStsAacSectErr	AAC: Invalid number of sections
ippStsAacTnsNumFiltErr	AAC: Invalid number of TNS filters
ippStsAacTnsLenErr	AAC: Invalid TNS region length
ippStsAacTnsOrderErr	AAC: Invalid order of TNS filter
ippStsAacTnsCoefResErr	AAC: Invalid bit-resolution for TNS filter coefficients
ippStsAacTnsCoefErr	AAC: Invalid TNS filter coefficients
ippStsAacTnsDirectErr	AAC: Invalid TNS filter direction
ippStsAacTnsProfileErr	AAC: Invalid TNS profile
ippStsAacErr	AAC: Internal error
ippStsAacBitOffsetErr	AAC: Invalid current bit offset in bitstream
ippStsAacAdtsSyncWordErr	AAC: Invalid ADTS syncword
ippStsAacSmplRateIdxErr	AAC: Invalid sample rate index
ippStsAacWinLenErr	AAC: Invalid window length (not short or long)
ippStsAacWinGrpErr	AAC: Invalid number of groups for current window length
ippStsAacWinSeqErr	AAC: Invalid window sequence range
ippStsAacComWinErr	AAC: Invalid common window flag

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

ippStsAacStereoMaskErr	AAC: Invalid stereo mask
ippStsAacChanErr	AAC: Invalid channel number
ippStsAacMonoStereoErr	AAC: Invalid mono-stereo flag
ippStsAacStereoLayerErr	AAC: Invalid this Stereo Layer flag
ippStsAacMonoLayerErr	AAC: Invalid this Mono Layer flag
ippStsAacScalableErr	AAC: Invalid scalable object flag
ippStsAacObjTypeErr	AAC: Invalid audio object type
ippStsAacWinShapeErr	AAC: Invalid window shape
ippStsAacPcmModeErr	AAC: Invalid PCM output interleaving indicator
ippStsVLCUsrTblHeaderErr	VLC: Invalid header inside table
ippStsVLCUsrTblUnsupportedFmtErr	VLC: Unsupported table format
ippStsVLCUsrTblEscAlgTypeErr	VLC: Unsupported Ecs-algorithm
ippStsVLCUsrTblEscCodeLengthErr	VLC: Incorrect Esc-code length inside table header
ippStsVLCUsrTblCodeLengthErr	VLC: Unsupported code length inside table
ippStsVLCInternalTblErr	VLC: Invalid internal table
ippStsVLCInputDataErr	VLC: Invalid input data
ippStsVLCAACEscCodeLengthErr	VLC: Invalid AAC-Esc code length
ippStsIncorrectLSPErr	Incorrect Linear Spectral Pair values
ippStsNoRootFoundErr	No roots are found for equation
ippStsLengthErr	Wrong value of string length
ippStsFBankFreqErr	Incorrect value of the filter bank frequency parameter
ippStsFBankFlagErr	Incorrect value of the filter bank parameter
ippStsFBankErr	Filter bank is not correctly initialized
ippStsNegOccErr	Negative occupation count
ippStsCdbkFlagErr	Incorrect value of the codebook flag parameter
ippStsSVDCnvgErr	No convergence of SVD algorithm
ippStsToneMagnErr	Tone magnitude is less than or equal to zero
ippStsToneFreqErr	Tone frequency is negative, or greater than or equal to 0.5
ippStsTonePhaseErr	Tone phase is negative, or greater than or equal to 2*PI
ippStsTrnglMagnErr	Triangle magnitude is less than or equal to zero

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

ippStsTrnglFreqErr	Triangle frequency is negative, or greater than or equal to 0.5
ippStsTrnglPhaseErr	Triangle phase is negative, or greater than or equal to 2*PI
ippStsTrnglAsymErr	Triangle asymmetry is less than -PI, or greater than or equal to PI
ippStsHugeWinErr	Incorrect size of the Kaiser window
ippStsJaehneErr	Magnitude value is negative
ippStsStepErr	Step value is less than or equal to zero
ippStsDlyLineIndexErr	Invalid value of the delay line sample index
ippStsStrideErr	Stride value is less than the row length
ippStsEpsValErr	Negative epsilon value error
ippStsScaleRangeErr	Scale bounds are out of range
ippStsThresholdErr	Invalid threshold bounds
ippStsWtOffsetErr	Invalid offset value of the wavelet filter
ippStsAnchorErr	Anchor point is outside the mask
ippStsMaskSizeErr	Invalid mask size
ippStsShiftErr	Shift value is less than zero
ippStsSampleFactorErr	Sampling factor is less than or equal to zero
ippStsSamplePhaseErr	Phase value is out of range, $0 \leq phase < factor$
ippStsFIRMRFactorErr	MR FIR sampling factor is less than or equal to zero
ippStsFIRMRPhaseErr	MR FIR sampling phase parameter is negative, or greater than or equal to the sampling factor
ippStsRelFreqErr	Relative frequency value is out of range
ippStsFIRLenErr	Length of a FIR filter is less than or equal to zero
ippStsIIROrderErr	Order of an IIR filter is less than or equal to zero
ippStsResizeFactorErr	Resize factor(s) is less than or equal to zero
ippStsDivByZeroErr	An attempt to divide by zero
ippStsInterpolationErr	Invalid interpolation mode
ippStsMirrorFlipErr	Invalid flip mode
ippStsMoment00ZeroErr	Moment value M(0,0) is too small to continue calculations
ippStsThreshNegLevelErr	Negative value of the level in the threshold operation
ippStsContextMatchErr	Context parameter doesn't match the operation

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

ippStsFftFlagErr	Invalid value of the FFT flag parameter
ippStsFftOrderErr	Invalid value of the FFT order parameter
ippStsMemAllocErr	Not enough memory allocated for the operation
ippStsNullPtrErr	Null pointer error
ippStsSizeErr	Wrong value of the data size
ippStsBadArgErr	Function argument/parameter is bad
ippStsErr	Unknown/unspecified error
ippStsNoErr	No error, it's OK
ippStsNoOperation	No operation has been executed
ippStsMisalignedBuf	Misaligned pointer in operation in which it must be aligned
ippStsSqrtNegArg	Negative value(s) of the argument in the function Sqrt
ippStsInvByZero	Inf result. Zero value was met by InvThresh with zero level
ippStsEvenMedianMaskSize	Even size of the Median Filter mask was replaced by the odd one
ippStsDivByZero	Zero value(s) of the divisor in the function Div
ippStsLnZeroArg	Zero value(s) of the argument in the function Ln
ippStsLnNegArg	Negative value(s) of the argument in the function Ln
ippStsNanArg	Not a Number (NaN) argument value warning
ippStsResFloor	All result values are floored
ippStsOverflow	Overflow occurred in the operation
ippStsZeroOcc	Zero occupation count
ippStsUnderflow	Underflow occurred in the operation
ippStsSingularity	Singularity occurred in the operation
ippStsDomain	Argument is out of the function domain
ippStsNotIntelCpu	The target cpu is not Genuine Intel
ippStsCpuMismatch	The library for given cpu cannot be set
ippStsNotIppFunctionFound	Application does not contain IPP functions calls
ippStsDllNotFoundBestUsed	The newest version of IPP DLL's not found by dispatcher
ippStsNoOperationInDll	The function does nothing in the dynamic version of the library
ippStsOvermuchStrings	Number of destination strings is more than expected
ippStsOverlongString	Length of one of the destination strings is more than expected

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

For example, if the integer function `ippsDiv_8u` meets an attempt to divide a positive value by zero, the function execution is not aborted. The result of the operation is set to the maximum value that can be represented by the source data type, and the function returns the warning status `ippStsDivByZero`. This is the case for the vector-vector operation `ippsDiv`. For the vector-scalar division operation `ippsDivC`, the function behavior is different: if the constant divisor is zero, then the function stops execution and returns immediately with the error status `ippStsDivByZeroErr`.

Code Examples

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

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

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

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

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

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

#define genPRINTcplx(TYPE,FMT) \
void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus st ) { \
   int n; \
   if( st > ippStsNoErr ) \
      printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \
   else if( st < ippStsNoErr ) \
      printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
   printf(" %s ", msg ); \
   for( n=0; n<len; ++n ) printf( FMT, buf[n].re, buf[n].im ); \
   printf("\n" ); \
}
genPRINT( 64f, " %f" )
genPRINT( 32f, " %f" )
genPRINT( 16s, " %d" )

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

Support Functions 3

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

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

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

Table 3-1 Intel IPP Support Functions

Function Base Name	Operation
Version Information Functions	
`GetLibVersion`	Returns information about the active library version.
Memory Allocation Functions	
`ippsMalloc`	Allocates memory aligned to 32-byte boundary.
`ippsFree`	Frees memory allocated by the function `ippsMalloc`.
Common Functions	
`CoreGetStatusString`	Translates a status code into a message.
`CoreGetCpuType`	Returns a processor type.
`CoreGetCpuClocks`	Returns a current value of the time stamp counter (TSC) register.
`GetCpuFreqMhz`	Estimates the processor frequency.
`CoreSetFlushToZero`	Enables or disables flush-to-zero mode.
`CoreSetDenormAreZeros`	Enables or disables denormals-are-zero mode.
`AlignPtr`	Aligns a pointer to the specified number of bytes.
`ippMalloc`	Allocates memory aligned to 32-byte boundary.

Table 3-1 Intel IPP Support Functions (continued)

Function Base Name	Operation
ippFree	Frees memory allocated by the function ippMalloc.
Dispatcher Control Functions	
StaticInit	Performs smart dispatching of merged libraries.
StaticFree	Frees resources taken by the function ippStaticInit.
StaticInitBest	Initializes the most appropriate static code.
StaticInitCpu	Initializes the specified version of the static code.

Version Information Functions

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

GetLibVersion

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

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

Discussion

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

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

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

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

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

Example 3-1 Using the ippsGetLibVersion Function

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

Output:
    ippsa6 v0.0 Alpha 0.0.5.5
```

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

Memory Allocation Functions

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

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

ippsMalloc

Allocates memory aligned to 32-byte boundary.

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

Arguments

len	Number of elements to allocate.

Discussion

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

Return Value

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

ippsFree

Frees memory allocated by the function ippsMalloc.

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

Arguments

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

Discussion

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

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

Common Functions

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

CoreGetStatusString

Translates a status code into a message.

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

Arguments

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

Discussion

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

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

Example 3-2 Using the ippCoreGetStatusString Function

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

Output:
    -8, Null Pointer Error
```

CoreGetCpuType

Returns a processor type.

```
IppCpuType ippCoreGetCpuType (void);
```

Discussion

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

Table 3-2 Processor Types Detected in Intel IPP

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

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

Returned Variable Value	Processor Type
ippCpuPIII	Intel® Pentium® III processor
ippCpuP4	Intel® Pentium® 4 processor
ippCpuP4HT	Intel Pentium 4 processor with Hyper-Threading Technology
ippCpuITP	Intel® Itanium® processor
ippCpuITP2	Intel® Itanium® 2 processor
ippCpuUnknown	Unknown processor

CoreGetCpuClocks

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

```
Ipp64u ippCoreGetCpuClocks (void);
```

Discussion

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

GetCpuFreqMhz

Estimates the processor operating frequency.

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

Arguments

pMhz	Pointer to the output result.

Discussion

The function ippGetCpuFreqMhz is declared in the ippcore.h file. This function estimates the processor operating frequency and returns its value in MHz as an integer stored in *pMhz*.
Note that no exact value of frequency is guaranteed so that the estimated value can vary depending on the processor workload.

Return Value

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

CoreSetFlushToZero

Enables or disables flush-to-zero mode.

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

Arguments

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

Discussion

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

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

Return Value

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

CoreSetDenormAreZeros

Enables or disables denormals-are-zero mode.

```
IppStatus ippCoreSetDenormAreZeros(int value);
```

Arguments

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

Discussion

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

Return Value

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

AlignPtr

Aligns a pointer to the specified number of bytes.

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

Arguments

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

Discussion

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

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

ippMalloc

Allocates memory aligned to 32-byte boundary.

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

Arguments

len	Number of elements to allocate.

Discussion

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

Return Value

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

ippFree

Frees memory allocated by the function `ippMalloc`.

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

Arguments

ptr	Pointer to a memory block to be freed.

Discussion

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

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

Functions to Control Merged Libraries Dispatching

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

StaticInit

Performs smart dispatching of merged libraries.

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

Discussion

The function `ippStaticInit` is declared in the `ippcore.h` file. This function performs the so-called “smart dispatching”. This means that the dispatcher searches for the latest versions of the Intel IPP DLLs. If it finds them, corresponding DLLs are loaded.
Otherwise, the dispatcher calls the function `ippStaticInitBest` to set the most appropriate static code of the Intel IPP software.

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

Return Value

`ippStsNoErr`	Indicates that the latest version of Intel IPP DLLs is successfully found and loaded.

ippStsNoIppFunctionFound	Indicates that the application does not contain calls to Intel IPP functions or has been built with non-emerged Intel IPP library.
ippStsDllNotFoundBestUsed	Indicates that the latest version of Intel IPP DLLs is not found by the dispatcher and the most appropriate static code is used.
ippStsNoOperationInDll	Indicates that there is no such operation in the dynamic version of the library.

StaticFree

Frees resources taken by the function ippStaticInit.

```
IppStatus ippStaticFree(void);
```

Discussion

The function ippStaticFree is declared in the ippcore.h file. This function frees resources taken by the function ippStaticInit and then calls the function ippStaticInitBest. The repetitive calls of the function ippStaticFree are enabled.

The function ippStaticFree should not be used in the driver implementation.

Return Value

ippStsNoErr	Indicates that the resources are freed and the function ippStaticInitBest is called successfully.
ippStsNonIntelCpu	Indicates that the static version of generic code for Intel® Architecture is set.
ippStsNoOperationInDll	Indicates that there is no such operation in the dynamic version of the library.

StaticInitBest

Initializes the most appropriate static code.

```
IppStatus ippStaticInitBest(void);
```

Discussion

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

Return Value

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

StaticInitCpu

Initializes the specified version of the static code.

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

Arguments

cpu	Processor type.

Discussion

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

Return Value

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

Compatibility with version 2.0

Certain common functions in Intel IPP version 2.0 were replaced by functions of the `ippcore` sub-library in the Intel IPP version 3.0. For compatibility, the use of these earlier functions is supported. The Table 3-3 lists both the earlier (replaced) and new functions.

Table 3-3 Intel IPP 3.0 Replaced Functions List

Replaced Functions	New Functions
`ippGetStatusString`	`ippCoreSetStatusString`
`ippGetCpuType`	`ippCoreGetCpuType`
`ippGetCpuClocks`	`ippCoreGetCpuClocks`
`ippSetFlushToZero`	`ippCoreSetFlushToZero`
`ippSetDenormAreZeros`	`ippCoreSetDenormAreZeros`

Vector Initialization Functions

4

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

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

Table 4-1 Intel IPP Vector Initialization Functions

Function Base Name	Operation
Vector Initialization Functions	
Copy	Copies the contents of one vector into another.
Move	Moves the contents of one vector to another vector.
Set	Initializes vector elements to a specified common value.
Zero	Initializes a vector to zero.
Tone and Triangle Generation Functions	
ToneInitAllocQ15	Allocates memory and initializes the tone generator state
ToneFree	Frees memory allocated by the function ippsToneInitAlloc.
ToneGetStateSizeQ15	Computes the length of the tone generator structure.
ToneInitQ15	Initializes the tone generator specification structure.
ToneQ15	Generates a tone in accordance with tone generator specification
Tone_Direct	Generates a tone with a given frequency, phase, and magnitude.
ToneQ15_Direct	Generates a tone with a given frequency, phase, and magnitude.
TriangleInitAllocQ15	Allocates memory and initializes the triangle generator state.
TriangleFree	Frees memory allocated by the function ippsTriangleInitAlloc.
TriangleGetStateSizeQ15	Computes the length of the triangle generator structure.

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

Function Base Name	Operation
TriangleInitQ15	Initializes the triangle generator specification structure.
TriangleQ15	Generates a triangle in accordance with triangle generator specification
Triangle_Direct	Generates a triangle with a given frequency, phase, and magnitude.
TriangleQ15_Direct	Generates a triangle with a given frequency, phase, and magnitude
Uniform Distribution Functions	
RandUniformInitAlloc	Allocates memory and initializes a noise generator with uniform distribution.
RandUniformFree	Closes the uniform distribution generator state.
RandUniformGetSize	Computes the length of the uniform distribution generator structure
RandUniformInit	Initializes a noise generator with uniform distribution.
RandUnifrom	Generates the pseudo-random samples with a uniform distribution.
RandUniform_Direct	Generates the pseudo-random samples with a uniform distribution in direct mode.
Gaussian Distribution Functions	
RandGaussInitAlloc	Allocates memory and initializes a noise generator with Gaussian distribution.
RandGaussFree	Closes the Gaussian distribution generator state.
RandGaussGetSize	Computes the length of the Gaussian distribution generator state
RandGaussInit	Initializes a noise generator with Gaussian distribution.
RandGauss	Generates the pseudo-random samples with a Gaussian distribution.
RandGauss_Direct	Generates pseudo-random samples with a Gaussian distribution in the direct mode.
Special Vector Functions	
VectorJaehne	Creates a Jaehne vector.
VectorRamp	Creates a ramp vector.

Vector Initialization Functions

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

Copy

Copies the contents of one vector into another.

```
IppStatus ippsCopy_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsCopy_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsCopy_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsCopy_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCopy_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCopy_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len );
IppStatus ippsCopy_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Arguments

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

Discussion

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

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

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

Example 4-1 Using the ippsCopy Function

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

Return Value

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

Move

Moves the contents of one vector to another vector.

```
IppStatus ippsMove_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

```
IppStatus ippsMove_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMove_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsMove_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMove_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsMove_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len );
IppStatus ippsMove_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Arguments

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

Discussion

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

Return Value

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

Set

Initializes vector elements to a specified common value.

```
IppStatus ippsSet_8u(Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsSet_16s(Ipp16s val, Ipp16s* pDst, int len);
```

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

Arguments

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

Discussion

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

Example 4-2 shows how to use the function ippsSet.

Example 4-2 Using the ippsSet Function

```
IppStatus set(void) {
      char src[] = “set”;
      return ippsSet_8u(‘0’, src, strlen(src));
}
```

Return Value

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

Zero

Initializes a vector to zero.

```
IppStatus ippsZero_8u(Ipp8u* pDst, int len);
IppStatus ippsZero_16s(Ipp16s* pDst, int len);
IppStatus ippsZero_16sc(Ipp16sc* pDst, int len);
IppStatus ippsZero_32f(Ipp32f* pDst, int len);
IppStatus ippsZero_32fc(Ipp32fc* pDst, int len);
IppStatus ippsZero_64f(Ipp64f* pDst, int len);
IppStatus ippsZero_64fc(Ipp64fc* pDst, int len);
```

Arguments

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

Discussion

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

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

Example 4-3 Using the ippsZero Function

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

Return Value

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

Sample-Generating Functions

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

Tone-Generating Functions

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

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

ToneInitAllocQ15

Allocates memory and initializes the tone generator specification structure.

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

Arguments

pToneState	Pointer to the pointer to the tone generator specification structure.

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

Discussion

The function `ippsToneInitAllocQ15` is declared in the `ipps.h` file. This function allocates memory and initializes the tone generator structure *pToneState* with the specified frequency *rFreqQ15*, phase *phaseQ15*, and magnitude *magn*.
Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0, 2π) for phase.

Return Value

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

ToneFree

Frees memory allocated by the function `ippsToneInitAllocQ15`.

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

Arguments

pToneState	Pointer to the tone generator specification structure.

Discussion

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

Return Value

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

ToneGetStateSizeQ15

Computes the length of the tone generator structure.

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

Arguments

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

Discussion

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

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the pointer *pToneStateSize* is NULL.

ToneInitQ15

Initializes the tone generator specification structure.

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

Arguments

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

Discussion

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

Return Value

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

ippStsToneFreqErr	Indicates an error when *rFreqQ15* is negative, or greater than 16383.
ippStsTonePhaseErr	Indicates an error when the *phaseQ15* value is negative, or greater than 205886.

ToneQ15

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

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

Arguments

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

Discussion

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

Return Value

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

Tone_Direct

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

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

Arguments

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

Discussion

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

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

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

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

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

Return Value

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

ToneQ15_Direct

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

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

Arguments

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

Discussion

The function ippsToneQ15_Direct is declared in the ipps.h file. This function generates the tone with the specified frequency *rFreqQ15*, phase *pPhaseQ15*, and magnitude *magn*. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0, 2π) for phase. The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values *x*[*n*] are computed using the same formulas as in ippsTone_Direct function for computing real tones.

Return Value

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

Triangle-Generating Functions

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

Application Notes

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

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

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

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

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

$$H = \pi + h$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TriangleInitAllocQ15

Allocates memory and initializes the triangle generator specification structure.

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

Arguments

pTriangleState	Pointer to the pointer to the triangle generator specification structure.
magn	Magnitude of the tone, that is, the maximum value attained by the wave.

rFreqQ15	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
phaseQ15	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
asymQ15	Asymmetry *h* of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If *h*=0, then the triangle is symmetric and a direct analog of a tone.

Discussion

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

Return Value

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

TriangleFree

Frees memory allocated by the function `ippsTriangleInitAlloc`.

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

Arguments

`pTriangleState`	Pointer to the triangle generator specification structure.

Discussion

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

Return Value

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

TriangleGetStateSizeQ15

Computes the length of the triangle generator structure.

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

Arguments

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

Discussion

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

Return Value

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

TriangleInitQ15

Initializes the triangle generator specification structure.

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

Arguments

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

Discussion

The function `ippsTriangleInitQ15` is declared in the `ipps.h` file. This function initializes the triangle generator structure *`pTriangleState`* with the specified magnitude *`magn`*, frequency *`rFreqQ15`*, phase *`phaseQ15`*, and asymmetry *`asymQ15`*. The structure is allocated in the external buffer, the size of which should be computed by the function `ippsTriangleGetStateSizeQ15`.
Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2π) for the phase, and (-π, π) for the asymmetry.

Return Value

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

TriangleQ15

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

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

Arguments

pDst	Pointer to the array which stores the generated samples.
len	Number of samples to be computed.
pTriangleState	Pointer to the triangle generator specification structure.

Discussion

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

Return Value

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

Triangle_Direct

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

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

```
IppStatus ippsTriangle_Direct_64f(Ipp64f* pDst, int len, double magn,
     double rFreq, double asym, double* pPhase);
IppStatus ippsTriangle_Direct_64fc(Ipp64fc* pDst, int len, double magn,
     double rFreq, double asym, double* pPhase);
```

Arguments

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

Discussion

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

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

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

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

See “Application Notes” on page 4-16 for the definition of functions $\mathbf{ct}_h$ and $\mathbf{st}_h$.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pDst* or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.
ippStsTrnglMagnErr	Indicates an error when *magn* is less than or equal to zero.
ippStsTrnglFreqErr	Indicates an error when *rFreq* is negative, or greater than or equal to 0.5.
ippStsTrnglPhaseErr	Indicates an error when the *pPhase* value is negative, or greater than or equal to IPP_2PI.
ippStsTrnglAsymErr	Indicates an error when *asym* is less than -IPP_PI, or greater than or equal to IPP_PI.

TriangleQ15_Direct

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

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

Arguments

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

Discussion

The function ippsTriangleQ15_Direct is declared in the ipps.h file. This function generates the triangle with the specified magnitude *magn*, frequency *rFreqQ15*, phase *pPhaseQ15*, and asymmetry *asymQ15*. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2π) for the phase, and [-π, π) for the asymmetry. The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values *x*[*n*] are computed using the same formulas as in ippsTriangle_Direct function for computing real triangles.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.
ippStsToneMagnErr	Indicates an error when *magn* is less than or equal to zero.
ippStsToneFreqErr	Indicates an error when *rFreqQ15* is negative, or greater than 16383.
ippStsTonePhaseErr	Indicates an error when the *phaseQ15* value is negative, or greater than 205886.
ippStsTriangleAsymErr	Indicates an error when the *asymQ15* value is less than -102943 or greater than 102943.

Uniform Distribution Functions

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

RandUniformInitAlloc

Allocates memory and initializes a noise generator with uniform distribution.

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

Arguments

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

Discussion

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

Return Value

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pRandUniState* pointer is NULL.
ippStsMemAllocErr	Indicates an error when there is not enough memory for the operation.

RandUniformFree

Closes the uniform distribution generator state.

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

Arguments

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

Discussion

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

Return Value

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

RandUniformInit

Initializes a noise generator with uniform distribution.

```
IppStatusippsRandUniformInit_16s(IppsRandUniState_16s*pRandUniState,
      Ipp16s low, Ipp16s high, unsigned int seed);
```

Arguments

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

Discussion

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

Return Value

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

RandUniformGetSize

Computes the length of the uniform distribution generator structure.

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

Arguments

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

Discussion

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

Return Value

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

RandUnifrom

Generates the pseudo-random samples with a uniform distribution.

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

```
IppStatus ippsRandUniform_32f(Ipp32f* pDst, int len,
      IppsRandUniState_32f* pRandUniState);
```

Arguments

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

Discussion

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

Return Value

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

RandUniform_Direct

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

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

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

Arguments

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

Discussion

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

Return Value

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

Gaussian Distribution Functions

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

RandGaussInitAlloc

Allocates memory and initializes a noise generator with Gaussian distribution.

```
IppStatus ippsRandGaussInitAlloc_8u(IppsRandGaussState_8u**
     pRandGaussState, Ipp8u mean, Ipp8u stdDev, unsigned int seed);
IppStatus ippsRandGaussInitAlloc_16s(IppsRandGaussState_16s**
     pRandGaussState, Ipp16s mean, Ipp16s stdDev, unsigned int seed);
IppStatus ippsRandGaussInitAlloc_32f(IppsRandGaussState_32f**
     pRandGaussState, Ipp32f mean, Ipp32f stdDev, unsigned int seed);
```

Arguments

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

Discussion

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

Return Value

`ppStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pRandGaussState* pointer is NULL.
ippStsMemAllocErr	Indicates an error when there is not enough memory for the operation.

RandGaussFree

Closes the Gaussian distribution generator state.

```
IppStatus ippsRandGaussFree_8u(IppsRandGaussState_8u* pRandGaussState);
IppStatusippsRandGaussFree_16s(IppsRandGaussState_16s*pRandGaussState);
IppStatusippsRandGaussFree_32f(IppsRandGaussState_32f*pRandGaussState);
```

Arguments

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

Discussion

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

Return Value

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

RandGaussGetSize

Computes the length of the Gaussian distribution generator structure.

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

Arguments

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

Discussion

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

Return Value

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

RandGaussInit

Initializes a noise generator with Gaussian distribution.

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

Arguments

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

Discussion

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

Return Value

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

RandGauss

Generates the pseudo-random samples with a Gaussian distribution.

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

```
IppStatus ippsRandGauss_32f(Ipp32f* pDst, int len,
      IppsRandGaussState_32f* pRandGaussState);
```

Arguments

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

Discussion

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

Return Value

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

RandGauss_Direct

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

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

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

Arguments

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

Discussion

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

Return Value

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

Special Vector Functions

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

VectorJaehne

Creates a Jaehne vector.

```
IppStatus ippsVectorJaehne_8u(Ipp8u* pDst, int len, Ipp8u magn);
IppStatus ippsVectorJaehne_8s(Ipp8s* pDst, int len, Ipp8s magn);
IppStatus ippsVectorJaehne_16u(Ipp16u* pDst, int len, Ipp16u magn);
IppStatus ippsVectorJaehne_16s(Ipp16s* pDst, int len, Ipp16s magn);
IppStatus ippsVectorJaehne_32u(Ipp32u* pDst, int len, Ipp32u magn);
IppStatus ippsVectorJaehne_32s(Ipp32s* pDst, int len, Ipp32s magn);
IppStatus ippsVectorJaehne_32f(Ipp32f* pDst, int len, Ipp32f magn);
IppStatus ippsVectorJaehne_64f(Ipp64f* pDst, int len, Ipp64f magn);
```

Arguments

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

Discussion

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

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

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

Example 4-4 Using the ippsVectorJaehne Function

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

Return Value

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

VectorRamp

Creates a ramp vector.

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

```
IppStatus ippsVectorRamp_64f(Ipp64f* pDst, int len, float offset,
     float slope);
```

Arguments

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

Discussion

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

pDst[n] = *offset* + *slope***n*, 0≤*n*<*len*.

Note that linear transform coefficients *offset* and *slope* have floating-point values for all function flavors.

Return Value

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

Essential Vector Functions 5

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

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

Table 5-1 Intel IPP Essential Vector Functions

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

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

Function Base Name	Operation
SubC	Subtracts a constant value from each element of a vector.
SubCRev	Subtracts each element of a vector from a constant value.
Sub	Subtracts the elements of two vectors.
DivC	Divides each element of a vector by a constant value.
DivCRev	Divides a constant value by each element of a vector.
Div	Divides the elements of two vectors.
Abs	Computes absolute values of vector elements.
Sqr	Computes a square of each element of a vector.
Sqrt	Computes a square root of each element of a vector.
Cubrt	Computes cube root of each element of a vector.
Exp	Computes e to the power of each element of a vector.
Ln	Computes the natural logarithm of each element of a vector.
10Log10	Computes the decimal logarithm of each element of a vector and multiplies it by 10.
SumLn	Sums natural logarithms of each element of a vector.
Arctan	Computes the inverse tangent of each element of a vector.
Normalize	Normalizes elements of a real or complex vector using offset and division operations.
Conversion Functions	
SortAscend, SortDescend	Sorts all elements of a vector.
SwapBytes	Reverses byte order of a vector.
Convert	Converts the data type of a vector and stores the results in a second vector.
Join	Converts the floating-point data of several vectors to integer data, and stores the results in a single vector.
Conj	Stores the complex conjugate values of a vector in a second vector or in-place.
ConjFlip	Computes the complex conjugate of a vector and stores the result in reverse order.
Magnitude	Computes the magnitudes of the elements of a complex vector.

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

Function Base Name	Operation
MagSquared	Computes the squared magnitudes of the elements of a complex vector.
Phase	Computes the phase angles of elements of a complex vector.
PowerSpectr	Computes the power spectrum of a complex vector.
Real	Returns the real part of a complex vector in a second vector.
Imag	Returns the imaginary part of a complex vector in a second vector.
RealToCplx	Returns a complex vector constructed from the real and imaginary parts of two real vectors.
CplxToReal	Returns the real and imaginary parts of a complex vector in two respective vectors.
Threshold Threshold_LT, Threshold_GT Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal	Performs the threshold operation on the elements of a vector by limiting the element values by *level*.
Threshold_LTInv	Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.
CartToPolar	Converts the elements of a complex vector to polar coordinate form.
PolarToCart	Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.
MaxOrder	Computes the maximum order of a vector.
Preemphasize	Computes preemphasis of a single precision real signal in-place.
Flip	Reverses the order of elements in a vector.
FindNearestOne	Finds an element of the table which is closest to the specified value.
FindNearest	Finds table elements that are closest to the elements of the specified vector.
Viterbi Decoder Functions	
GetVarPointDV	Fills the array with the information about points that are closest to the received point.

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

Function Base Name	Operation
CalcStatesDV	Calculates the states of the Viterbi decoder.
BuildSymblTableDV4D	Fills the array with the information about possible 4D symbols.
UpdatePathMetricsDV	Searches for the state with the minimum path metric.
Windowing Functions	
WinBartlett	Multiplies a vector by a Bartlett windowing function.
WinBlackman	Multiplies a vector by a Blackman windowing function.
WinHamming	Multiplies a vector by a Hamming windowing function.
WinHann	Multiplies a vector by a Hann windowing function.
WinKaiser	Multiplies a vector by a Kaiser windowing function.
Statistical Functions	
Sum	Computes the sum of the elements of a vector.
Max	Returns the maximum value of a vector.
MaxIndx	Returns the maximum value of a vector and the index of the maximum element.
Min	Returns the minimum value of a vector.
MinIndx	Returns the minimum value of a vector and the index of the minimum element.
MinMax	Returns the maximum and minimum values of a vector.
MinMaxIndx	Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.
Mean	Computes the mean value of a vector.
StdDev	Computes the standard deviation value of a vector.
Norm	Computes the C, L1, or L2 norm of a vector.
NormDiff	Computes the C, L1, or L2 norm of two vectors' difference.
DotProd	Computes the dot product of two vectors.
MaxEvery, MinEvery	Computes maximum or minimum value for each pair of elements of two vectors.
Sampling Functions	

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

Function Base Name	Operation
SampleUp	Up-samples a signal, conceptually increasing its sampling rate by an integer factor.
SampleDown	Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

Logical and Shift Functions

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

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

AndC

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

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

Arguments

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

Discussion

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

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

Return Value

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

And

Computes the bitwise AND of two vectors.

```
IppStatus ippsAnd_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     Ipp8u* pDst, int len);
IppStatus ippsAnd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
     Ipp16u* pDst, int len);
IppStatus ippsAnd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2,
     Ipp32u* pDst, int len);
```

```
IppStatus ippsAnd_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsAnd_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsAnd_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

Arguments

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

Discussion

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

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

Return Value

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

OrC

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

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

Arguments

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

Discussion

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

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

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Or

Computes the bitwise OR of two vectors.

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

Arguments

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

Discussion

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

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

Return Value

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

XorC

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

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

Arguments

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

Discussion

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

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

Return Value

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

Xor

Computes the bitwise XOR of two vectors.

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

Arguments

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

Discussion

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

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

Return Value

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

Not

Computes the bitwise NOT of the vector elements.

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

Arguments

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

Discussion

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

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

Return Value

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

LShiftC

Shifts bits in vector elements to the left.

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

Arguments

val	Number of bits by which the function shifts each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.

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

Discussion

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

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

Return Value

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

RShiftC

Shifts bits in vector elements to the right.

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

Arguments

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

Discussion

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

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

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

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

Return Value

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

Example 5-1 Using the Logical and Shift Functions

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

Arithmetic Functions

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

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

AddC

Adds a constant value to each element of a vector.

```
IppStatus ippsAddC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
     int len);
IppStatus ippsAddC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
     int len);
IppStatus ippsAddC_32fc(const Ipp32fc* pSrc, Ipp32fc val,Ipp32fc* pDst,
     int len);
IppStatus ippsAddC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst,
     int len);
IppStatus ippsAddC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsAddC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsAddC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsAddC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsAddC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsAddC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int scaleFactor);
IppStatus ippsAddC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
     pDst, int len, int scaleFactor);
IppStatus ippsAddC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s*
     pDst, int len, int scaleFactor);
IppStatus ippsAddC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc*
     pDst, int len, int scaleFactor);
IppStatus ippsAddC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc*
     pDst, int len, int scaleFactor);
IppStatus ippsAddC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsAddC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsAddC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
     int scaleFactor);
```

```
IppStatus ippsAddC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsAddC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
     int scaleFactor);
```

Arguments

val	Scalar value used to increment each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector which stores the result of the addition *pSrc[n]* + *val*.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

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

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

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

Return Value

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

Add

Adds the elements of two vectors.

```
IppStatus ippsAdd_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len);
IppStatus ippsAdd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
      Ipp16u* pDst, int len);
IppStatus ippsAdd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2,
      Ipp32u* pDst, int len);
IppStatus ippsAdd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int len);
IppStatus ippsAdd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
      Ipp64f* pDst, int len);
IppStatus ippsAdd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
      Ipp32fc* pDst, int len);
IppStatus ippsAdd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
      Ipp64fc* pDst, int len);
IppStatus ippsAdd_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
      Ipp16u* pDst, int len);
IppStatus ippsAdd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp32f* pDst, int len);
IppStatus ippsAdd_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsAdd_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsAdd_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsAdd_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsAdd_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsAdd_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
      Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsAdd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsAdd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
      Ipp32s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsAdd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
     Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsAdd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
     Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsAdd_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsAdd_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsAdd_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len,
      int scaleFactor);
IppStatus ippsAdd_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int
     len, int scaleFactor);
IppStatus ippsAdd_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int
     len, int scaleFactor)
```

Arguments

pSrc1, pSrc2	Pointers to the vectors whose elements are to be added together.
pDst	Pointer to the destination vector which stores the result of the addition *pSrc2[n]* + *pSrc1[n]*.
pSrc	Pointer to the source vector whose elements are to be added to the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

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

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

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

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

Return Value

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

Example 5-2 Using the ippsAdd Function

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

AddProduct

Adds product of two vectors to the accumulator vector.

```
IppStatus ippsAddProduct_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
     Ipp32f* pSrcDst, int len);
IppStatus ippsAddProduct_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
     Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsAddProduct_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
      Ipp32fc* pSrcDst, int len);
IppStatus ippsAddProduct_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
      Ipp64fc* pSrcDst, int len);
IppStatus ippsAddProduct_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddProduct_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
      Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddProduct_16s32s_Sfs(const Ipp16s* pSrc1,
      const Ipp16s* pSrc2, Ipp32s* pSrcDst, int len, int scaleFactor);
```

Arguments

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

Discussion

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

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

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

Return Value

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

MulC

Multiplies each elements of a vector by a constant value.

```
IppStatus ippsMulC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
     int len);
IppStatus ippsMulC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
     int len);
IppStatus ippsMulC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc*
     pDst, int len);
IppStatus ippsMulC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc*
     pDst, int len);
IppStatus ippsMulC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsMulC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsMulC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsMulC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsMulC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsMulC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int scaleFactor);
IppStatus ippsMulC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
     pDst, int len, int scaleFactor);
IppStatus ippsMulC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s*
     pDst, int len, int scaleFactor);
IppStatus ippsMulC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc*
     pDst, int len, int scaleFactor);
IppStatus ippsMulC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc*
     pDst, int len, int scaleFactor);
IppStatus ippsMulC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
     int scaleFactor);
```

```
IppStatus ippsMulC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsMulC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsMulC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsMulC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsMulC_32f16s_Sfs(const Ipp32f* pSrc, Ipp32f val, Ipp16s* pDst,
     int len, int scaleFactor);
IppStatus ippsMulC_Low_32f16s(const Ipp32f* pSrc, Ipp32f val, Ipp16s* pDst,
     int len);
```

Arguments

val	The scalar value used to multiply each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc[n] * val*.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	The number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

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

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

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

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

Return Value

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

Mul

Multiplies the elements of two vectors.

```
IppStatus ippsMul_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len);
IppStatus ippsMul_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int len);
IppStatus ippsMul_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
      Ipp64f* pDst, int len);
IppStatus ippsMul_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
      Ipp32fc* pDst, int len);
IppStatus ippsMul_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
      Ipp64fc* pDst, int len);
IppStatus ippsMul_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
      Ipp16u* pDst, int len);
IppStatus ippsMul_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp32f* pDst, int len);

IppStatus ippsMul_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMul_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMul_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsMul_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsMul_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);

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

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

Arguments

pSrc1, pSrc2	Pointers to the vectors whose elements are to be multiplied.

pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc1[n] * pSrc2[n]*.
pSrc	Pointer to the vector whose elements are to be multiplied by the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

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

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

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

Return Value

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

SubC

Subtracts a constant value from each element of a vector.

```
IppStatus ippsSubC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
     int len);
IppStatus ippsSubC_32fc(const Ipp32fc* pSrc, Ipp32fc val,
     Ipp32fc* pDst, int len)
IppStatus ippsSubC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
     int len);
IppStatus ippsSubC_64fc(const Ipp64fc* pSrc, Ipp64fc val,
     Ipp64fc* pDst, int len);
IppStatus ippsSubC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsSubC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsSubC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int scaleFactor);
IppStatus ippsSubC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val,
     Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSubC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val,
     Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSubC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val,
     Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSubC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val,
     Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsSubC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
     int scaleFactor);
```

```
IppStatus ippsSubC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
     int scaleFactor);
```

Arguments

val	Scalar value used to decrement each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector which stores the result of the subtraction *pSrc[n] - val*.
pSrcDst	Pointer to the vector whose elements are to be decreased by the value *val* and stored in *pSrcDst* for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

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

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

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

Return Value

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

SubCRev

Subtracts each element of a vector from a constant value.

```
IppStatus ippsSubCRev_32f(const Ipp32f* pSrc, Ipp32f val,
     Ipp32f* pDst, int len);
IppStatus ippsSubCRev_64f(const Ipp64f* pSrc, Ipp64f val,
     Ipp64f* pDst, int len);
IppStatus ippsSubCRev_32fc(const Ipp32fc* pSrc, Ipp32fc val,
     Ipp32fc* pDst, int len);
IppStatus ippsSubCRev_64fc(const Ipp64fc* pSrc, Ipp64fc val,
     Ipp64fc* pDst, int len);
IppStatus ippsSubCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubCRev_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubCRev_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubCRev_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsSubCRev_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int scaleFactor);
IppStatus ippsSubCRev_16s_Sfs(const Ipp16s* pSrc, Ipp16s val,
     Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_32s_Sfs(const Ipp32s* pSrc, Ipp32s val,
     Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val,
     Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val,
     Ipp32sc* pDst, int len, int scaleFactor))
IppStatus ippsSubCRev_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst,
     int len, int scaleFactor);
IppStatus ippsSubCRev_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubCRev_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
     int scaleFactor);
```

```
IppStatus ippsSubCRev_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
     int scaleFactor);
IppStatus ippsSubCRev_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
     int scaleFactor);
```

Arguments

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

Discussion

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

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

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

Return Value

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

Sub

Subtracts the elements of two vectors.

```
IppStatus ippsSub_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len);
IppStatus ippsSub_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int len);
IppStatus ippsSub_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
      Ipp64f* pDst, int len);
IppStatus ippsSub_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
      Ipp32fc* pDst, int len);
IppStatus ippsSub_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
      Ipp64fc* pDst, int len);
IppStatus ippsSub_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp32f* pDst, int len);
IppStatus ippsSub_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsSub_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsSub_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsSub_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsSub_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsSub_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
      Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsSub_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSub_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
      Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSub_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
      Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSub_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
      Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsSub_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst,
      int len, int scaleFactor);
```

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

Arguments

pSrc1	Pointer to the source vector whose elements are to be subtracted from *pSrc2*.
pSrc2	Pointer to the source vector whose elements are to be decreased by the elements of *pSrc1*.
pDst	Pointer to the destination vector which stores the result of the subtraction *pSrc2[n]* - *pSrc1[n]*.
pSrc	Pointer to the vector whose elements are to be subtracted from the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the vector whose elements are to be decreased by the elements of *pSrc* and stored in *pSrcDst* for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

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

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

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

Return Value

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

DivC

Divides each element of a vector by a constant value.

```
IppStatus ippsDivC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
     int len);
IppStatus ippsDivC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
     int len);
IppStatus ippsDivC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc*
     pDst, int len);
IppStatus ippsDivC_64fc(const Ipp64fc* pSrc, Ipp64fc val,
     Ipp64fc* pDst, int len);
IppStatus ippsDivC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsDivC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsDivC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsDivC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsDivC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
     int len, int ScaleFactor);
IppStatus ippsDivC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
     pDst, int len, int ScaleFactor);
IppStatus ippsDivC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val,
     Ipp16sc* pDst, int len, int ScaleFactor);
IppStatus ippsDivC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
     int ScaleFactor);
```

```
IppStatus ippsDivC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
     int ScaleFactor);
IppStatus ippsDivC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
      int ScaleFactor);
```

Arguments

val	Scalar value used to divide each element of the vector *pSrc* or *pSrcDst*.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector which stores the result of the division *pSrc[n] / val*.
pSrcDst	Pointer to the vector whose elements are divided by the value *val* and stored in *pSrcDst* for the in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

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

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

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

Return Value

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

DivCRev

Divides a constant value by each element of a vector.

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

Arguments

val	Constant value that is used as a dividend in the operation.
pSrc	Pointer to the source vector whose elements are used as divisors.
pDst	Pointer to the destination vector which stores the result of the division *val / pSrc[n]*.
pSrcDst	Pointer to the vector whose elements are used both as divisors for the in-place operation and to store the resulting quotients.
len	Number of elements in the vector

Discussion

The function ippsDivCRev is declared in the ipps.h file. This function divides the constant value *val* by each element of the vector *pSrc* and stores the results in *pDst*. The in-place flavors of ippsDivC divide the constant value *val* by each element of the vector *pSrcDst* and store the results in *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.

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

Div

Divides the elements of two vectors.

```
IppStatus ippsDiv_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int len);
IppStatus ippsDiv_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
      Ipp64f* pDst, int len);
IppStatus ippsDiv_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
      Ipp32fc* pDst, int len);
IppStatus ippsDiv_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
      Ipp64fc* pDst, int len);
IppStatus ippsDiv_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsDiv_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsDiv_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsDiv_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsDiv_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
      Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsDiv_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsDiv_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
      Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsDiv_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
      Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsDiv_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst,
      int len, int ScaleFactor);
IppStatus ippsDiv_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
      int len, int ScaleFactor);
```

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

Arguments

pSrc1	Pointer to the vector whose elements are used as divisors for the elements of *pSrc2*.
pSrc2	Pointer to the vector whose elements are to be divided by the elements of *pSrc1*.
pDst	Pointer to the destination vector *pDst* which stores the result of the division *pSrc2[n]* / *pSrc1[n]*.
pSrc	Pointer to the source vector whose elements are used as divisors for the elements of *pSrcDst* in case of the in-place operation.
pSrcDst	Pointer to the vector whose elements are to be divided by the elements of *pSrc* and stored in *pSrcDst* for the in-place operation.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

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

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

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

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

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsDivByZero`	Indicates a warning for zero-valued divisor vector element. Operation execution is not aborted. The value of the destination vector element in the floating-point operations:
	`NaN` For zero-valued dividend vector element.
	`+Inf` For positive dividend vector element.
	`-Inf` For negative dividend vector element.

Example 5-3 Using the ippsDiv_16s_ISfs Function

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

Example 5-4 Using the ippsDiv_32f_I Function

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

Abs

Computes absolute values of vector elements.

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

Arguments

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

len	Number of elements in the vector.

Discussion

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

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

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

Return Value

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

Example 5-5 Using the ippsAbs Function

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

Sqr

Computes a square of each element of a vector.

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

Arguments

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

len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

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

$pDst[n] = pSrc[n]^2$

The in-place flavors of ippsSqr compute the square of each element of the vector *pSrcDst* and store the result in *pSrcDst*. The computation is performed as follows:

$pSrcDst[n] = pSrcDst[n]^2$

When computing the square of an integer number, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. Example 5-6 shows how to get the value of 200^2. Without scaling this result is clipped to 32767. Scaling retains the output data range but results in the precision loss in low-order bits.

Return Value

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

Example 5-6 Using the ippsSqr Function

```
IppStatus sqr(void) {
      Ipp16s x[4] = {-3, 2, 30, 200};
      IppStatus st = ippsSqr_16s_ISfs(x, 4, 1);
      printf_16s("sqr =", x, 4, st);
      return st;
}
Output:
    sqr =  4 2 450 20000
```

Sqrt

Computes a square root of each element of a vector.

```
IppStatus ippsSqrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqrt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqrt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqrt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqrt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqrt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqrt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqrt_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqrt_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len,
     int scaleFactor);
IppStatus ippsSqrt_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, int scaleFactor);
IppStatus ippsSqrt_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len,
     int scaleFactor);
```

```
IppStatus ippsSqrt_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, int scaleFactor);
IppStatus ippsSqrt_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len,
     int scaleFactor);
IppStatus ippsSqrt_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16sc_ISfs(Ipp16sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);
```

Arguments

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

Discussion

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

$$pDst[n] = \sqrt{pSrc[n]}$$

The in-place flavors of ippsSqrt compute the square root of each element of the vector *pSrcDst* and store the result in *pSrcDst*. The computation is performed as follows:

$$pSrcDst[n] = \sqrt{pSrcDst[n]}$$

The square root of complex vector elements is computed as follows:

$$\sqrt{a + j \cdot b} = \sqrt{\frac{\sqrt{a^2 + b^2} + a}{2}} + j \cdot \mathrm{sign}(b) \cdot \sqrt{\frac{\sqrt{a^2 + b^2} - a}{2}}$$

Application Notes

If the function `ippsSqrt` encounters a negative value in the input, it returns a warning status. Operation execution is not aborted. To increase precision of an integer output, use the scale factor.

Example 5-7 shows how to call the function `ippsSqrt_16s_ISfs`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc`*, *`pDst`*, *`or pSrcDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsSqrtNegArg`	Indicates a warning. Operation execution is not aborted. The value of the destination vector element is:
	`NaN` For negative input vector element in floating-point operations.
	0 For negative input vector element in integer operations.

Example 5-7 Using the ippsSqrt Function

```
IppStatus sqrt(void) {
      Ipp16s x[4] = {-3, 2, 30, 300};
      IppStatus st = ippsSqrt_16s_ISfs(x, 4, -1);
      printf_16s("sqrt =", x, 4, st);
      return st;
}
Output:
    -- warning 3, Negative value(s) in the argument of the function Sqrt
    sqrt = 0 3 11 35
```

Cubrt

Computes cube root of each element of a vector.

```
IppStatus ippsCubrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCubrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst,
     int len, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsCubrt is declared in the ipps.h file. This function computes cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

$pDst[n] = \sqrt[3]{pSrc[n]}, 0 \le n < len$.

Return Value

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

Exp

Computes e *to the power of each element of a vector.*

```
IppStatus ippsExp_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExp_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsExp_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     int scaleFactor);
IppStatus ippsExp_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
     int scaleFactor);
IppStatus ippsExp_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len,
     int scaleFactor);
IppStatus ippsExp_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);
```

Arguments

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

Discussion

The function ippsExp is declared in the ipps.h file. This function computes the exponential function of each element of the vector *pSrc*, and stores the result in *pDst*.

The computation is performed as follows:

$pDst[n] = e^{pSrc[n]}$

The in-place flavors of ippsExp compute the exponential function of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

The computation is performed as follows:

$pSrcDst[n] = e^{pSrcDst[n]}$

When computing the exponent of an integer number, the output result can exceed the data range and become saturated. The scaling retains the output data range but results in precision loss in low-order bits. The function ippsExp_32f64f computes the output result in a higher precision data range.

Example 5-8 shows how to call the function ippsExp_16s_ISfs. Example 5-9 shows how to call the function ippsExp_64f_I.

Application Notes

For the functions ippsExp and ippsLn the result is rounded to the nearest integer after scaling.

Return Value

ippStsNoErr	Indicates no error.

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

Example 5-8 Using the ippsExp_16s_ISfs Function

```
IppStatus exp16s(void) {
      Ipp16s x[4] = {-1, 2, 30, 0};
      IppStatus st = ippsExp_16s_ISfs(x, 4, -1);
      printf_16s("exp16s =", x, 4, st);
      return st;
}
Output:
    exp16s =  1 15 32767 2
```

Example 5-9 Using the ippsExp_64f_I Function

```
IppStatus exp64f(void) {
      Ipp64f x[4] = {-1, 2, 1, log(1.234567)};
      IppStatus st = ippsExp_64f_I(x, 4);
      printf_64f("exp64f =", x, 4, st);
      return st;
}
Output:
    exp64f =  0.367879 7.389056 2.718282 1.234567
```

Ln

Computes the natural logarithm of each element of a vector.

```
IppStatus ippsLn_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsLn_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsLn_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     int scaleFactor);
IppStatus ippsLn_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
     int scaleFactor);
IppStatus ippsLn_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
     int scaleFactor);
IppStatus ippsLn_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsLn_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
```

Arguments

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

Discussion

The function ippsLn is declared in the ipps.h file. This function computes the natural logarithm of each element of the vector *pSrc* and stores the result in *pDst* as given by

$pDst[n] = \log_e(pSrc[n])$

The in-place flavors of ippsLn compute the natural logarithm of each element of the vector *pSrcDst* and store the result in *pSrcDst* as given by

pSrcDst[n] = $\log_e$ (*pSrcDst[n]*)

Example 5-10 shows how to call the function ippsLn_32f_I.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is -Inf.
ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is NaN.

Example 5-10 Using the ippsLn Function

```
IppStatus ln32f(void) {
      Ipp32f x[4] = {-1, (float)IPP_E, 0, (float)(exp(1.234567))};
      IppStatus st = ippsLn_32f_I(x, 4);
      printf_32f("Ln =", x, 4, st);
      return st;
}
Output:
    -- warning 8, Negative value(s) of argument in the Ln function
    Ln = -1.#IND00 1.000000 -1.#INF00 1.234567
```

10Log10

Computes the decimal logarithm of each element of a vector and multiplies it by 10.

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

Arguments

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

Discussion

The function ipps10Log10 is declared in the ipps.h file. This function computes the decimal logarithm of each element of the vector *pSrc*, multiplies it by 10, and stores the result in *pDst* as given by

$pDst[n] = 10*\log_{10}(pSrc[n])$

The in-place flavor of ipps10Log10 computes the decimal logarithm of each element of the vector *pSrcDst*, multiplies it by 10, and stores the result in *pSrcDst* as given by

$pSrcDst[n] = 10*\log_{10}(pSrcDst[n])$

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted.
ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted.

SumLn

Sums natural logarithms of each element of a vector.

```
IppStatus ippsSumLn_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum);
IppStatus ippsSumLn_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_32f64f(const Ipp32f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pSum);
```

Arguments

pSrc	Pointer to the source vector.
pSum	Pointer to the output result.
len	Number of elements in the vector.

Discussion

The function ippsSumLn is declared in the ipps.h file. This function computes the sum of natural logarithms of each element of the vector *pSrc* and stores the result value in *pSum*. The summation is given by:

$$sum = \sum_{n=0}^{len-1} \ln(pSrc[n])$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSum* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to -Inf.
ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to NaN.

Arctan

Computes the inverse tangent of each element of a vector.

```
IppStatus ippsArctan_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsArctan_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsArctan_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsArctan_64f_I(Ipp64f* pSrcDst, int len);
```

Arguments

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

Discussion

The function ippsAtan is declared in the ipps.h file. This function computes the inverse tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

pDst[n] = arctan(*pSrc*[n]), $0 \le n < len$.

Return Value

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

Normalize

Normalizes elements of a real or complex vector using offset and division operations.

```
IppStatus ippsNormalize_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
      Ipp32f vsub, Ipp32f vdiv);
IppStatus ippsNormalize_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
      Ipp64f vsub, Ipp64f vdiv);
IppStatus ippsNormalize_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int len, Ipp32fc vsub, Ipp32f vdiv);
IppStatus ippsNormalize_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      int len, Ipp64fc vsub, Ipp64f vdiv);
IppStatus ippsNormalize_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
      int len, Ipp16s vsub, int vdiv, int scaleFactor);
IppStatus ippsNormalize_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
      int len, Ipp16sc vsub, int vdiv, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
vsub	Subtrahend value.
vdiv	Denominator value.
pDst	Pointer to the vector which stores the normalized elements.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsNormalize is declared in the ipps.h file. This function subtracts *vsub* from elements of the input vector *pSrc*, divides the differences by *vdiv*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = \frac{pSrc[n] - vsub}{vdiv}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZeroErr	Indicates an error when *vdiv* is equal to 0 or less than the minimum floating-point positive number.

Conversion Functions

The functions described in this section perform the following conversion operations for vectors:

- Sorting all elements of a vector
- Data type conversion (including floating-point to integer and integer to floating-point)
- Joining several vectors
- Extracting components from a complex vector and constructing a complex vector

- Computing the complex conjugates of vectors
- Cartesian to polar and polar to Cartesian coordinate conversion.

This section also describes the Intel IPP functions that extract real and imaginary components from a complex vector or construct a complex vector using its real and imaginary components.
The functions `ippsReal` and `ippsImag` return the real and imaginary parts of a complex vector in a separate vector, respectively.
The function `ippsRealToCplx` constructs a complex vector from real and imaginary components stored in two respective vectors.
The function `ippsCplxToReal` returns the real and imaginary parts of a complex vector in two respective vectors.
The function `ippsMagnitude` computes the magnitude of a complex vector elements.

Additionally this section describes functions that perform the Viterbi decoding for V34 receiver.

SortAscend, SortDescend

Sorts all elements of a vector.

```
IppStatus ippsSortAscend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortAscend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortAscend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortAscend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortAscend_64f_I(Ipp64f* pSrcDst, int len);

IppStatus ippsSortDescend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortDescend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortDescend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortDescend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortDescend_64f_I(Ipp64f* pSrcDst, int len);
```

Arguments

pSrcDst	Pointer to the source and destination vector.
len	Number of elements in the vector.

Discussion

The functions ippsSortAscend and ippsSortDescend are declared in the ipps.h file. These functions rearrange all elements of the specified vector *pSrcDst* in the ascending or descending order, respectively, and store the result in the destination vector *pSrcDst*.

Return Value

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

SwapBytes

Reverses the byte order of a vector.

```
IppStatus ippsSwapBytes_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsSwapBytes_24u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsSwapBytes_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsSwapBytes_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSwapBytes_24u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSwapBytes_32u_I(Ipp32u* pSrcDst, int len);
```

Arguments

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

len	Number of elements in the vector.

Discussion

The function ippsSwapBytes is declared in the ipps.h file. This function reverses the endian order (byte order) of the source vector *pSrc* (*pSrcDst* for the in-place operation) and stores the result in *pDst* (*pSrcDst*).
When the low-order byte is stored in memory at the lowest address, and the high-order byte at the highest address, the little-endian order is implemented.When the high-order byte is stored in memory at the lowest address, and the low-order byte at the highest address, the big-endian order is implemented. The function ippsSwapBytes allows to switch from one order to the other in either direction.

Return Value

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

Convert

Converts the data type of a vector and stores the results in a second vector.

```
IppStatus ippsConvert_8s16s(const Ipp8s* pSrc,Ipp16s* pDst, int len);
IppStatus ippsConvert_8s32f(const Ipp8s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16u32f(const Ipp16u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s16s(const Ipp32s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsConvert_32s32f(const Ipp32s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s64f(const Ipp32s* pSrc, Ipp64f* pDst, int len);
```

```
IppStatus ippsConvert_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32f_Sfs(const Ipp16s* pSrc, Ipp32f* pDst,
      int len, int scaleFactor);
IppStatus ippsConvert_16s64f_Sfs(const Ipp16s* pSrc, Ipp64f* pDst,
      int len, int scaleFactor);
IppStatus ippsConvert_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst,
      int len, int scaleFactor);
IppStatus ippsConvert_32s32f_Sfs(const Ipp32s* pSrc, Ipp32f* pDst,
      int len, int scaleFactor);
IppStatus ippsConvert_32s64f_Sfs(const Ipp32s* pSrc, Ipp64f* pDst,
      int len, int scaleFactor);
IppStatus ippsConvert_32f8s_Sfs(const Ipp32f* pSrc, Ipp8s* pDst,
      int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f8u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst,
      int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16s_Sfs(const Ipp32f* pSrc, Ipp16s* pDst,
      int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16u_Sfs(const Ipp32f* pSrc, Ipp16u* pDst,
      int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f32s_Sfs(const Ipp32f* pSrc, Ipp32s* pDst,
      int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64f32s_Sfs(const Ipp64f* pSrc, Ipp32s* pDst,
      int len, IppRoundMode rndMode, int scaleFactor);

IppStatus ippsConvert_24u32u(const Ipp8u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsConvert_24u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32u24u_Sfs(const Ipp32u* pSrc, Ipp8u* pDst, int len,
      int scaleFactor);
IppStatus ippsConvert_32f24u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
      int scaleFactor);

IppStatus ippsConvert_24s32s(const Ipp8u* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_24s32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsConvert_32s24s_Sfs(const Ipp32s* pSrc, Ipp8u* pDst, int len,
      int scaleFactor);
IppStatus ippsConvert_32f24s_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
      int scaleFactor);

IppStatus ippsConvert_16s16f(const Ipp16s* pSrc, Ipp16f* pDst, int len,
      IppRoundMode rndMode);
IppStatus ippsConvert_32f16f(const Ipp32f* pSrc, Ipp16f* pDst, int len,
      IppRoundMode rndMode);
IppStatus ippsConvert_16f16s_Sfs(const Ipp16f* pSrc, Ipp16s* pDst,
      int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_16f32f(const Ipp16f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.	
pDst	Pointer to the destination vector.	
rndMode	Rounding mode which can be `ippRndZero` or `ippRndNear`:	
	`ippRndZero`	Specifies that floating-point values must be truncated toward zero.
	`ippRndNear`	Specifies that floating-point values must be rounded to the nearest integer.
len	Number of elements in the vector.	
scaleFactor	Refer to Integer Scaling.	

Discussion

The function `ippsConvert` is declared in the `ipps.h` file. This function converts the type of data contained in the vector *pSrc* and stores the results in *pDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. The converted result is saturated if it exceeds the output data range.

Return Value

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

Join

Converts the floating-point data of several vectors to integer data, and stores the results in a single vector.

```
IppStatus ippsJoin_32f16s_D2L(const Ipp32f** pSrc, int nChannels,
     int chanLen, Ipp16s* pDst);
```

Arguments

pSrc	Pointer to an array of pointers to the input vectors.
pDst	Pointer to the output vector.
nChannels	Number of vectors.
chanLen	Number of elements in each input vector.

Discussion

The function ippsJoin is declared in the ipps.h file. This function converts floating-point data of *nChannels* input vectors stored in the array *pSrc* to integer data type and writes the results to the destination vector *pDst* in the following order: first element of first vector, first element of second vector, ..., first element of the last vector in the array; second element of first vector, second element of second vector, and so on.
The converted value is saturated if it exceeds the output data range.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error condition if *nChannels* or *chanLen* is less than or equal to 0.

Conj

Stores the complex conjugate values of a vector in a second vector or in-place.

```
IppStatus ippsConj_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsConj_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsConj_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsConj_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsConj_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsConj_64fc_I(Ipp64fc* pSrcDst, int len);
```

Arguments

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

Discussion

The function ippsConj is declared in the ipps.h file. This function stores in *pDst* the element-wise conjugation of the complex vector *pSrc*. The element-wise conjugation of the vector is defined as follows:

```
pDst[n].re = pSrc[n].re
pDst[n].im = - pSrc[n].im
```

The in-place flavors of ippsConj store in *pSrcDst* the element-wise conjugation of the complex vector *pSrcDst*.

The element-wise conjugation of the vector is defined as follows:

```
pSrcDst[n].re = pSrcDst[n].re
pSrcDst[n].im = - pSrcDst[n].im
```

Return Value

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

ConjFlip

Computes the complex conjugate of a vector and stores the result in reverse order.

```
IppStatus ippsConjFlip_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len);
IppStatus ippsConjFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len);
IppStatus ippsConjFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len);
```

Arguments

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

Discussion

The function ippsConjFlip is declared in the ipps.h file. This function computes the conjugate of the vector *pSrc* and stores the result, in reverse order, in *pDst*.

The complex conjugate, stored in reverse order, is defined as follows:

pDst[n] = conj(pSrc[len - n - 1])

The in-place flavors of ippsConjFlip compute the conjugate of the vector *pSrcDst* and store the result, in reverse order, in *pSrcDst*.

The complex conjugate, stored in reverse order, is defined as follows:

pSrcDst[n] = conj(pSrcDst[len - n - 1])

Note that if *pSrc* and *pDst* overlap in memory, the function returns unpredictable results.

Return Value

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

Magnitude

Computes the magnitudes of the elements of a complex vector.

```
IppStatus ippsMagnitude_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
     Ipp32f* pDst, int len);
IppStatus ippsMagnitude_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
     Ipp64f* pDst, int len);
IppStatus ippsMagnitude_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
```

```
IppStatus ippsMagnitude_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
     Ipp32f* pDst, int len);
IppStatus ippsMagnitude_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst,
     int len);
IppStatus ippsMagnitude_16s_Sfs(const Ipp16s* pSrcRe,
     const Ipp16s* pSrcIm, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsMagnitude_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst,
     int len, int scaleFactor);
IppStatus ippsMagnitude_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst,
     int len, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the vector with the real parts of complex elements.
pSrcIm	Pointer to the vector with the imaginary parts of complex elements.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsMagnitude is declared in the ipps.h file. The complex flavor of this function computes the element-wise magnitude of the complex vector *pSrc* and stores the result in *pDst*. The element-wise magnitude is defined by the formula:

$$magn[n] = \sqrt{pSrc[n].re^2 + pSrc[n].im^2}$$

The real flavor of the function ippsMagnitude computes the element-wise magnitude of the complex vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the result in *pDst*. The element-wise magnitude is defined by the formula:

$$magn[n] = \sqrt{pSrcRe[n]^2 + pSrcIm[n]^2}$$

Example 5-11 verifies the identity $\sin^2 x + \cos^2 x = 1$.

Return Value

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

Example 5-11 Using the ippsMagnitude Function

```
void magn(void) {
      Ipp64f x[6], magn[4];
      int n;
      for (n = 0; n<6; ++n) x[n] = sin(IPP_2PI * n / 8);
      ippsMagnitude_64f(x, x+2, magn, 4);
      printf_64f("magn =", magn, 4, ippStsNoErr);
}
Output:
    magn =  1.000000 1.000000 1.000000 1.000000
Matlab* Analog:
    >> n = 0:9; x = sin(2*pi*n/8); z = [x(1:8)+j*x(3:10)]; abs(z(1:4))
```

MagSquared

Computes the squared magnitudes of the elements of a complex vector.

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

Arguments

`pSrc`	Pointer to the source vector.

pDst	Pointer to the destination vector.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsMagSquared is declared in the ipps.h file. This function computes the element-wise squared magnitude of the complex vector *pSrc* and stores the result in *pDst*. The element-wise squared magnitude is defined by the formula:

$magn[n] = pSrc[n].re^2 + pSrc[n].im^2$

Return Value

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

Phase

Computes the phase angles of elements of a complex vector.

```
IppStatus ippsPhase_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPhase_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len,
     int scaleFactor);
IppStatus ippsPhase_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len,
     int scaleFactor);
IppStatus ippsPhase_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
     Ipp64f* pDst, int len);
```

```
IppStatus ippsPhase_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
     Ipp32f* pDst, int len);
IppStatus ippsPhase_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
     Ipp32f* pDst, int len);
IppStatus ippsPhase_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
     Ipp16s* pDst, int len, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components.
pSrcIm	Pointer to the source vector which stores the imaginary components.
pDst	Pointer to the vector which stores the phase (angle) components of the elements in radians. Phase values are in the range (-π,π].
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsPhase is declared in the ipps.h file. This function returns the phase angles of elements of the complex input vector *pSrc*, or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the result in the vector *pDst*. Phase values are returned in radians and are in the range (-π, π].

Return Value

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

PowerSpectr

Computes the power spectrum of a complex vector.

```
IppStatus ippsPowerSpectr_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPowerSpectr_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPowerSpectr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst,int
     len, int scaleFactor);
IppStatus ippsPowerSpectr_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst,
     int len);
IppStatus ippsPowerSpectr_64f(const Ipp64f* pSrcRe, const Ipp64f*
     pSrcIm, Ipp64f* pDst, int len);
IppStatus ippsPowerSpectr_32f(const Ipp32f* pSrcRe, const Ipp32f*
     pSrcIm, Ipp32f* pDst, int len);
IppStatus ippsPowerSpectr_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
     pSrcIm, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsPowerSpectr_16s32f(const Ipp16s* pSrcRe, const Ipp16s*
     pSrcIm, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components.
pSrcIm	Pointer to the source vector which stores the imaginary components.
pDst	Pointer to the vector which stores the spectrum components of the elements.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsPowerSpectr is declared in the ipps.h file. This function returns the power spectrum of the complex input vector *pSrc*, or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the results in the vector *pDst*. The power spectrum elements are squares of the magnitudes of the complex input vector elements:

pDst[n] = (*pSrc[n].re*)^2 + (*pSrc[n].im*)^2 , or
pDst[n] = (*pSrcRe[n]*)^2 + (*pSrcIm[n]*)^2 .

To compute magnitudes, use the function ippsMagnitude on page 5-66.

Return Value

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

Real

Returns the real part of a complex vector in a second vector.

```
IppStatus ippsReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, int len);
IppStatus ippsReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, int len);
IppStatus ippsReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, int len);
```

Arguments

pSrc	Pointer to the complex source vector.
pDstRe	Pointer to the destination vector with real parts.
len	Number of elements in the vector.

Discussion

The function ippsReal is declared in the ipps.h file. This function returns the real part of the complex vector *pSrc* in the vector *pDstRe*.

Return Value

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

Imag

Returns the imaginary part of a complex vector in a second vector.

```
IppStatus ippsImag_16sc(const Ipp16sc* pSrc, Ipp16s* pDstIm, int len);
IppStatus ippsImag_32fc(const Ipp32fc* pSrc, Ipp32f* pDstIm, int len);
IppStatus ippsImag_64fc(const Ipp64fc* pSrc, Ipp64f* pDstIm, int len);
```

Arguments

pSrc	Pointer to the complex source vector.
pDstIm	Pointer to the destination vector with imaginary parts.
len	Number of elements in the vector.

Discussion

The function ippsImag is declared in the ipps.h file. This function returns the imaginary part of a complex vector *pSrc* in the vector *pDstIm*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDstIm* or *pSrc* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

RealToCplx

Returns a complex vector constructed from the real and imaginary parts of two real vectors.

```
IppStatus ippsRealToCplx_16s(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
     Ipp16sc* pDst, int len);
IppStatus ippsRealToCplx_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
     Ipp32fc* pDst, int len);
IppStatus ippsRealToCplx_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
     Ipp64fc* pDst, int len);
```

Arguments

pSrcRe	Pointer to the vector with real parts of complex elements.
pSrcIm	Pointer to the vector with imaginary parts of complex elements.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.

Discussion

The function ippsRealToCplx is declared in the ipps.h file. This function returns a complex vector *pDst* constructed from the real and imaginary parts of the input vectors *pSrcRe* and *pSrcIm*.

If *pSrcRe* is NULL, the real component of the vector is set to zero.

If *pSrcIm* is NULL, the imaginary component of the vector is set to zero.

Note that both pointers can not be NULL.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pDst* pointer is NULL. The pointer *pSrcRe* or *pSrcIm* can be NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

CplxToReal

Returns the real and imaginary parts of a complex vector in two respective vectors.

```
IppStatus ippsCplxToReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe,
     Ipp16s* pDstIm, int len);
IppStatus ippsCplxToReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe,
     Ipp32f* pDstIm, int len);
IppStatus ippsCplxToReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe,
     Ipp64f* pDstIm, int len);
```

Arguments

pSrc	Pointer to the complex vector *pSrc*.
pDstRe	Pointer to the output vector with real parts.
pDstIm	Pointer to the output vector with imaginary parts.
len	Number of elements in the vector.

Discussion

The function ippsCplxToReal is declared in the ipps.h file. This function returns the real and imaginary parts of a complex vector *pSrc* in two vectors *pDstRe* and *pDstIm*.

Return Value

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

Threshold

Performs the threshold operation on the elements of a vector by limiting the element values by `level`.

```
IppStatus ippsThreshold_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
      Ipp16s level, IppCmpOp relOp);
IppStatus ippsThreshold_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
      Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
      Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
       Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
       Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
       Ipp16s level, IppCmpOp relOp);
IppStatus ippsThreshold_16s_I(Ipp16s* pSrcDst, int len,
      Ipp16s level, IppCmpOp relOp);
IppStatus ippsThreshold_32f_I(Ipp32f* pSrcDst, int len,
      Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64f_I(Ipp64f* pSrcDst, int len,
      Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_32fc_I(Ipp32fc* pSrcDst, int len,
      Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64fc_I(Ipp64fc* pSrcDst, int len,
      Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_16sc_I(Ipp16sc* pSrcDst, int len,
      Ipp16s level, IppCmpOp relOp);
```

Arguments

`pSrc`	Pointer to the source vector.
`pDst`	Pointer to the destination vector.

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.
relOp	Values of this argument specify which relational operator to use and whether *level* is an upper or lower bound for the input. The *relOp* must have one of the following values:
ippCmpLess	Specifies the “less than” operator and *level* is a lower bound.
ippCmpGreater	Specifies the “greater than” operator and *level* is an upper bound.

Discussion

The function ippsThreshold is declared in the ipps.h file. This function performs the threshold operation on the vector *pSrc* by limiting each element by the threshold value *level*. Function operation is similar to that of the functions ippsThreshold_LT, ippsThreshold_GT but its interface contains the *relop* argument that specifies the type of the comparison operation to perform.

The in-place flavors of ippsThreshold perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

The *relOp* argument specifies which relational operator to use: “greater than” or “less than,” and determines whether *level* is an upper or lower bound for the input, respectively. The formula for ippsThreshold called with the *ippCmpLess* flag is:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold, the *level* argument is always real. The formula for complex ippsThreshold called with the *ippCmpLess* flag is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the function ippsThreshold. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the "less than" operation (with the *ippCmpLess* flag) the coordinates are rounded to the infinity (+Inf for positive coordinates, and -Inf for negative), and for the "greater than" operation (with the *ippCmpGreater* flag) the coordinates are rounded to 0. See Example 5-13 for more information about using the "complex" function ippsThreshold_16sc_I.

Example 5-12 shows how to use the "real" function ippsTreshold_16s_I.

Return Value

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

Example 5-12 Using the Real ippsThreshold Function

```
IppStatus threshold( void ) {
     Ipp16s x[4] = { -1, 0, 2, 3 };
     IppStatus st = ippsThreshold_16s_I( x, 4, 2, ippCmpLess );
     printf_16s("threshold result =", x, 4, st );
     return st;
}
Output:
    threshold result =  2 2 2 3
```

Example 5-13 Using the Complex ippsThreshold Function

```
IppStatus cmplx_threshold(void) {
     Ipp16sc x[4] = {{2,3}, {3,3}, {4,3}, {4,2}};
     /// level is near to the point {2,3} = 3.6
     /// the point {2,3} is to be replaced
     /// the computed coordinates are {2.2188,3.3282}
     /// the point used is {3,4};
     /// notice that it is the point with the phase,
     /// nearest to the source
     IppStatus st = ippsThreshold_16sc_I(x, 4, 4, ippCmpLess);
     printf_16sc("complex threshold result =", x, 4, st);
     return st;
}
Output:
    complex threshold result = {3, 4} {3, 3} {4, 3} {4, 2}
```

Threshold_LT, Threshold_GT

Performs the threshold operation on the elements of a vector by limiting the element values by `level`.

```
IppStatus ippsThreshold_LT_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s level);
IppStatus ippsThreshold_LT_32s(const Ipp32s* pSrc, Ipp32s* pDst,
     int len, Ipp32s level);
IppStatus ippsThreshold_LT_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level);
```

```
IppStatus ippsThreshold_LT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16s level);
IppStatus ippsThreshold_LT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_LT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s level);
IppStatus ippsThreshold_GT_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.

Discussion

The functions ippsThreshold_LT, ippsThreshold_GT are declared in the ipps.h file. They implement thresholding of the vector *pSrc* by limiting each element by the threshold value *level*. These functions perform the similar operation to the ippsThreshold function but are designed for the fixed type of the compare operation to use: ippsThreshold_LT is for the "less than" comparison, while ippsThreshold_GT is for the "greater than" comparison.

The in-place flavors perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

ippsThreshold_LT. The ippsThreshold_LT function performs the operation “less than”, and *level* is a lower bound for the input. The formula for ippsThreshold_LT is the following:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_LT, the *level* argument is always real.

The formula for complex ippsThreshold_LT is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_GT. The function ippsThreshold_GT performs the operation "greater than" and *level* is an upper bound for the input.

The formula for ippsThreshold_GT is:

$$pDst[n] = \begin{cases} level, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_GT, the *level* argument is always real.

The formula for complex ippsThreshold_GT is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the threshold functions. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the "less than" operation (the ippsThreshold_LT function) the coordinates are rounded to the infinity (+Inf for positive coordinates, and -Inf for negative), and for the "greater than" operation (the ippsThreshold_GT function) the coordinates are rounded to 0.

Return Value

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

Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal

Performs the threshold operation on the elements of a vector by limiting the element values by `level`.

```
IppStatus ippsThreshold_LTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_LTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_LTVal_16s_I(Ipp16s* pSrcDst, int len,
     Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTVal_32f_I(Ipp32f* pSrcDst, int len,
     Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTVal_64f_I(Ipp64f* pSrcDst, int len,
     Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc_I(Ipp16sc* pSrcDst, int len,
     Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_LTVal_32fc_I(Ipp32fc* pSrcDst, int len,
     Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc_I(Ipp64fc* pSrcDst, int len,
     Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_GTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, Ipp16s level, Ipp16s value);
```

```
IppStatus ippsThreshold_GTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int len, Ipp32f level, Ipp32f value);

IppStatus ippsThreshold_GTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level, Ipp64f value);

IppStatus ippsThreshold_GTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, Ipp16s level, Ipp16sc value);

IppStatus ippsThreshold_GTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level, Ipp32fc value);

IppStatus ippsThreshold_GTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level, Ipp64fc value);

IppStatus ippsThreshold_GTVal_16s_I(Ipp16s* pSrcDst, int len,
     Ipp16s level, Ipp16s value);

IppStatus ippsThreshold_GTVal_32f_I(Ipp32f* pSrcDst, int len,
     Ipp32f level, Ipp32f value);

IppStatus ippsThreshold_GTVal_64f_I(Ipp64f* pSrcDst, int len,
     Ipp64f level, Ipp64f value);

IppStatus ippsThreshold_GTVal_16sc_I(Ipp16sc* pSrcDst, int len,
     Ipp16s level, Ipp16sc value);

IppStatus ippsThreshold_GTVal_32fc_I(Ipp32fc* pSrcDst, int len,
     Ipp32f level, Ipp32fc value);

IppStatus ippsThreshold_GTVal_64fc_I(Ipp64fc* pSrcDst, int len,
     Ipp64f level, Ipp64fc value);


IppStatus ippsThreshold_LTValGTVal_16s(const Ipp16s* pSrc,
     Ipp16s* pDst, int len, Ipp16s levelLT, Ipp16s valueLT, Ipp16s
     levelGT, Ipp16s valueGT);

IppStatus ippsThreshold_LTValGTVal_32f(const Ipp32f* pSrc,
     Ipp32f* pDst, int len, Ipp32f levelLT, Ipp32f valueLT, Ipp32f
     levelGT, Ipp32f valueGT);

IppStatus ippsThreshold_LTValGTVal_64f(const Ipp64f* pSrc,
     Ipp64f* pDst, int len, Ipp64f levelLT, Ipp64f valueLT, Ipp64f
     levelGT, Ipp64f valueGT);


IppStatus ippsThreshold_LTValGTVal_16s_I(Ipp16s* pSrcDst, int len,
     Ipp16s levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);

IppStatus ippsThreshold_LTValGTVal_32f_I(Ipp32f* pSrcDst, int len,
     Ipp32f levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);
```

```
IppStatus ippsThreshold_LTValGTVal_64f_I(Ipp64f* pSrcDst, int len,
     Ipp64f levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.
levelLT	Low bound used to limit each element of *pSrc* or *pSrcDst* for the ippsThreshold_LTValGTVal function.
levelGT	Upper bound used to limit each element of *pSrc* or *pSrcDst* for the ippsThreshold_LTValGTVal function.
value	Value to be assigned to vector elements which are “less than” or “greater than” *level*.
valueLT	Value to be assigned to vector elements which are less than *levelLT* for the ippsThreshold_LTValGTVal function.
valueGT	Value to be assigned to vector elements which are greater than *levelGT* for the ippsThreshold_LTValGTVal function.

Discussion

These functions are declared in the ipps.h file. They perform the threshold operation on the vector *pSrc* by limiting each element by the threshold value.

The in-place flavors of the function perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value.

ippsThreshold_LTVal. The ippsThreshold_LTVal function performs the operation “less than” and *level* is a lower bound for the input. The vector elements less than *level* are set to *value*.

The formula for ippsThreshold_LTVal is:

$$pDst[n] = \begin{cases} value, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_LTVal, the *level* argument is always real.

The formula for complex ippsThreshold_LTVal is:

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_GTVal. The ippsThreshold_GTVal function performs the operation "greater than" and *level* is an upper bound for the input. The vector elements greater than *level* are set to *value*.

The formula for ippsThreshold_GtVal is:

$$pDst[n] = \begin{cases} value, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function ippsThreshold_GTVal, the *level* argument is always real.

The formula for complex ippsThreshold_GTVal is:

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_LTValGTVal. The ippsThreshold_LTValGTVal function checks both the "less than" and "greater than" conditions. Argument *levelLT* is a lower bound and argument *levelGT* is an upper bound for the input. The source vector elements less than *levelLT* are set to *valueLT*, and the source vector elements greater than *levelGT* are set to *valueGT*. The value of *levelLT* should be less than or equal to *levelGT*.

The formula for ippsThreshold_LTValGTVal is:

$$pDst[n] = \begin{cases} valueLT, & pSrc[n] < levelLT \\ pSrc[n], & levelLT \le pSrc[n] \le levelGT \\ valueGT, & pSrc[n] > levelGT \end{cases}$$

Application Notes

For all complex versions, *level* must be positive and represent a magnitude.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsThresholdErr	Indicates an error when *levelLT* is greater than *levelGT*.
ippStsThreshNegLevelErr	Indicates an error when *level* for the complex function version is negative.

Threshold_LTInv

Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.

```
IppStatus ippsThreshold_LTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32f_I(Ipp32f* pSrcDst, int len,
     Ipp32f level);
IppStatus ippsThreshold_LTInv_64f_I(Ipp64f* pSrcDst, int len,
     Ipp64f level);
```

```
IppStatus ippsThreshold_LTInv_32fc_I(Ipp32fc* pSrcDst, int len,
     Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc_I(Ipp64fc* pSrcDst, int len,
     Ipp64f level);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real and positive.

Discussion

The function ippsThreshold_LTInv is declared in the ipps.h file. This function computes the inverse of elements of the vector *pSrc* and stores the result in *pDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The in-place flavors of ippsThreshold_LTInv compute the inverse of elements of the vector *pSrcDst* and store the result in *pSrcDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The threshold operation is performed to avoid division by zero. Since *level* represents a magnitude, it is always real and must be positive. The formula for ippsThreshold_LTInv is the following:

$$pDst[n] = \begin{cases} \frac{1}{level}, & abs(pSrc[n]) = 0 \\ \frac{abs(pSrc[n])}{pSrc[n] \cdot level}, & 0 < abs(pSrc[n]) < level \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

If the function encounters zero-valued vector elements and *level* is also 0, the output value is set to Inf (infinity), but operation execution is not aborted:

$$pDst[n] = \begin{cases} Inf, & pSrc[n] = 0 \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

Example 5-14 shows how to use the function ippsThreshold_LTInv_32f_I.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsThreshNegLevelErr	Indicates an error when *level* is negative.
ippStsInvZero	Indicates a warning when *level* and a vector element are equal to 0. Operation execution is not aborted. The value of the destination vector element is Inf.

Example 5-14 Using the ippsThreshold_LTInv Function

```
IppStatus invThreshold(void) {
     Ipp32f x[4] = {-1, 0, 2, 3};
     IppStatus st = ippsThreshold_LTInv_32f_I(x, 4, 0);
     printf_32f("inv threshold =", x, 4, st);
     return st;
}
Output:
   -- warning 4, INF result. Zero value met by invThreshold with zero level
    inv threshold = -1.000000 1.#INF00 0.500000 0.333333
```

CartToPolar

Converts the elements of a complex vector to polar coordinate form.

```
IppStatus ippsCartToPolar_32fc(const Ipp32fc* pSrc, Ipp32f* pDstMagn,
      Ipp32f* pDstPhase, int len);
IppStatus ippsCartToPolar_64fc(const Ipp64fc* pSrc, Ipp64f* pDstMagn,
      Ipp64f* pDstPhase, int len);
IppStatus ippsCartToPolar_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
      Ipp32f* pDstMagn, Ipp32f* pDstPhase, int len);
IppStatus ippsCartToPolar_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
      Ipp64f* pDstMagn, Ipp64f* pDstPhase, int len);
```

Arguments

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components of Cartesian X,Y pairs.
pSrcIm	Pointer to the source vector which stores the imaginary components of Cartesian X,Y pairs.
pDstMagn	Pointer to the vector which stores the magnitude (radius) component of the elements of the vector *pSrc*.
pDstPhase	Pointer to the vector which stores the phase (angle) component of the elements of the vector *pSrc* in radians. Phase values are in the range $(-\pi,\pi]$.
len	Number of elements in the vector.

Discussion

The function `ippsCartToPolar` is declared in the `ipps.h` file. This function converts the elements of a complex input vector *pSrc* or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*,

respectively, to polar coordinate form, and stores the magnitude (radius) component of each element in the vector *pDstMagn* and the phase (angle) component of each element in the vector *pDstPhase*.

Example 5-15 verifies that points are lying in the unit radius circle.

Return Value

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

Example 5-15 Using the ippsCartToPolar Function

```
IppStatus cart2polar( void ) {
     Ipp64f cart[6], magn[4], phase[4];
     int n;
     for (n=0; n<6; ++n) cart[n] = sin(IPP_2PI * n / 8);
    IppStatus st = ippsCartToPolar_64f( cart, cart+2, magn, phase, 4 );
     printf_64f( "magn =", magn, 4, st );
     return st;
}

Output:
    magn =  1.000000 1.000000 1.000000 1.000000
```

PolarToCart

Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.

```
IppStatus ippsPolarToCart_32fc(const Ipp32f* pSrcMagn,
     const Ipp32f* pSrcPhase, Ipp32fc* pDst, int len);
IppStatus ippsPolarToCart_64fc(const Ipp64f* pSrcMagn,
     const Ipp64f* pSrcPhase, Ipp64fc* pDst, int len);

IppStatus ippsPolarToCart_32f(const Ipp32f* pSrcMagn, const
     Ipp32f* pSrcPhase, Ipp32f* pDstRe, Ipp32f* pDstIm, int len);
IppStatus ippsPolarToCart_64f(const Ipp64f* pSrcMagn, const
     Ipp64f* pSrcPhase, Ipp64f* pDstRe, Ipp64f* pDstIm, int len);
```

Arguments

pSrcMagn	Pointer to the source vector which stores the magnitude (radius) components of the elements in polar coordinate form.
pSrcPhase	Pointer to the vector which stores the phase (angle) components of the elements in polar coordinate form in radians. Phase values are in the range $(-\pi,\pi]$.
pDst	Pointer to the resulting vector which stores the complex pairs in Cartesian coordinates (X + iY).
pDstRe	Pointer to the resulting vector which stores the real components of Cartesian X,Y pairs.
pDstIm	Pointer to the resulting vector which stores the imaginary components of Cartesian X,Y pairs.
len	Number of elements in the vectors.

Discussion

The function ippsPolarToCart is declared in the ipps.h file. This function converts the polar form magnitude/phase pairs stored in the input vectors *pSrcMagn* and *pSrcPhase* into a complex vector and stores the results in the vector *pDst*, or stores the real components of the result in the vector *pDstRe* and the imaginary components in the vector *pDstIm*.

Return Value

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

MaxOrder

Computes the maximum order of a vector.

```
IppStatus ippsMaxOrder_16s(const Ipp16s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32s(const Ipp32s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32f(const Ipp32f* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_64f(const Ipp64f* pSrc, int len, int* pOrder);
```

Arguments

pSrc	Pointer to the source vector.
len	Number of elements in the vector.
pOrder	Pointer to the result value.

Discussion

The function ippsMaxOrder is declared in the ipps.h file. This function finds the maximum binary number in elements of the exponent vector *pSrc*, and stores the result in *pOrder*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pOrder* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsNanArg	Indicates a warning when NaN is encountered in the input data vector.

Preemphasize

Computes preemphasis of a single precision real signal in-place.

```
IppStatus ippsPreemphasize_16s(Ipp16s* pSrcDst, int len, Ipp32f val);
IppStatus ippsPreemphasize_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
```

Arguments

pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
val	Multiplier factor used in the difference signal preemphasis equation.

Discussion

The in-place function ippsPreemphasize is declared in the ipps.h file. This function computes preemphasis of a real signal *pSrcDst*. The computation is performed according to the difference signal preemphasis equation:

$y(n) = x(n) - val \cdot x(n - 1)$,

where $y(n)$ is the preemphasized output, $x(n)$ is the input, and *val* is the multiplier factor.

Note that usually *val* = 0.95 for speech signals.

Return Value

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

Flip

Reverses the order of elements in a vector.

```
IppStatus ippsFlip_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsFlip_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsFlip_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsFlip_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsFlip_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsFlip_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsFlip_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsFlip_64f_I(Ipp64f* pSrcDst, int len);
```

Arguments

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

Discussion

The function ippsFlip is declared in the ipps.h file. This function stores the elements of a source vector *pSrc* to a destination vector *pDst* in reverse order according to the following formula:

pDst[*n*] = *pSrc*[*len*-*n*-1], *n*=0..*len*-1

Return Value

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

FindNearestOne

Finds an element of the table which is closest to the specified value.

```
IppStatus ippsFindNearestOne_16u(Ipp16u inpVal, Ipp16u* pOutVal,
     int* pOutIndex, const Ipp16u *pTable, int tblLen);
```

Arguments

inpVal	Reference value.
pOutVal	Pointer to the output value.
pOutIndx	Pointer to the output index.
pTable	Pointer to the table for searching.
tblLen	Number of elements in the table.

Discussion

The function ippsFindNearestOne is declared in the ipps.h file. This function searches through the table *pTable* for an element which is closest to the specified reference value *inpVal*. The resulting element and its index are stored in *pOutVal* and *pOutIndex*, respectively.
The table elements should satisfy the condition $pTable[i] \le pTable[i+1]$.
The function uses the following distance criterion for determining the closest element: $\min(|inpVal - pTable[i]|)$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *tblLen* is less than or equal to 0.

FindNearest

Finds table elements that are closest to the elements of the specified vector.

```
IppStatus ippsFindNearest_16u(const Ipp16u* pVals, Ipp16u* pOutVals,
     int* pOutIndexes, int len, const Ipp16u *pTable, int tblLen);
```

Arguments

pVals	Pointer to the vector containing reference values.
pOutVals	Pointer to the output vector.
pOutIndexes	Pointer to the array that stores output indexes.
len	Number of elements in the input vector.
pTable	Pointer to the table for searching.
tblLen	Number of elements in the table.

Discussion

The function ippsFindNearest is declared in the ipps.h file. This function searches through the table *pTable* for elements which are closest to the reference elements of the input vector *pVals*. The resulting elements and their indexes are stored in *pOutVals* and *pOutIndexes*, respectively.
The table elements should satisfy the condition *pTable*[*i*] ≤*pTable*[*i*+1].
The function uses the following distance criterion for determining the table element closest to *pVal*[*k*] :
min(|*pVals*[*k*]-*pTable*[i]|).

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *tblLen* or *len* is less than or equal to 0.

Viterbi Decoder Functions

This section describes the functions that perform operations of Viterbi decoding in the V34 receiver. The encoding is performed in accordance with the algorithm that is described in the section 9.6.3 of the ITU-T Recommendation V.34 (see [ITUV34]).

GetVarPointDV

Fills the array with the information about points that are closest to the received point.

```
IppStatus ippsGetVarPointDV_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
      Ipp16sc* pVariantPoint, const Ipp8u* pLabel, int state);
```

Arguments

pSrc	Pointer to the reference point in format 9:7
pDst	Pointer to the closest to the reference point left and bottom complex point in format 9:7
pVariantPoint	Pointer to the array where the point information is stored
pLabel	Pointer to the table that stores the labels.
state	Number of states of a convolution coder.

Discussion

The function ippsGetVarPointDV is declared in the ipps.h file. This function fills the specified array *pVariantPoint* with the information about 2D complex points that are closest to the reference point *pSrc*. This information includes the corresponding labels from the offset table and calculated errors. The number of possible states may be 16, 32, and 64. The number of points in the array depends on the number of states: if *state* =16 , then 4 points will be referenced in the array, if *state* =32 or 64, then 8 points will be referenced.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.

CalcStatesDV

Calculates the states of the Viterbi decoder.

```
IppStatus ippsCalcStatesDV_16sc(const Ipp16u* pathError,
     const Ipp8u* pNextState, Ipp16u* pBranchError,
     const Ipp16s* pCurrentSubsetPoint, Ipp16s* pPathTable,
     int state, int presentIndex);
```

Arguments

pPathError	Pointer to the table of path error metrics.
pNextState	Pointer to the next state table.
pBranchError	Pointer to the branch error table.
pCurrentSubsetPoint	Pointer to the current 4D subset.
pPathTable	Pointer to the Viterbi path table.
state	Number of states of a convolutional encoder.
presentIndex	Start index in Viterbi path table.

Discussion

The function ippsCalcStatesDV is declared in the ipps.h file. This function computes the possible states of the Viterbi decoder and fills the table of the accumulated errors *pPathError* and the working Viterbi path table *pPathTable*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.

BuildSymblTableDV4D

Fills the array with the information about possible 4D symbols.

```
IppStatus ippsBuildSymblTableDV4D_16sc(const Ipp16sc* pVariantPoint,
     Ipp16sc* pCurrentSubsetPoint, int state, int bitInversion);
```

Arguments

pVariantPoint	Pointer to the array of possible 2D points.
pCurrentSubsetPoint	Pointer to the array with information about possible 4D symbols.
state	Number of states of a convolutional encoder.
bitInversion	Bit inversion.

Discussion

The function ippsBuildSymblTableDV4D is declared in the ipps.h file. This function fills the array *pCurrentSubsetPoint* with information about possible 4D symbols.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.

UpdatePathMetricsDV

Searches for the state with the minimum path metric.

```
IppStatus ippsUpdatePathMetricsDV_16u(Ipp16u* pBranchError, Ipp16u*
     pMinPathError, Ipp8u* pMinSost, Ipp16u* pPathError, int state);
```

Arguments

pBranchError	Pointer to the branch error table.
pMinPathError	Pointer to the current minimum path error metric.
pMinSost	Pointer to the number of states with the minimum metric.
pPathError	Pointer to the table of accumulated path errors.
state	Number of states of a convolutional encoder.

Discussion

The function ippsBuildSymblTableDV4D is declared in the ipps.h file. This function searches for the state with the minimum path error metric and stores its number in the *pMinSost*. For all states, the minimum metric is subtracted from the path metric. The branch errors for all states are set to be infinitely large as is required for the next step of the Viterbi decoding.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.

Windowing Functions

This chapter describes several of the windowing functions commonly used in signal processing. A window is a mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

Understanding Window Functions

The Intel IPP provides the following functions to generate window samples:

- Bartlett windowing function
- Blackman family of windowing functions
- Hamming windowing function
- Hann windowing function
- Kaiser windowing function

These functions generate the window samples and multiply them into an existing signal. To obtain the window samples themselves, initialize the vector argument to the unity vector before calling the window function.

If you want to multiply different frames of a signal by the same window multiple times, it is better to first calculate the window by calling one of the windowing functions (`ippsWinHann`, for example) on a vector with all elements set to 1.0. Then use one of the vector multiplication functions (`ippsMul`, for example) to multiply the window into the signal each time a new set of input samples is available. This avoids repeatedly calculating the window samples. This is illustrated in Example 5-16.

Example 5-16 Window and FFT Many Frames of a Signal

```
void multiFrameWin( void ) {
     Ipp32f win[LEN], x[LEN], X[LEN];
     IppsFFTSpec_R_32f* ctx;
     ippsSet_32f( 1, win, LEN );
     ippsWinHann_32f_I( win, LEN );
     /// ... initialize FFT context
     while(1 ){
        /// ... get x signal
        ///
        ippsMul_32f_I( win, x, LEN );
        ippsFFTFwd_RToPack_32f( x, X, ctx, 0 );
     }
}
```

Related Topics

For more information on windowing, see: [Jac89], section 7.3, *Windows in Spectrum Analysis*; [Jac89], section 9.1, *Window-Function Technique*; and [Mit93], section 16-2, *Fourier Analysis of Finite-Time Signals*.
For more information on these references, see also the Bibliography at the end of this manual.

WinBartlett

Multiplies a vector by a Bartlett windowing function.

```
IppStatus ippsWinBartlett_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len);
```

```
IppStatus ippsWinBartlett_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len);
IppStatus ippsWinBartlett_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len);
IppStatus ippsWinBartlett_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len);
IppStatus ippsWinBartlett_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len);
IppStatus ippsWinBartlett_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len);
IppStatus ippsWinBartlett_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBartlett_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBartlett_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBartlett_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBartlett_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBartlett_64fc_I(Ipp64fc* pSrcDst, int len);
```

Arguments

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

Discussion

The function `ippsWinBartlett` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Bartlett (triangle) window, and stores the result in *pDst*.

The in-place flavors of `ippsWinBartlett` multiply the *pSrcDst* by the Bartlett (triangle) window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window.

The Bartlett window is defined as follows:

$$w_{bartlett}(n) = \begin{cases} \frac{2n}{len-1}, & 0 \le n \le \frac{len-1}{2} \\ 2 - \frac{2n}{len-1}, & \frac{len-1}{2} < n \le len-1 \end{cases}$$

Example 5-17 shows how to use the function ippsWinBartlett_32f_I.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 3.

Example 5-17 Using the ippsWinBartlett Function

```
void bartlett(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBartlett_32f_I(x, 8);
      printf_32f("bartlett (half) =", x, 4, ippStsNoErr);
}
Output:
    bartlett (half) =  0.000000 0.285714 0.571429 0.857143
Matlab* Analog:
    >> b = bartlett(8); b(1:4)'
```

WinBlackman

Multiplies a vector by a Blackman windowing function.

```
IppStatus ippsWinBlackmanQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, int alphaQ15);
```

```
IppStatus ippsWinBlackmanQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, int alphaQ15);

IppStatus ippsWinBlackman_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, float alpha);

IppStatus ippsWinBlackman_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, float alpha);

IppStatus ippsWinBlackman_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len, float alpha);

IppStatus ippsWinBlackman_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, float alpha);

IppStatus ippsWinBlackman_64f(Ipp64f* pSrc, Ipp64f* pDst, int len,
     double alpha);

IppStatus ippsWinBlackman_64fc(Ipp64fc* pSrc, Ipp64fc* pDst, int len,
     double alpha);

IppStatus ippsWinBlackmanQ15_16s_I(Ipp16s* pSrcDst, int len,
     int alphaQ15);

IppStatus ippsWinBlackmanQ15_16s_ISfs(Ipp16s* pSrcDst, int len,
     int alphaQ15, int scaleFactor);

IppStatus ippsWinBlackmanQ15_16sc_I(Ipp16sc* pSrcDst, int len,
     int alphaQ15);

IppStatus ippsWinBlackman_16s_I(Ipp16s* pSrcDst, int len,
     float alpha);

IppStatus ippsWinBlackman_16sc_I(Ipp16sc* pSrcDst, int len,
     float alpha);

IppStatus ippsWinBlackman_32f_I(Ipp32f* pSrcDst, int len,
     float alpha);

IppStatus ippsWinBlackman_32fc_I(Ipp32fc* pSrcDst, int len,
     float alpha);

IppStatus ippsWinBlackman_64f_I(Ipp64f* pSrcDst, int len,
     double alpha);

IppStatus ippsWinBlackman_64fc_I(Ipp64fc* pSrcDst, int len,
     double alpha);

IppStatus ippsWinBlackmanStd_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len);

IppStatus ippsWinBlackmanStd_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len);
```

```
IppStatus ippsWinBlackmanStd_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len);
IppStatus ippsWinBlackmanStd_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len);
IppStatus ippsWinBlackmanStd_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len);
IppStatus ippsWinBlackmanStd_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len);
IppStatus ippsWinBlackmanStd_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len);
IppStatus ippsWinBlackmanOpt_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64fc_I(Ipp64fc* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
alpha	Adjustable parameter associated with the Blackman windowing equation.
alphaQ15	Scaled version of *alpha*. The *scaleFactor* value is 15.
len	Number of elements in the vector.
scaleFactor	Refer to Integer Scaling.

Discussion

The ippsWinBlackman family of functions are declared in the ipps.h file. These functions multiply the vector *pSrc* by the Blackman window, and store the result in *pDst*.

The in-place flavors of ippsWinBlackman multiply the vector *pSrcDst* by the Blackman window, and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The functions for the Blackman family of windows are defined below.

ippsWinBlackman. The function ippsWinBlackman allows the application to specify *alpha*. The Blackman window is defined as follows:

$$w_{blackman}(n) = \frac{alpha+1}{2} - 0.5\cos\left(\frac{2\pi n}{len-1}\right) - \frac{alpha}{2}\cos\left(\frac{4\pi n}{len-1}\right)$$

ippsWinBlackmanQ15. The function ippsWinBlackmanQ15 multiplies a vector by a Blackman window with *alphaQ15* scaled according to the factor 15.

ippsWinBlackmanStd. The standard Blackman window is provided by the function ippsWinBlackmanStd, which simply multiplies a vector by a Blackman window with the standard value of *alpha* shown below:

alpha = -0.16

ippsWinBlackmanOpt. The function ippsWinBlackmanOpt provides a modified window that has a 30 dB/octave roll-off by multiplying a vector by a Blackman window with the optimal value of *alpha* shown below:

$$alpha = -\frac{0.5}{1 + \cos\frac{2\pi}{len - 1}}$$

The minimum *len* is equal to 4. For large *len*, the optimal *alpha* converges asymptotically to the asymptotic *alpha*; the application can use the asymptotic value of *alpha* shown below:

alpha = -0.25

Example 5-18 shows how to use the function ippsWinBlackmanStd_32f_I

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 4 for the function ippsWinBlackmanOpt and less than 3 for all other functions of the family.

Example 5-18 Using the ippsWinBlackmanStd Function

```
void blackman(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBlackmanStd_32f_I(x, 8);
      printf_32f("blackman (half) =", x, 4, ippStsNoErr);
}
Output:
    blackman(half) =  0.000000 0.090453 0.459183 0.920364
Matlab* Analog:
    >> b = blackman(8)'; b(1:4)
```

WinHamming

Multiplies a vector by a Hamming windowing function.

```
IppStatus ippsWinHamming_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHamming_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHamming_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHamming_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHamming_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHamming_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHamming_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHamming_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHamming_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHamming_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHamming_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHamming_64fc_I(Ipp64fc* pSrcDst, int len);
```

Arguments

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

Discussion

The function ippsWinHamming is declared in the ipps.h file. This function multiplies the vector *pSrc* by the Hamming window and stores the result in *pDst*.

The in-place flavors of ippsWinHamming multiply the vector *pSrcDst* by the Hamming window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hamming window is defined as follows:

$$w_{hamming}(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{len - 1}\right)$$

Example 5-19 shows how to use the function `ippsWinHamming_32f_I`

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc`*, *`pDst`*, *`or pSrcDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than 3.

Example 5-19 Using the ippsWinHamming Function

```
void hamming(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinHamming_32f_I(x, 8);
      printf_32f("hamming(half) =", x, 4, ippStsNoErr);
}
Output:
    hamming(half) =  0.080000 0.253195 0.642360 0.954446
Matlab* Analog:
    >> b = hamming(8); b(1:4)'
```

WinHann

Multiplies a vector by a Hann windowing function.

```
IppStatus ippsWinHann_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHann_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHann_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHann_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHann_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHann_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHann_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHann_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHann_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHann_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHann_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHann_64fc_I(Ipp64fc* pSrcDst, int len);
```

Arguments

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

Discussion

The function ippsWinHann is declared in the ipps.h file. This function multiplies the vector *pSrc* by the Hann window and stores the result in *pDst*.

The in-place flavors of ippsWinHann multiply the vector *pSrcDst* by the Hann window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hann window is defined as follows:

$$w_{hann}(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{len - 1}\right)$$

Example 5-20 shows how to use the function `ippsWinHann_32f_I`

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc`*, *`pDst`*, *`or pSrcDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than 3.

Example 5-20 Using the ippsWinHann Function

```
void hann(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinHann_32f_I(x, 8);
      printf_32f("hann(half) =", x, 4, ippStsNoErr);
}
Output:
    hann(half) =  0.000000 0.188255 0.611260 0.950484
Matlab* Analog:
    >> N = 8; n = 0:N-1; 0.5*(1-cos(2*pi*n/(N-1)))
```

WinKaiser

Multiplies a vector by a Kaiser windowing function.

```
IppStatus ippsWinKaiser_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     float alpha);
IppStatus ippsWinKaiser_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
     float alpha);
IppStatus ippsWinKaiser_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
     float alpha);
IppStatus ippsWinKaiser_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, float alpha);
IppStatus ippsWinKaiser_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int len, float alpha);
IppStatus ippsWinKaiser_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int len, float alpha);
IppStatus ippsWinKaiserQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, int alphaQ15);
IppStatus ippsWinKaiser_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32f_I(Ipp32f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_64f_I(Ipp64f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_64fc_I(Ipp64fc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiserQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);
```

Arguments

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
alpha	Adjustable parameter associated with the Kaiser windowing equation.
alphaQ15	Scaled version of *alpha*. The *scaleFactor* value is 15.
len	Number of elements in the vector.

Discussion

The function ippsWinKaiser is declared in the ipps.h file. This function multiplies the vector *pSrc* by the Kaiser window, and stores the result in *pDst*.

The in-place flavors of ippsWinKaiser multiply the vector *pSrcDst* by the Kaiser window and store the result in *pSrcDst*.

ippsWinKaiser. The function ippsWinKaiser allows the application to specify *alpha*. The function multiplies both real and imaginary parts of the complex vector by the same window. The Kaiser family of windows are defined as follows:

$$w_{kaiser}(n) = \frac{I_0\left(alpha\sqrt{\left(\frac{len-1}{2}\right)^2-\left(n-\left(\frac{len-1}{2}\right)\right)^2}\right)}{I_0\left(alpha\left(\frac{len-1}{2}\right)\right)}$$

Here $I_0()$ is the modified zero-order Bessel function of the first kind.

ippsWinKaiserQ15. The function ippsWinKaiserQ15 multiplies a vector by a Kaiser window with *alphaQ15* scaled according to the factor 15.

Example 5-21 shows how to use the function ippsWinKaiser_32f_I

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst*, *pSrc*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 1.
ippStsHugeWinErr	Indicates an error when the Kaiser window is too big.

Example 5-21 Using the ippsWinKaiser Function

```
void kaiser(void) {
     Ipp32f x[8];
     IppStatus st;
     ippsSet_32f(1, x, 8);
     st = ippsWinKaiser_32f_I( x, 8, 1.0f );
     printf_32f("kaiser(half) =", x, 4, ippStsNoErr);
}
Output:
   kaiser(half) =  0.135534 0.429046 0.755146 0.970290
Matlab* Analog:
    >> kaiser(8,7/2)'
```

Statistical Functions

This section describes the Intel IPP functions that compute the vector measure values: maximum, minimum, mean, and standard deviation.

Sum

Computes the sum of the elements of a vector.

```
IppStatus ippsSum_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum,
     IppHintAlgorithm hint);
IppStatus ippsSum_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pSum,
     IppHintAlgorithm hint);
```

```
IppStatus ippsSum_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSum_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pSum);
IppStatus ippsSum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pSum,
     int scaleFactor);
IppStatus ippsSum_32s_Sfs(const Ipp32s* pSrc, int len, Ipp32s* pSum,
     int scaleFactor);
IppStatus ippsSum_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pSum,
     int scaleFactor);
IppStatus ippsSum_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pSum,
     int scaleFactor);
IppStatus ippsSum_16sc32sc_Sfs(const Ipp16sc* pSrc, int len, Ipp32sc* pSum,
     int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pSum	Pointer to the output result.
len	Number of elements in the vector.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsSum is declared in the ipps.h file. This function computes the sum of the elements of the vector *pSrc* and stores the result in *pSum*.

The sum of the elements of *pSrc* is defined by the formula:

$$sum = \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

When computing the sum of integer numbers, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. The scaling is performed in accordance with the *scaleFactor* value.

Example 5-22 shows how to use the function ippsSum.

Return Value

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

Example 5-22 Using the ippsSum Function

```
void sum(void) {
      Ipp16s x[4] = {-32768, 32767, 32767, 32767}, sm;
      ippsSum_16s_Sfs(x, 4, &sm, 1);
      printf_16s("sum =", &sm, 1, ippStsNoErr);
}
Output:
    sum = 32766
Matlab* Analog:
    >> x = [-32768, 32767, 32767, 32767]; sum(x)/2
```

Max

Returns the maximum value of a vector.

```
IppStatus ippsMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax);
IppStatus ippsMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax);
IppStatus ippsMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax);
```

Arguments

pSrc	Pointer to the source vector.
pMax	Pointer to the output result.
len	Number of elements in the vector

Discussion

The function ippsMax is declared in the ipps.h file. This function returns the maximum value of the input vector *pSrc*, and stores the result in *pMax*.

Example 5-23 shows how to use the function ippsMax_32f.

Return Value

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

MaxIndx

Returns the maximum value of a vector and the index of the maximum element.

```
IppStatus ippsMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax,
     int* pIndx);
IppStatus ippsMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax,
     int* pIndx);
IppStatus ippsMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax,
     int* pIndx);
IppStatus ippsMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax,
     int* pIndx);
```

Arguments

pSrc	Pointer to the source vector.

pMax	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the maximum element.

Discussion

The function ippsMaxIndx is declared in the ipps.h file. This function returns the maximum value of the input vector *pSrc*, and stores the result in *pMax*. If *pIndx* is not a NULL pointer, the function returns the index of the maximum element and stores it in *pIndx*. If there are several equal maximum elements, the first index from the beginning is returned.

Return Value

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

Min

Returns the minimum value of a vector.

```
IppStatus ippsMin_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin);
IppStatus ippsMin_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin);
IppStatus ippsMin_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin);
```

Arguments

pSrc	Pointer to the source vector.
pMin	Pointer to the output result.
len	Number of elements in the vector.

Discussion

The function ippsMin is declared in the ipps.h file. This function returns the minimum value of the input vector *pSrc*, and stores the result in *pMin*.

Example 5-23 shows how to use the function ippsMin_32f.

Return Value

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

MinIndx

Returns the minimum value of a vector and the index of the minimum element.

```
IppStatus ippsMinIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin,
     int* pIndx);
IppStatus ippsMinIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin,
     int* pIndx);
IppStatus ippsMinIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin,
     int* pIndx);
```

Arguments

pSrc	Pointer to the source vector.
pMin	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the minimum element.

Discussion

The function ippsMinIndx is declared in the ipps.h file. This function returns the minimum value of the input vector *pSrc* and stores the result in *pMin*. If *pIndx* is not a NULL pointer, the function returns the index of the minimum element and stores it in *pIndx*. If there are several equal minimum elements, the first index from the beginning is returned.

Return Value

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

Example 5-23 Using the ippsMin and ippsMax Functions

```
void minmax(void) {
      Ipp32f *x = ippsMalloc_32f(1000), minmax[2];
      int i;
      for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsMin_32f(x, 1000, &minmax[0]);
      ippsMax_32f(x, 1000, &minmax[1]);
      printf_32f("min max =", minmax, 2, ippStsNoErr);
      ippsFree(x);
}
Output:
    min max = 0.000855 0.999695
Matlab* Analog:
    >> x = rand(1,1000); min(x), max(x)
```

MinMax

Returns the maximum and minimum values of a vector.

```
IppStatus ippsMinMax_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin,
      Ipp8u* pMax);
IppStatus ippsMinMax_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin,
      Ipp16u* pMax);
IppStatus ippsMinMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin,
      Ipp16s* pMax);
IppStatus ippsMinMax_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin,
      Ipp32u* pMax);
IppStatus ippsMinMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin,
      Ipp32s* pMax);
IppStatus ippsMinMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin,
      Ipp32f* pMax);
IppStatus ippsMinMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin,
      Ipp64f* pMax);
```

Arguments

pSrc	Pointer to the source vector.
pMin	Pointer to the minimum value.
pMax	Pointer to the maximum value.
len	Number of elements in the vector.

Discussion

The function ippsMinMax is declared in the ipps.h file. This function returns the minimum and maximum values of the input vector *pSrc*, and stores the results in *pMin* and *pMax,* respectively.

Return Value

ippStsNoErr	Indicates no error.

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

MinMaxIndx

Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.

```
IppStatus ippsMinMaxIndx_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin,
      int* pMinIndx, Ipp8u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin,
      int* pMinIndx, Ipp16u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin,
      int* pMinIndx, Ipp16s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin,
      int* pMinIndx, Ipp32u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin,
      int* pMinIndx, Ipp32s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin,
      int* pMinIndx, Ipp32f* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin,
      int* pMinIndx, Ipp64f* pMax, int* pMaxIndx);
```

Arguments

pSrc	Pointer to the source vector.
pMin	Pointer to the minimum value.
pMax	Pointer to the maximum value.
len	Number of elements in the vector.
pMinIndx	Pointer to the index value of the minimum element.
pMaxIndx	Pointer to the index value of the maximum element.

Discussion

The function ippsMinMaxIndx is declared in the ipps.h file. This function returns the minimum and maximum values of the input vector *pSrc* and stores the result in *pMin* and *pMax,* respectively. The function also returns the indexes of the minimum and maximum elements and stores them in *pMinIndx* and *pMaxIndx,* respectively. If there are several equal minimum or maximum elements, the first index from the beginning is returned.

Return Value

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

Mean

Computes the mean value of a vector.

```
IppStatus ippsMean_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean,
     IppHintAlgorithm hint);
IppStatus ippsMean_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pMean,
     IppHintAlgorithm hint);
IppStatus ippsMean_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean);
IppStatus ippsMean_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pMean);
IppStatus ippsMean_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean,
     int scaleFactor);
IppStatus ippsMean_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pMean,
     int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.

pMean	Pointer to the output result.
len	Number of elements in the vector
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsMean is declared in the ipps.h file. This function computes the mean (average) of the vector *pSrc*, and stores the result in *pMean*. The mean of *pSrc* is defined by the formula:

$$mean = \frac{1}{len} \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Example 5-24 shows how to use the function ippsMean_32f

Return Value

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

Example 5-24 Using the ippsMean Function

```
void mean(void) {
      Ipp32f *x = ippsMalloc_32f(1000), mean;
      int i;
      for(i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsMean_32f(x, 1000, &mean, ippAlgHintFast);
      printf_32f("mean =", &mean, 1, ippStsNoErr);
      ippsFree(x);
}
Output:
    mean = 0.492591
Matlab* Analog:
    >> x = rand(1,1000); mean(x)
```

StdDev

Computes the standard deviation value of a vector.

```
IppStatus ippsStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pStdDev,
     IppHintAlgorithm hint);
IppStatus ippsStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pStdDev);
IppStatus ippsStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len,
     Ipp32s* pStdDev, int scaleFactor);
IppStatus ippsStdDev_16s_Sfs(const Ipp16s* pSrc, int len,
     Ipp16s* pStdDev, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pStdDev	Pointer to the output result.

len	Number of elements in the vector
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsStdDev is declared in the ipps.h file. This function computes the standard deviation of the input vector *pSrc*, and stores the result in *pStdDev*. The vector length can not be less than 2. The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$stdev = \sqrt{\frac{\sum_{n=0}^{len-1} (pSrc[n] - mean(pSrc))^2}{len - 1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Example 5-25 shows how to use the function ippsStdDev_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pStdDev* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 1.

Example 5-25 Using the ippsStdDev Function

```
void stdev(void) {
      Ipp32f *x = ippsMalloc_32f(1000), stdev;
      int i;
      for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsStdDev_32f(x, 1000, &stdev, ippAlgHintFast);
      printf_32f("stdev =", &stdev, 1, ippStsNoErr);
      ippsFree(x);
}
Output:
    stdev = 0.286813
Matlab* Analog:
    >> x = rand(1,1000); std(x)
```

Norm

Computes the C, L1, or L2 norm of a vector.

```
IppStatus ippsNorm_Inf_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_32fc32f(const Ipp32fc* pSrc, int len,
     Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64fc64f(const Ipp64fc* pSrc, int len,
     Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32s_Sfs(const Ipp16s* pSrc, int len,
     Ipp32s* pNorm, int scaleFactor);

IppStatus ippsNorm_L1_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
```

```
IppStatus ippsNorm_L1_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_32fc64f(const Ipp32fc* pSrc, int len,
     Ipp64f* pNorm);
IppStatus ippsNorm_L1_64fc64f(const Ipp64fc* pSrc, int len,
     Ipp64f* pNorm);
IppStatus ippsNorm_L1_16s32s_Sfs(const Ipp16s* pSrc, int len,
     Ipp32s* pNorm, int scaleFactor);

IppStatus ippsNorm_L2_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_32fc64f(const Ipp32fc* pSrc, int len,
     Ipp64f* pNorm);
IppStatus ippsNorm_L2_64fc64f(const Ipp64fc* pSrc, int len,
     Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32s_Sfs(const Ipp16s* pSrc, int len,
     Ipp32s* pNorm, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector *pSrc*.
pNorm	Pointer to the output result.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsNorm is declared in the ipps.h file. This function computes the C, L1, or L2 norm of the source vector *pSrc* and stores the result in *pNorm*.

ippsNorm_Inf. The function ippsNorm_Inf computes the C norm defined by the formula:

$$Norm_C = \max_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L1. The function ippsNorm_L1 computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L2. The function ippsNorm_L2 computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc[n]|^2}$$

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value.

Return Value

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

NormDiff

Computes the C, L1, or L2 norm of two vectors' difference.

```
IppStatus ippsNormDiff_Inf_32f(const Ipp32f* pSrc1,
     const Ipp32f* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64f(const Ipp64f* pSrc1,
     const Ipp64f* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32f(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_32fc32f(const Ipp32fc* pSrc1, const
     Ipp32fc* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64fc64f(const Ipp64fc* pSrc1, const
     Ipp64fc* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32s_Sfs(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
```

```
IppStatus ippsNormDiff_L1_32f(const Ipp32f* pSrc1,
     const Ipp32f* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_64f(const Ipp64f* pSrc1,
     const Ipp64f* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_16s32f(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_32fc64f(const Ipp32fc* pSrc1,
     const Ipp32fc* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_64fc64f(const Ipp64fc* pSrc1,
     const Ipp64fc* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_16s32s_Sfs(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L2_32f(const Ipp32f* pSrc1,
     const Ipp32f* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_64f(const Ipp64f* pSrc1,
     const Ipp64f* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_16s32f(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_32fc64f(const Ipp32fc* pSrc1,
     const Ipp32fc* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_64fc64f(const Ipp64fc* pSrc1,
     const Ipp64fc* pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_16s32s_Sfs(const Ipp16s* pSrc1,
     const Ipp16s* pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
```

Arguments

pSrc1, pSrc2	Pointers to the two source vectors; *pSrc2* can be NULL.
pNorm	Pointer to the output result.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsNorm is declared in the ipps.h file. This function computes the C, L1, or L2 norm of the source vectors' difference, and stores the result in *pNorm*.

ippsNormDiff_Inf. The function ippsNormDiff_Inf computes the C norm defined by the formula:

$$Norm_{Inf} = \max_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L1. The function ippsNormDiff_L1 computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L2. The function ippsNormDiff_L2 computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|^2}$$

Functions with Sfs suffixes perform scaling of the result value in accordance with the *scaleFactor* value.

Example 5-26 shows how to use the function ippsNormDiff.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pNorm* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-26 Using the ippsNorm Function

```
int norm( void ) {
      Ipp16s x[LEN];
      Ipp32f Norm[3];
      IppStatus st;
      int i;
      for( i=0; i<LEN; ++i ) x[i] = (Ipp16s)rand();
      ippsNormDiff_Inf_16s32f( x, 0, LEN, Norm );
      ippsNormDiff_L1_16s32f( x, 0, LEN, Norm+1 );
      st = ippsNormDiff_L2_16s32f( x, 0, LEN, Norm+2 );
      printf_32f("Norm (oo,L1,L2) =", Norm, 3, st );
      return Norm[2] <= Norm[1] && Norm[1] <= LEN*Norm[0];
}
Output:
    Norm (oo,L1,L2) =  31993.000000 1526460.000000 180270.781250
Matlab* analog:
    >> x = 32767*rand(1,100);norm(x,inf),norm(x,1),norm(x,2)
```

DotProd

Computes the dot product of two vectors.

```
IppStatus ippsDotProd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      int len, Ipp32f* pDp);
IppStatus ippsDotProd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
      int len, Ipp32fc* pDp);
IppStatus ippsDotProd_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2,
      int len, Ipp32fc* pDp);
IppStatus ippsDotProd_32f64f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      int len, Ipp64f* pDp);
```

```
IppStatus ippsDotProd_32fc64fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
     int len, Ipp64fc* pDp);

IppStatus ippsDotProd_32f32fc64fc(const Ipp32f* pSrc1,
     const Ipp32fc* pSrc2, int len, Ipp64fc* pDp);

IppStatus ippsDotProd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
     int len, Ipp64f* pDp);

IppStatus ippsDotProd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
     int len, Ipp64fc* pDp);

IppStatus ippsDotProd_64f64fc(const Ipp64f* pSrc1, const Ipp64fc*
     pSrc2, int len, Ipp64fc* pDp);

IppStatus ippsDotProd_16s64s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     int len, Ipp64s* pDp);

IppStatus ippsDotProd_16sc64sc(const Ipp16sc* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp64sc* pDp);

IppStatus ippsDotProd_16s16sc64sc(const Ipp16s* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp64sc* pDp);

IppStatus ippsDotProd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
     int len, Ipp32f* pDp);

IppStatus ippsDotProd_16sc32fc(const Ipp16sc* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp32fc* pDp);

IppStatus ippsDotProd_16s16sc32fc(const Ipp16s* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp32fc* pDp);

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

IppStatus ippsDotProd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp16sc* pDp, int scaleFactor);

IppStatus ippsDotProd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s*
     pSrc2, int len, Ipp32s* pDp, int scaleFactor);

IppStatus ippsDotProd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc*
     pSrc2, int len, Ipp32sc* pDp, int scaleFactor);

IppStatus ippsDotProd_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
     pSrc2, int len, Ipp32s* pDp, int scaleFactor);

IppStatus ippsDotProd_16s16sc32sc_Sfs(const Ipp16s* pSrc1, const
     Ipp16sc* pSrc2, int len, Ipp32sc* pDp, int scaleFactor);

IppStatus ippsDotProd_16s32s32s_Sfs(const Ipp16s* pSrc1, const Ipp32s*
     pSrc2, int len, Ipp32s* pDp, int scaleFactor);
```

```
IppStatus ippsDotProd_16s16sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc*
     pSrc2, int len, Ipp16sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16sc32sc_Sfs(const Ipp16sc* pSrc1, const
     Ipp16sc* pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc*
     pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
```

Arguments

pSrc1	Pointer to the first vector to compute the dot product value.
pSrc2	Pointer to the second vector to compute the dot product value.
pDp	Pointer to the output result.
len	Number of elements in the vector
scaleFactor	Refer to Integer Scaling.

Discussion

The function ippsDotProd is declared in the ipps.h file. This function computes the dot product (scalar value) of two vectors, *pSrc1* and *pSrc2*, and stores the result in *pDp*.

The computation is performed as follows:

$$dp = \sum_{n=0}^{len-1} pSrc1[n] \cdot pSrc2[n]$$

To compute the dot product of complex data, use the function ippsConj to conjugate one of the operands. The vectors *pSrc1* and *pSrc2* must be of equal length.

Example 5-27 shows how to use the function ippsDotProd_64f to verify orthogonality of the sine and cosine functions. Two vectors are orthogonal to each other when the dot product of the two vectors is zero.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pDp*, *pSrc1*, or *pSrc*2 pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example 5-27 Using ippsDotProd to Verify Orthogonality of Sin and Cos

```
void dotprod(void) {
      Ipp64f x[10], dp;
      int n;
      for (n = 0; n<10; ++n) x[n] = sin(IPP_2PI * n / 8);
      ippsDotProd_64f(x, x+2, 8, &dp);
      printf_64f("dp =", &dp, 1, ippStsNoErr);
}
Output:
    dp = 0.000000
Matlab* Analog:
    >> n = 0:9; x = sin(2*pi*n/8); a = x(1:8); b = x(3:10); a*b'
```

MaxEvery, MinEvery

Computes maximum or minimum value for each pair of elements of two vectors.

```
IppStatus ippsMaxEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMaxEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMaxEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMinEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMinEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMinEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the first input vector.

pSrcDst	Pointer to the second input vector which stores the result.
len	Number of elements in the vector.

Discussion

The function ippsMaxEvery is declared in the ipps.h file. This function computes the maximum between each pair of corresponding elements of two input vectors and stores the result in *pSrcDst*.
The function ippsMinEvery is declared in the ipps.h file. This function computes minimum values likewise.

Return Value

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

Sampling Functions

The functions described in this section manipulate signal samples. Sampling functions are used to change the sampling rate of the input signal and thus to obtain the signal vector of a required length. The functions perform the following operations:

- Insert zero-valued samples between neighboring samples of a signal (up-sample).
- Remove samples from between neighboring samples of a signal (down-sample).

The upsampling and downsampling functions are used by some filtering functions described in Chapter 6.

SampleUp

Up-samples a signal, conceptually increasing its sampling rate by an integer factor.

```
IppStatus ippsSampleUp_16s (const Ipp16s* pSrc, int srcLen,
     Ipp16s* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32f (const Ipp32f* pSrc, int srcLen,
     Ipp32f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64f (const Ipp64f* pSrc, int srcLen,
     Ipp64f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_16sc (const Ipp16sc* pSrc, int srcLen,
     Ipp16sc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32fc (const Ipp32fc* pSrc, int srcLen,
     Ipp32fc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64fc (const Ipp64fc* pSrc, int srcLen,
     Ipp64fc* pDst, int* pDstLen, int factor, int* pPhase);
```

Arguments

pSrc	Pointer to the input array (the signal to be up-sampled).
srcLen	Number of samples in the input array *pSrc*.
pDst	Pointer to the output array.
pDstLen	Pointer to the length value of the output array *pDst*.
factor	Factor by which the signal is up-sampled. That is, *factor* -1 zeros are inserted after each sample of the input array *pSrc*.
pPhase	Pointer to the **input** phase value which determines where each sample from *pSrc* lies within each output block of *factor* samples in *pDst*. The value of *pPhase* is required to be in the range [0; *factor*-1]. The **output** value of *pPhase* can be used for the next up-sampling with the same *factor* and next *pSrc*.

Discussion

The function ippsSampleUp is declared in the ipps.h file. This function up-samples the *srcLen*-length input array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Up-sampling inserts *factor*-1 zeros between each sample of *pSrc*. The *pPhase* argument determines where each sample from the input array lies within each output block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1].

For example, if the input phase is 0, then every *factor* samples of the output array begin with the corresponding input array sample, the other *factor*-1 samples are equal to 0. The output array length is stored by the *pDstLen* address.

The *pPhase* value is the phase of an input array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

The ippsSampleUp functionality can be described as follows:

$pDst[factor * n + phase] = pSrc[n],\ 0 \le n < srcLen$

$pDst[factor * n + m] = 0,\ 0 \le n < srcLen, 0 \le m < factor,\ m \ne phase$

$pDstLen = factor * srcLen$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst*, *pSrc*, *pDstLen*, or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcLen* is less than or equal to 0.
ippStsSampleFactorErr	Indicates an error when *factor* is less than or equal to 0.
ippStsSamplePhaseErr	Indicates an error when *pPhase* is negative, or bigger than or equal to *factor*.

SampleDown

Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

```
IppStatus ippsSampleDown_16s(const Ipp16s* pSrc, int srcLen,
     Ipp16s* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32f(const Ipp32f* pSrc, int srcLen,
     Ipp32f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64f(const Ipp64f* pSrc, int srcLen,
     Ipp64f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_16sc(const Ipp16sc* pSrc, int srcLen,
     Ipp16sc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32fc(const Ipp32fc* pSrc, int srcLen,
     Ipp32fc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64fc(const Ipp64fc* pSrc, int srcLen,
     Ipp64fc* Dst, int* pDstLen, int factor, int* pPhase);
```

Arguments

pSrc	Pointer to the input array holding the samples to be down-sampled.
srcLen	Number of samples in the input array *pSrc*.
pDst	Pointer to the array that holds the output of the function ippsSampleDown.
pDstLen	Pointer to the length of the output array *pDst*.
factor	Factor by which the signal is down-sampled. That is, *factor* - 1 samples are discarded from every block of *factor* samples in *pSrc*.
pPhase	Pointer to the **input** phase value that determines which of the samples within each block of *factor* samples from *pSrc* is not discarded and copied to *pDst*. The value of *pPhase* is

required to be in the range [0; *factor*-1]. The **output** value of *pPhase* can be used for the next down-sampling (sub-sampling) with the same *factor* and next *pSrc*.

Discussion

The function ippsSampleDown is declared in the ipps.h file. This function down-samples the *srcLen* -length array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Down-sampling discards *factor* - 1 samples from *pSrc*, copying one sample from each block of *factor* samples from *pSrc* to *pDst*. The *pPhase* argument determines which of the samples in each block is not discarded and where it lies within each input block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1]. The output array length is stored by the *pDstLen* address.

The *pPhase* value is the phase of an input array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

You can use the FIR multi-rate filter to combine filtering and resampling, for example, for antialiasing filtering before the sub-sampling procedure.

The ippsSampleDown functionality can be described as follows:

pDstLen = (srcLen + factor - 1 - phase)/factor

*pDst[n] = pSrc[factor * n + phase]*, $0 \le n < pDstLen$

phase = (factor + phase - srcLen %*factor)*%*factor*

Example 5-28 shows how to use the function ippsSampleDown.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst*, *pSrc*, *pDstLen*, or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcLen* is less than or equal to 0.
ippStsSampleFactorErr	Indicates an error when *factor* is less than or equal to 0.

`ippStsSamplePhaseErr` Indicates an error when *`pPhase`* is negative, or bigger than or equal to *`factor`*.

Example 5-28 Using the ippsSampleDown Function

```
void sampling( void ) {
      Ipp16s x[8] = { 1,2,3,4,5,6,7,8 };
      Ipp16s y[8] = {9,10,11,12,13,14,15,16}, z[8];
      int dstLen1, dstLen2, phase = 2;
      IppStatus st = ippsSampleDown_16s(x, 8, z, &dstLen1, 3, &phase);
      st = ippsSampleDown_16s(y, 8, z+dstLen1, &dstLen2, 3, &phase);
      printf_16s("down-sumpling =", z, dstLen1+dstLen2, st);
}
Output:
    down-sampling = 3 6 9 12 15
```

Filtering Functions 6

This chapter describes Intel® IPP functions that perform convolution and correlation operations, as well as linear and non-linear filtering. These functions are grouped into the following sections:

Convolution and Correlation Functions

Filtering Functions

Each section starts with a table that lists functions described in more detail later in this section, together with the brief description of operations that these functions perform.

Convolution and Correlation Functions

Convolution is an operation used to define an output signal from any linear time-invariant (LTI) processor in response to any input signal.

The correlation functions described later in this section estimate either the auto-correlation of a source vector or the cross-correlation of two vectors

The full list of functions in this group is given in Table 6-1.

Table 6-1 Intel IPP Convolution and Correlation Functions

Function Base Name	Operation
`Conv`	Performs finite, linear convolution of two sequences.
`ConvCyclic`	Performs cyclic convolution of two sequences of the fixed size.
`AutoCorr`	Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.
`CrossCorr`	Estimates the cross-correlation of two vectors.
`UpdateLinear`	Integrates an input vector with specified integration weight.

Table 6-1 Intel IPP Convolution and Correlation Functions (continued)

Function Base Name	Operation
UpdatePower	Integrates the square of an input vector with specified integration weight

Conv

Performs finite, linear convolution of two sequences.

```
IppStatus ippsConv_32f(const Ipp32f* pSrc1, int lenSrc1,
     const Ipp32f* pSrc2, int lenSrc2, Ipp32f* pDst);
IppStatus ippsConv_64f(const Ipp64f* pSrc1, int lenSrc1,
     const Ipp64f* pSrc2, int lenSrc2, Ipp64f* pDst);
IppStatus ippsConv_16s_Sfs(const Ipp16s* pSrc1, int lenSrc1,
     const Ipp16s* pSrc2, int lenSrc2, Ipp16s* pDst, int scaleFactor);
```

Arguments

pSrc1, pSrc2	Pointers to the two vectors to be convolved.
lenSrc1	Number of elements in the vector *pSrc1*.
lenSrc2	Number of elements in the vector *pSrc2*.
pDst	Pointer to the vector *pDst*. This vector stores the result of the convolution
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsConv is declared in the ipps.h file. This function performs finite linear convolution of two sequences. The *lenSrc1*- length vector *pSrc1* is convolved with the *lenSrc2*-length vector *pSrc2* to produce an (*lenSrc1* + *lenSrc2* - 1) -length vector *pDst*. The result of the convolution is defined as follows:

$$pDst[n] = \sum_{k=0}^{n} pSrc1[k] \cdot pSrc2[n-k], \quad 0 \le n < lenSrc1 + lenSrc2 - 1$$

Here $pSrc1[i] = 0$, if $i \ge lenSrc1$, and $pSrc2[j] = 0$, if $j \ge lenSrc2$.

Example 6-1 shows the code for the convolution of two vectors using ippsConv_16s_Sfs function.

Example 6-1 Using the ippsConv Function to Convolve Two Vectors

```
IppStatus convolution(void) {
      Ipp16s x[5] = {-2,0,1,-1,3}, h[2] = {0,1}, y[6];
      IppStatus st = ippsConv_16s_Sfs(x, 5, h, 2, y, 0);
      printf_16s("conv =", y, 6, st);
      return st;
}
Output:
    conv = 0 -2 0 1 -1 3
Matlab* Analog:
    >> x = [-2,0,1,-1,3]; h = [0,1]; y = conv(x,h)
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when there is not enough memory for internal buffers.

ConvCyclic

Performs cyclic convolution of two sequences of the fixed size.

```
IppStatus ippsConvCyclic8x8_32f(const Ipp32f* x, const Ipp32f* h,
      Ipp32f* y);
IppStatus ippsConvCyclic4x4_32f32fc(const Ipp32f* x, const Ipp32fc* h,
      Ipp32fc* y);
IppStatus ippsConvCyclic8x8_16s_Sfs(const Ipp16s* x, const Ipp16s* h,
      Ipp16s* y, int scaleFactor);
```

Arguments

x, h	Pointers to the vectors to be convolved.
y	Pointer to the vector *y* that stores the result of convolution
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function `ippsConvCyclic` is declared in the `ipps.h` file. This function performs cyclic convolution of two sequences of the fixed size.

Return Value

`ippStsNoErr`	Indicates no error.

AutoCorr

Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.

```
IppStatus ippsAutoCorr_32f(const Ipp32f* pSrc, int srcLen,
      Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_32f(const Ipp32f* pSrc, int srcLen,
      Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_32f(const Ipp32f* pSrc, int srcLen,
      Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_64f(const Ipp64f* pSrc, int srcLen,
      Ipp64f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_64f(const Ipp64f* pSrc, int srcLen,
      Ipp64f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_64f(const Ipp64f* pSrc, int srcLen,
      Ipp64f* pDst, int dstLen );
IppStatus ippsAutoCorr_32fc(const Ipp32fc* pSrc, int srcLen,
      Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_32fc(const Ipp32fc* pSrc, int srcLen,
      Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_32fc(const Ipp32fc* pSrc, int srcLen,
      Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_64fc(const Ipp64fc* pSrc, int srcLen,
      Ipp64fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_64fc(const Ipp64fc* pSrc, int srcLen,
      Ipp64fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_64fc(const Ipp64fc* pSrc, int srcLen,
      Ipp64fc* pDst, int dstLen);
IppStatus ippsAutoCorr_16s_Sfs(const Ipp16s* pSrc, int srcLen,
      Ipp16s* pDst, int dstLen, int scaleFactor );
IppStatus ippsAutoCorr_NormA_16s_Sfs( const Ipp16s* pSrc, int srcLen,
      Ipp16s* pDst, int dstLen, int scaleFactor );
```

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

Arguments

pSrc	Pointer to the source vector
srcLen	The number of elements in the source vector.
pDst	Pointer to the destination vector, which stores the estimated auto-correlation results of the source vector.
dstLen	The number of elements in the destination vector (length of auto-correlation).
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The ippsAutoCorr function is declared in the ipps.h file. This function estimates normal auto-correlation of the *srcLen*-length source vector *pSrc* and stores the results in the *dstLen*-length vector *pDst*. Function flavors ippsAutoCorr_NormA and ippsAutoCorr_NormB compute biased and unbiased auto-correlation of the source vector, respectively. The resulting vector *pDst* is defined by the following equations:

$$pDst[n] = \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(normal)}$$

$$pDst[n] = \frac{1}{srcLen} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(biased)}$$

$$pDst[n] = \frac{1}{srcLen - n} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \qquad \text{(unbiased)}$$

where

$$pSrc[i] = \begin{cases} pSrc[i], & 0 \le i < srcLen \\ 0, & otherwise \end{cases}$$

Application Note: The auto-correlation estimates are computed only for positive lags, since the auto-correlation for a negative lag value is the complex conjugate of the auto-correlation for the equivalent positive lag.

Return Value

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

See Also

`CrossCorr`	Estimates the cross-correlation of two vectors.

CrossCorr

Estimates the cross-correlation of two vectors.

```
IppStatus ippsCrossCorr_32f(const Ipp32f* pSrc1, int len1,
      const Ipp32f* pSrc2, int len2, Ipp32f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64f(const Ipp64f* pSrc1, int len1,
      const Ipp64f* pSrc2, int len2, Ipp64f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_32fc(const Ipp32fc* pSrc1, int len1,
      const Ipp32fc* pSrc2, int len2, Ipp32fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64fc(const Ipp64fc* pSrc1, int len1,
      const Ipp64fc* pSrc2, int len2, Ipp64fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_16s_Sfs(const Ipp16s* pSrc1, int len1,
      const Ipp16s* pSrc2, int len2, Ipp16s* pDst, int dstLen, int lowLag,
      int scaleFactor);
```

Arguments

pSrc1	Pointer to the first source vector.
len1	Number of elements in the vector *pSrc1*.
pSrc2	Pointer to the second source vector.
len2	Number of elements in the vector *pSrc2*.
pDst	Pointer to the vector which stores the results of the estimated cross-correlation of the vectors *pSrc1* and *pSrc2*.
dstLen	Number of elements in the vector *pDst*, which determines the range of lags at which the correlation estimates are computed.
lowLag	Lower value of the range of lags at which the correlation estimates are computed.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsCrossCorr is declared in the ipps.h file. This function estimates cross-correlation of the *len1*-length vector *pSrc1* and the *len2*-length vector *pSrc2*, and stores the results in the vector *pDst*.
The resulting vector *pDst* is defined by the equation:

$$pDst[n] = \sum_{i=0}^{len1-1} conj(pSrc1[i]) \cdot pSrc2[n + i + lowLag],$$

where $0 \le n < dstLen$, and

$$pSrc2[j] = \begin{cases} pSrc2[j], & 0 \le j < len2 \\ 0, & otherwise \end{cases}$$

Example 6-2 shows how to use the function ippsCrossCorr.

Example 6-2 Using the ippsCrossCorr Function

```
void crossCorr(void) {
      #undef LEN
      #define LEN 11
      Ipp32f win[LEN], y[LEN];
      IppStatus st;
      ippsSet_32f (1, win, LEN);
      ippsWinHamming_32f_I (win, LEN);
      st = ippsCrossCorr_32f (win, LEN, win, LEN, y, LEN, -(LEN-1));
      printf_32f("cross corr =", y,7,st);
}
Output:
    cross corr = 0.006400 0.026856 0.091831 0.242704 0.533230
    1.009000 1.672774
Matlab* analog:
    >> x = hamming(11)'; y = xcorr(x,x); y(1:7)
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc1`* or *`pSrc2`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len1`* or *`len2`* is less than or equal to 0.

UpdateLinear

Integrates an input vector with specified integration weight.

```
IppStatus ippsUpdateLinear_16s32s_I(const Ipp16s* pSrc, int len,
     Ipp32s* pSrcDst, int srcShiftRight, Ipp16s alpha,
     IppHintAlgorithm hint);
```

Arguments

pSrc	Pointer to the source vector.
len	Number of elements of the vector.
pSrcDst	Pointer to the input value and output result.
srcShiftRight	Shift value; must be non-negative.
alpha	Integration weight.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."

Discussion

The function ippsUpdateLinear is declared in the ipps.h file. This function performs *len* iterations in which the sum

$alpha*pSrcDst + (1-alpha)*pSrc[i]_{shift}$

is calculated and stored in *pSrcDst*.
Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and $pSrc[i]_{shift}$ is a source vector element right-shifted by the non-negative value *srcShiftRight*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

UpdatePower

Integrates the square of an input vector with specified integration weight.

```
IppStatus ippsUpdatePower_16s32s_I(const Ipp16s* pSrc, int len,
     Ipp32s* pSrcDst, int srcShiftRight,Ipp16s alpha,
     IppHintAlgorithm hint);
```

Arguments

pSrc	Pointer to the source vector.
len	Number of elements of the vector.
pSrcDst	Pointer to input and output
srcShiftRight	Shift value.
alpha	Integration weight.
hint	Suggests using specific code. The values for the *hint* argument are described in "Flag and Hint Arguments."

Discussion

The function ippsUpdatePower is declared in the ipps.h file. This function performs *len* iterations in which the sum
*alpha***pSrcDst* + (1-*alpha*)**pSrc*[*i*]$_{shift}$ **pSrc*[*i*]$_{shift}$
is calculated and stored in *pSrcDst*.
Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and *pSrc*[*i*]$_{shift}$ is a source vector element right-shifted by the non-negative value *srcShiftRight*.

Return Value

ippStsNoErr	Indicates no error.

`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

Filtering Functions

The Intel IPP functions described in this section implement the following types of filters:

- finite impulse response (FIR) filter
- adaptive finite impulse response using least mean squares (LMS) filter
- infinite impulse response (IIR) filter
- median filter.

A special set of functions is designed to generate filter coefficients for different types of FIR filters.

The full list of filtering functions is given in Table 6-2

Table 6-2 Intel IPP Filtering Functions

Function Base Name	Operation
FIR Filter Functions	
FIRInitAlloc, FIRMRInitAlloc	Allocates memory and initializes a single-rate or multi-rate FIR filter state structure.
FIRFree	Frees memory allocated by the ippsFIRInitAlloc or ippsFIRMRInitAlloc function.
FIRInit, FIRMRInit	Initializes a single-rate or multi-rate FIR filter state structure.
FIRGetStateSize, FIRMRGetStateSize	Computes the size of an external buffer for FIR filter structure.
FIRGetTaps, FIRSetTaps	Gets and sets the taps values of a FIR filter state.
FIRGetDlyLine, FIRSetDlyLine	Gets and sets the delay line contents of a FIR filter state.
FIROne	Filters a single sample through a FIR filter.
FIR	Filters a block of samples through a single-rate or multi-rate FIR filter.
Direct versions	
FIROne_Direct	Directly filters a single sample through a FIR filter.
FIR_Direct	Directly filters a block of samples through a single-rate FIR filter.
FIRMR_Direct	Directly filters a block of samples through a multi-rate FIR filter.

Table 6-2 Intel IPP Filtering Functions (continued)

Function Base Name	Operation
FIR Filter Coefficient Generating Functions	
FIRGenLowpass	Computes the lowpass FIR filter coefficients.
FIRGenHighpass	Computes the highpass FIR filter coefficients.
FIRGenBandpass	Computes the bandpass FIR filter coefficients.
FIRGenBandstop	Computes the bandstop FIR filter coefficients.
Single-Rate FIR LMS Filter Functions	
FIRLMSInitAlloc	Allocates memory and initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.
FIRLMSFree	Closes an adaptive FIR filter that uses the LMS algorithm.
FIRLMSGetTaps	Gets the taps of a FIR LMS filter.
FIRLMSGetDlyLine, FIRLMSSetDlyLine	Gets and sets the delay line contents of a FIR LMS filter.
FIRLMS	Filters an array through a FIR LMS filter.
Direct versions	
FIRLMSOne_Direct	Filters a single sample through a FIR LMS filter.
Multi-Rate FIR LMS Filter Functions	
FIRLMSMRInitAlloc	Allocates memory and initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.
FIRLMSMRFree	Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.
FIRLMSMRSetMu	Sets the adaptation step.
FIRLMSMRUpdateTaps	Updates the filter coefficients using the adaptation error value.
FIRLMSMRGetTaps, FIRLMSMRSetTaps	Gets and sets taps of a multi-rate FIR LMS filter.
FIRLMSMRGetTapsPointer	Returns the pointer to the filter coefficients.
FIRLMSMRGetDlyLine, FIRLMSMRSetDlyLine	Gets and sets the delay line contents of a multi-rate FIR LMS filter state.
FIRLMSMRGetDlyVal	Gets one delay line values from the specified position.
FIRLMSMRPutVal	Places the input value in the delay line.
FIRLMSMROne	Filters data placed in the delay line.

Table 6-2 Intel IPP Filtering Functions (continued)

Function Base Name	Operation
FIRLMSMROneVal	Filters one input value.
IIR Filter Functions	
IIRInitAlloc, IIRInitAlloc_BiQuad	Allocates memory and initializes an IIR filter.
IIRFree	Closes an IIR filter state.
IIRInit, IIRInit_BiQuad	Initializes an IIR filter state
IIRGetStateSize, IIRGetStateSize_BiQuad	Returns the length of the IIR filter state structure
IIRSetTaps	Sets the taps in an IIR filter state
IIRGetDlyLine, IIRSetDlyLine	Gets and sets the delay line values in an IIR filter state.
IIROne	Filters a single sample through an IIR filter.
Discussion	Filters a block of samples through an IIR filter.
Direct versions	
IIROne_Direct, IIROne_BiQuadDirect	Directly filters a single sample through an IIR filter.
IIR_Direct, IIR_BiQuadDirect	Directly filters a block of samples through an IIR filter
Median Filter Functions	
FilterMedian	Computes median values for each input array element.

FIR Filter Functions

The functions described in this section perform a finite impulse response filtering of input data. The functions initialize a finite impulse response filter, get and set the delay lines and filter coefficients (taps), and perform filtering. To use the FIR filter functions, follow this general scheme:

1. Call either `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc` to allocate memory and initialize the taps and the delay line in the filter state structure of a single-rate or multi-rate filter, respectively.
   Or call either `ippsFIRInit` or `ippsFIRMRInit` to initialize the taps and the delay line in the corresponding filter state structure in the previously created external buffer. The size of this buffer can be computed by calling the functions `ippsFIRGetStateSize` or `ippsFIRMRGetStateSize`, respectively.
2. Call `ippsFIROne` to filter a single sample through a single-rate filter and/or call `ippsFIR` to filter a block of consecutive samples through a single-rate or multi-rate filter.
3. To set new taps and delay line values in the previously initialized filter state, call the functions `ippsFIRSetTaps` and `ippsFIRSetDlyLine`. To get taps and delay line values of the initialized filter state, call the functions `ippsFIRGetTaps` and `ippsFIRGetDlyLine`.
4. Call `ippsFIRFree` to free dynamic memory associated with the FIR filter state structure created by `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

Alternatively, you may use the direct version of the functions. These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

Special set of functions allows to compute the filter coefficients for different filters.

FIRInitAlloc, FIRMRInitAlloc

Allocates memory and initializes a single-rate or multi-rate FIR filter state.

```
IppStatus ippsFIRInitAlloc_32f(IppsFIRState_32f** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp32f* pDlyLine);
IppStatus ippsFIRMRInitAlloc_32f(IppsFIRState_32f** pState,
     const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32f* pDlyLine);
IppStatus ippsFIRInitAlloc_64f(IppsFIRState_64f** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp64f* pDlyLine);
IppStatus ippsFIRMRInitAlloc_64f(IppsFIRState_64f** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp64f* pDlyLine);
IppStatus ippsFIRInitAlloc_32fc(IppsFIRState_32fc** pState,
     const Ipp32fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc_32fc(IppsFIRState_32fc** pState,
     const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32fc* pDlyLine);
IppStatus ippsFIRInitAlloc_64fc(IppsFIRState_64fc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp64fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc_64fc(IppsFIRState_64fc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp64fc* pDlyLine);

IppStatus ippsFIRInitAlloc32s_16s(IppsFIRState32s_16s** pState,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor,
     const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32s_16s(IppsFIRState32s_16s** pState,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor,
     int upPhase, int downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32s_16s32f(IppsFIRState32s_16s** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32s_16s32f(IppsFIRState32s_16s** pState,
     const Ipp32f *pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine);

IppStatus ippsFIRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** pState,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
     const Ipp16sc* pDlyLine);

IppStatus ippsFIRMRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** pState,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor,
     int upPhase, int downFactor, int downPhase, const Ipp16sc* pDlyLine);

IppStatus ippsFIRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);

IppStatus ippsFIRMRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc**
     pState, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine);


IppStatus ippsFIRInitAlloc32f_16s(IppsFIRState32f_16s** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);

IppStatus ippsFIRMRInitAlloc32f_16s(IppsFIRState32f_16s** pState,
     const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine);

IppStatus ippsFIRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);

IppStatus ippsFIRMRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine);


IppStatus ippsFIRInitAlloc64f_16s(IppsFIRState64f_16s** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp16s* pDlyLine);

IppStatus ippsFIRMRInitAlloc64f_16s(IppsFIRState64f_16s** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine);

IppStatus ippsFIRInitAlloc64f_32s(IppsFIRState64f_32s** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp32s* pDlyLine);

IppStatus ippsFIRMRInitAlloc64f_32s(IppsFIRState64f_32s** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32s* pDlyLine);
```

```
IppStatus ippsFIRInitAlloc64f_32f(IppsFIRState64f_32f** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp32f* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_32f(IppsFIRState64f_32f** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32f* pDlyLine);

IppStatus ippsFIRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp32sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32fc* pDlyLine);
```

Arguments

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).
downFactor	Factor used by the function ippsFIRMRInit for downsampling the multi-rate signals.
downPhase	Phase used by the function ippsFIRMRInit for downsampling the multi-rate signals.
upFactor	Factor used by the function ippsFIRMRInit for upsampling the multi-rate signals.

upPhase	Phase used by the function ippsFIRMRInit for upsampling the multi-rate signals.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *tapsLen* for single-rate filters and (*tapsLen* + *upFactor* - 1) / *upFactor* for multi-rate filters.
pState	Pointer to the FIR state structure to be created.

Discussion

The functions ippsFIRInitAlloc and ippsFIRMRInitAlloc are declared in the ipps.h file. These functions allocate memory and initialize a single-rate or multi-rate FIR filter state, respectively. The initialization functions copy the taps from the *tapsLen* -length array *pTaps* into the state structure *pState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

If the state is not created, the initialization function returns an error status.

ippsFIRInitAlloc. The function ippsFIRInitAlloc initializes the taps and the delay line in the state structure *pState* of a single-rate filter. The *tapsLen* -length array *pTaps* specifies the taps. If the delay line array *pDlyLine* is non-NULL its length is equal to *tapsLen*. Note that the delay line length is different than that for direct FIR filters (where this length is doubled).

ippsFIRMRInitAlloc. The function ippsFIRMRInitAlloc initializes the taps and the delay line in the state structure *pState* of a multi-rate filter; that is, a filter that internally upsamples and/or downsamples using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see ippsSampleUp function on page 5-139). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The argument *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see ippsSampleDown function on page 5-141). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The argument *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array *pDlyLine* is non-NULL, its length is defined as (*tapsLen* + *upFactor* - 1) / *upFactor*.

Return Value

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsFIRMRFactorErr	Indicates an error when *upFactor* (*downFactor*) is less than or equal to 0.
ippStsFIRMRPhaseErr	Indicates an error when *upPhase* (*downPhase*) is negative, or greater than or equal to *upFactor* (*downFactor*).
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRFree

Closes a FIR filter state.

```
IppStatus ippsFIRFree_32f(IppsFIRState_32f* pState);
IppStatus ippsFIRFree_64f(IppsFIRState_64f* pState);
IppStatus ippsFIRFree_32fc(IppsFIRState_32fc* pState);
```

```
IppStatus ippsFIRFree_64fc(IppsFIRState_64fc* pState);
IppStatus ippsFIRFree32s_16s(IppsFIRState32s_16s* pState);
IppStatus ippsFIRFree32f_16s(IppsFIRState32f_16s* pState);
IppStatus ippsFIRFree32sc_16sc(IppsFIRState32sc_16sc* pState);
IppStatus ippsFIRFree32fc_16sc(IppsFIRState32fc_16sc* pState);
IppStatus ippsFIRFree64f_16s(IppsFIRState64f_16s* pState);
IppStatus ippsFIRFree64f_32s(IppsFIRState64f_32s* pState);
IppStatus ippsFIRFree64f_32f(IppsFIRState64f_32f* pState);
IppStatus ippsFIRFree64fc_16sc(IppsFIRState64fc_16sc* pState);
IppStatus ippsFIRFree64fc_32sc(IppsFIRState64fc_32sc* pState);
IppStatus ippsFIRFree64fc_32fc(IppsFIRState64fc_32fc* pState);
```

Arguments

pState	Pointer to the FIR filter state structure to be closed.

Discussion

The function ippsFIRFree is declared in the ipps.h file. This function closes the FIR filter state by freeing all memory associated with the filter state structure created by ippsFIRInitAlloc or ippsFIRMRInitAlloc. Call ippsFIRFree after filtering is completed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRInit, FIRMRInit

Initializes a single-rate or multi-rate FIR filter state.

```
IppStatus ippsFIRInit_32f(IppsFIRState_32f** pState, const Ipp32f*
     pTaps, int tapsLen, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit_32f(IppsFIRState_32f** pState,
     const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_64f(IppsFIRState_64f** pState, const Ipp64f* pTaps,
     int tapsLen, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit_64f(IppsFIRState_64f** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_32fc(IppsFIRState_32fc** pState, const Ipp32fc*
     pTaps, int tapsLen, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit_32fc(IppsFIRState_32fc** pState,
     const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_64fc(IppsFIRState_64fc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp64fc* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit_64fc(IppsFIRState_64fc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);


IppStatus ippsFIRInit32s_16s(IppsFIRState32s_16s** pState,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor,
     const Ipp16s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32s_16s(IppsFIRState32s_16s** pState,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor,
     int upFactor, int upPhase, int downFactor, int downPhase,
     const Ipp16s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRInit32s_16s32f(IppsFIRState32s_16s** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32s_16s32f(IppsFIRState32s_16s** pState,
     const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRInit32sc_16sc(IppsFIRState32sc_16sc** pState,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
     const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32sc_16sc(IppsFIRState32sc_16sc** pState,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor,
     int upPhase, int downFactor, int downPhase,
     const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32sc_16sc32fc(IppsFIRState32sc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32sc_16sc32fc(IppsFIRState32sc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine,
     Ipp8u* pBuffer);


IppStatus ippsFIRInit32f_16s(IppsFIRState32f_16s** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32f_16s(IppsFIRState32f_16s** pState,
     const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRInit32fc_16sc(IppsFIRState32fc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit32fc_16sc(IppsFIRState32fc_16sc** pState,
     const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine,
     Ipp8u* pBuffer);
```

```
IppStatus ippsFIRInit64f_16s(IppsFIRState64f_16s** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp16s* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64f_16s(IppsFIRState64f_16s** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16s* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRInit64f_32s(IppsFIRState64f_32s** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp32s* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64f_32s(IppsFIRState64f_32s** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32s* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRInit64f_32f(IppsFIRState64f_32f** pState,
     const Ipp64f* pTaps, int tapsLen, const Ipp32f* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64f_32f(IppsFIRState64f_32f** pState,
     const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32f* pDlyLine,
     Ipp8u* pBuffer);


IppStatus ippsFIRInit64fc_16sc(IppsFIRState64fc_16sc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64fc_16sc(IppsFIRState64fc_16sc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp16sc* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRInit64fc_32sc(IppsFIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp32sc* pDlyLine,
     Ipp8u* pBuffer);

IppStatus ippsFIRMRInit64fc_32sc(IppsFIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32sc* pDlyLine,
     Ipp8u* pBuffer);
```

```
IppStatus ippsFIRInit64fc_32fc(IppsFIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine,
     Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64fc_32fc(IppsFIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
     int downFactor, int downPhase, const Ipp32fc* pDlyLine,
     Ipp8u* pBuffer);
```

Arguments

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).
downFactor	Factor used by the function ippsFIRMRInit for downsampling the multi-rate signals.
downPhase	Phase used by the function ippsFIRMRInit for downsampling the multi-rate signals.
upFactor	Factor used by the function ippsFIRMRInit for upsampling the multi-rate signals.
upPhase	Phase used by the function ippsFIRMRInit for upsampling the multi-rate signals.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *tapsLen* for single-rate filters and (*tapsLen* + *upFactor* - 1) / *upFactor* for multi-rate filters.
pState	Pointer to the FIR state structure to be created.
pBuffer	Pointer to the external buffer for FIR state structure.

Discussion

The functions ippsFIRInit and ippsFIRMRInit are declared in the ipps.h file. These functions initialize a single-rate or multi-rate FIR filter state structure, respectively, in the external buffer. The size of this buffer should be computed

previously by calling the functions FIRGetStateSize, FIRMRGetStateSize. The initialization functions copy the taps from the *tapsLen* -length array *pTaps* into the state structure *pState*. To scale integer taps, use the *tapsFactor* value.
The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array contents is copied into the state structure *pState*, otherwise the delay line values in the state structure are initialized to 0.

ippsFIRInit. The function ippsFIRInit initializes the taps and the delay line in the state structure *pState* of a single-rate filter. The *tapsLen* -length array *pTaps* specifies the taps. If the delay line array *pDlyLine* is not NULL, its length is equal to *tapsLen*. Note that the delay line length is different than that for direct FIR filters (where this length is doubled).

ippsFIRMRInit. The function ippsFIRMRInit initializes the taps and the delay line in the state structure *pState* of a multi-rate filter, that is, a filter that internally upsamples and/or downsamples signals using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see ippsSampleUp function on page 5-139). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The argument *upPhase* is the parameter, which determines where a non-zero sample lies within the *upFactor*-length block of the upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see ippsSampleDown function on page 5-141). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

The argument *downPhase* is the parameter, which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array *pDlyLine* is not NULL, its length is defined as (*tapsLen* + *upFactor* - 1) / *upFactor*.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsFIRMRFactorErr	Indicates an error when *upFactor* (*downFactor*) is less than or equal to 0.
ippStsFIRMRPhaseErr	Indicates an error when *upPhase* (*downPhase*) is negative, or greater than or equal to *upFactor* (*downFactor*).

FIRGetStateSize, FIRMRGetStateSize

Returns the length of the FIR filter state structure.

```
IppStatus ippsFIRGetStateSize_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_32f(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_64f(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_32fc(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_64fc(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);

IppStatus ippsFIRGetStateSize32s_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32s_16s(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize));
IppStatus ippsFIRGetStateSize32s_16s32f(int tapsLen, int*
     pBufferSize));
IppStatus ippsFIRMRGetStateSize32s_16s32f(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize));
```

```
IppStatus ippsFIRGetStateSize32sc_16sc(int tapsLen, int*
     pBufferSize));
IppStatus ippsFIRMRGetStateSize32sc_16sc(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize));
IppStatus ippsFIRGetStateSize32sc_16sc32fc(int tapsLen, int*
     pBufferSize));
IppStatus ippsFIRMRGetStateSize32sc_16sc32fc(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize));


IppStatus ippsFIRGetStateSize32f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32f_16s(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize32fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32fc_16sc(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);


IppStatus ippsFIRGetStateSize64f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_16s(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_32s(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_32f(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);


IppStatus ippsFIRGetStateSize64fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_16sc(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_32sc(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32fc(int tapsLen, int* pBufferSize);
```

```
IppStatus ippsFIRMRGetStateSize64fc_32fc(int tapsLen, int upFactor,
     int downFactor, int* pBufferSize);
```

Arguments

tapsLen	Number of elements in the array containing the tap values.
downFactor	Factor used by the function ippsFIRMRInit for downsampling the multi-rate signals.
upFactor	Factor used by the function ippsFIRMRInit for upsampling the multi-rate signals.
pBufferSize	Pointer to the computed buffer size value.

Discussion

The functions ippsFIRGetStateSize and ippsFIRMRGetStateSize are declared in the ipps.h file. These functions compute the size of the external buffer for a single-rate or multi-rate FIR filter state, respectively, and store the result in *pBufferSize*.

To compute the size of a buffer for the single-rate FIR filter state, the number of taps *tapsLen* only need be specified.
To compute the size of a buffer for the multi-rate FIR filter state, the number of taps *tapsLen* and upsampling/downsampling parameters *upFactor* and *downFactor* should be specified.
The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see ippsSampleUp function on page 5-139). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see ippsSampleDown function on page 5-141). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pBufferSize* pointer is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.

`ippStsFIRMRFactorErr`	Indicates an error when *`upFactor`* (*`downFactor`*) is less than or equal to 0.

FIRGetTaps, FIRSetTaps

Gets an sets the taps of a FIR filter state.

```
IppStatus ippsFIRGetTaps_32f(const IppsFIRState_32f* pState,
     Ipp32f* pTaps);
IppStatus ippsFIRGetTaps_64f(const IppsFIRState_64f* pState,
     Ipp64f* pTaps);
IppStatus ippsFIRGetTaps32f_16s(const IppsFIRState32f_16s* pState,
     Ipp32f* pTaps);
IppStatus ippsFIRGetTaps64f_16s(const IppsFIRState64f_16s* pState,
     Ipp64f* pTaps);
IppStatus ippsFIRGetTaps64f_32s(const IppsFIRState64f_32s* pState,
     Ipp64f* pTaps);
IppStatus ippsFIRGetTaps64f_32f(const IppsFIRState64f_32f* pState,
     Ipp64f* pTaps);
IppStatus ippsFIRGetTaps_32fc(const IppsFIRState_32fc* pState,
     Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps_64fc(const IppsFIRState_64fc* pState,
     Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps32fc_16sc(const IppsFIRState32fc_16sc* pState,
     Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps64fc_16sc(const IppsFIRState64fc_16sc* pState,
     Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps64fc_32sc(const IppsFIRState64fc_32sc* pState,
     Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps64fc_32fc(const IppsFIRState64fc_32fc* pState,
     Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps32s_16s32f(const IppsFIRState32s_16s* pState,
     Ipp32f* pTaps);
```

```
IppStatus ippsFIRGetTaps32sc_16sc32fc(const IppsFIRState32sc_16sc*
     pState, Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps32s_16s(const IppsFIRState32s_16s* pState,
     Ipp32s* pTaps, int* tapsFactor);
IppStatus ippsFIRGetTaps32sc_16sc(const IppsFIRState32sc_16sc* pState,
     Ipp32sc* pTaps, int* tapsFactor);


IppStatus ippsFIRSetTaps_32f(const Ipp32f* pTaps,
     IppsFIRState_32f* pState);
IppStatus ippsFIRSetTaps_64f(const Ipp64f* pTaps,
     IppsFIRState_64f* pState);
IppStatus ippsFIRSetTaps32f_16s(const Ipp32f* pTaps,
     IppsFIRState32f_16s* pState);
IppStatus ippsFIRSetTaps64f_16s(const Ipp64f* pTaps,
     IppsFIRState64f_16s* pState);
IppStatus ippsFIRSetTaps64f_32s(const Ipp64f* pTaps,
     IppsFIRState64f_32s* pState);
IppStatus ippsFIRSetTaps64f_32f(const Ipp64f* pTaps,
     IppsFIRState64f_32f* pState);
IppStatus ippsFIRSetTaps_32fc(const Ipp32fc* pTaps,
     IppsFIRState_32fc* pState);
IppStatus ippsFIRSetTaps_64fc(const Ipp64fc* pTaps,
     IppsFIRState_64fc* pState);
IppStatus ippsFIRSetTaps32fc_16sc(const Ipp32fc* pTaps,
     IppsFIRState32fc_16sc* pState);
IppStatus ippsFIRSetTaps64fc_16sc(const Ipp64fc* pTaps,
     IppsFIRState64fc_16sc* pState);
IppStatus ippsFIRSetTaps64fc_32sc(const Ipp64fc* pTaps,
     IppsFIRState64fc_32sc* pState);
IppStatus ippsFIRSetTaps64fc_32fc(const Ipp64fc* pTaps,
     IppsFIRState64fc_32fc* pState);
IppStatus ippsFIRSetTaps32s_16s32f(const Ipp32f* pTaps,
     IppsFIRState32s_16s* pState);
IppStatus ippsFIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps,
     IppsFIRState32sc_16sc* pState);
```

```
IppStatus ippsFIRSetTaps32s_16s(const Ipp32s* pTaps,
     IppsFIRState32s_16s* pState, int tapsFactor);
IppStatus ippsFIRSetTaps32sc_16sc(const Ipp32sc* pTaps,
     IppsFIRState32sc_16sc* pState, int tapsFactor);
```

Arguments

pState	Pointer to the FIR filter state structure.
pTaps	Pointer to the array containing the tap values.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).

Discussion

The functions and ippsFIRSetTaps are declared in the ipps.h file.

ippsFIRGetTaps. The function ippsFIRGetTaps copies the tap values from the initialized FIR state structure *pState* to the *tapsLen* -length array *pTaps*. To scale integer taps, use the *tapsFactor* value.

ippsFIRSetTaps. The function ippsFIRSetTaps sets new tap values in the previously initialized FIR filter state structure *pState*. New tap values must be specified in the array *pTaps*. The length of the array *pTaps* must be equal to the *tapsLen* parameter value of the initialized filter state.

To scale integer taps, use the *tapsFactor* value.

Before calling ippsFIRGetTaps or ippsFIRSetTaps functions, initialize the filter state by calling one of the initialization functions.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRGetDlyLine, FIRSetDlyLine

Gets and sets the delay line contents of a FIR filter state.

```
IppStatus ippsFIRGetDlyLine_32f(const IppsFIRState_32f* pState,
      Ipp32f* pDlyLine);
IppStatus ippsFIRGetDlyLine_64f(const IppsFIRState_64f* pState,
      Ipp64f* pDlyLine);
IppStatus ippsFIRGetDlyLine32s_16s(const IppsFIRState32s_16s* pState,
      Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine32f_16s(const IppsFIRState32f_16s* pState,
      Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_16s(const IppsFIRState64f_16s* pState,
      Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32s(const IppsFIRState64f_32s* pState,
      Ipp32s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32f(const IppsFIRState64f_32f* pState,
      Ipp32f* pDlyLine);


IppStatus ippsFIRGetDlyLine_32fc(const IppsFIRState_32fc* pState,
      Ipp32fc* pDlyLine);
IppStatus ippsFIRGetDlyLine_64fc(const IppsFIRState_64fc* pState,
      Ipp64fc* pDlyLine);
IppStatus ippsFIRGetDlyLine32sc_16sc(const IppsFIRState32sc_16sc*
      pState, Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine32fc_16sc(const IppsFIRState32fc_16sc*
      pState, Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_16sc(const IppsFIRState64fc_16sc*
      pState, Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_32sc(const IppsFIRState64fc_32sc*
      pState, Ipp32sc* pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine64fc_32fc(const IppsFIRState64fc_32fc*
     pState, Ipp32fc* pDlyLine);


IppStatus ippsFIRSetDlyLine_32f(IppsFIRState_32f* pState,
     const Ipp32f* pDlyLine);
IppStatus ippsFIRSetDlyLine_64f(IppsFIRState_64f* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsFIRSetDlyLine32s_16s(IppsFIRState32s_16s* pState,
     const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine32f_16s(IppsFIRState32f_16s* pState,
     const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_16s(IppsFIRState64f_16s* pState,
     const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32s(IppsFIRState64f_32s* pState,
     const Ipp32s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32f(IppsFIRState64f_32f* pState,
     const Ipp32f* pDlyLine);


IppStatus ippsFIRSetDlyLine_32fc(IppsFIRState_32fc* pState,
     const Ipp32fc* pDlyLine);
IppStatus ippsFIRSetDlyLine_64fc(IppsFIRState_64fc* pState,
     const Ipp64fc* pDlyLine);
IppStatus ippsFIRSetDlyLine32sc_16sc(IppsFIRState32sc_16sc* pState,
     const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine32fc_16sc(IppsFIRState32fc_16sc* pState,
     const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_16sc(IppsFIRState64fc_16sc* pState,
     const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32sc(IppsFIRState64fc_32sc* pState,
     const Ipp32sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32fc(IppsFIRState64fc_32fc* pState,
     const Ipp32fc* pDlyLine);
```

Arguments

pState	Pointer to the FIR filter state structure.

pDlyLine	Pointer to the array holding the delay line values.

Discussion

The functions ippsFIRGetDlyLine and ippsFIRSetDlyLine are declared in the ipps.h file. These functions get and set the delay line values of a FIR filter state.

ippsFIRGetDlyLine. The function ippsFIRGetDlyLine copies the delay line values from the state structure *pState* and stores them into *pDlyLine*. The destination array *pDlyLine* will contain samples in the reverse order as compared to the order of samples in the source vector.

ippsFIRSetDlyLine. The function ippsFIRSetDlyLine copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. The source array *pDlyLine* should contain samples in the reverse order as compared to the order of samples in the source vector.

Before calling either ippsFIRGetDlyLine or ippsFIRSetDlyLine, initialize the filter state by calling the function ippsFIRInitAlloc or ippsFIRMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIROne

Filters a single sample through a FIR filter.

```
IppStatus ippsFIROne_32f(Ipp32f src, Ipp32f* pDstVal,
     IppsFIRState_32f* pState);
IppStatus ippsFIROne_64f(Ipp64f src, Ipp64f* pDstVal,
     IppsFIRState_64f* pState);
IppStatus ippsFIROne64f_32f(Ipp32f src, Ipp32f* pDstVal,
     IppsFIRState64f_32f* pState);
```

```
IppStatus ippsFIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     IppsFIRState_32fc* pState);
IppStatus ippsFIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal,
     IppsFIRState_64fc* pState);
IppStatus ippsFIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     IppsFIRState64fc_32fc* pState);

IppStatus ippsFIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
     IppsFIRState64f_32s* pState, int scaleFactor);

IppStatus ippsFIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
     IppsFIRState64fc_32sc* pState, int scaleFactor);
```

Arguments

pState	Pointer to the FIR filter state structure.
src	Input sample to be filtered by the function ippsFIROne.
pDstVal	Pointer to the output sample filtered by the function ippsFIROne.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIROne is declared in the ipps.h file. This function filters a single sample *src* through a single-rate filter and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)* and the taps are denoted *h(i)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Before calling ippsFIROne, initialize the filter state by calling ippsFIRInitAlloc. Specify the number of taps *tapsLen*, the tap values in *pTaps*, and the delay line values in *pDlyLine* beforehand.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIR

Filters a block of samples through a single-rate or multi-rate FIR filter.

```
IppStatus ippsFIR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int numIters, IppsFIRState_32f* pState);
IppStatus ippsFIR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     int numIters, IppsFIRState_64f* pState);
IppStatus ippsFIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int numIters, IppsFIRState_32fc* pState);
IppStatus ippsFIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     int numIters, IppsFIRState_64fc* pState);
```

```
IppStatus ippsFIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     int numIters, IppsFIRState64f_32f* pState);

IppStatus ippsFIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     int numIters, IppsFIRState64fc_32fc* pState);


IppStatus ippsFIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int numIters, IppsFIRState32s_16s* pState, int scaleFactor);

IppStatus ippsFIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int numIters, IppsFIRState32f_16s* pState, int scaleFactor);

IppStatus ippsFIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int numIters, IppsFIRState64f_16s* pState, int scaleFactor);

IppStatus ippsFIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst,
     int numIters, IppsFIRState64f_32s* pState, int scaleFactor);

IppStatus ippsFIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, IppsFIRState32sc_16sc* pState, int scaleFactor);

IppStatus ippsFIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, IppsFIRState32fc_16sc* pState, int scaleFactor);

IppStatus ippsFIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, IppsFIRState64fc_16sc* pState, int scaleFactor);

IppStatus ippsFIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
     int numIters, IppsFIRState64fc_32sc* pState, int scaleFactor);


IppStatus ippsFIR_32f_I(Ipp32f* pSrcDst, int numIters,
     IppsFIRState_32f* pState);

IppStatus ippsFIR_64f_I(Ipp64f* pSrcDst, int numIters,
     IppsFIRState_64f* pState);

IppStatus ippsFIR64f_32f_I(Ipp32f* pSrcDst, int numIters,
     IppsFIRState64f_32f* pState);

IppStatus ippsFIR_32fc_I(Ipp32fc* pSrcDst, int numIters,
     IppsFIRState_32fc* pState);

IppStatus ippsFIR_64fc_I(Ipp64fc* pSrcDst, int numIters,
     IppsFIRState_64fc* pState);

IppStatus ippsFIR64fc_32fc_I(Ipp32fc* pSrcDst, int numIters,
     IppsFIRState64fc_32fc* pState);
```

```
IppStatus ippsFIR32s_16s_ISfs(Ipp16s* pSrcDst, int numIters,
     IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIR32f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
     IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
     IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_32s_ISfs(Ipp32s* pSrcDst, int numIters,
     IppsFIRState64f_32s* pState, int scaleFactor);
IppStatus ippsFIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
     IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
     IppsFIRState64fc_32sc* pState, int scaleFactor);
```

Arguments

pState	Pointer to the FIR filter state structure.
pSrc	Pointer to the input array to be filtered by the function ippsFIR.
pDst	Pointer to the output array filtered by the function ippsFIR.
pSrcDst	Pointer to the input and output array (for the in-place operation) to be filtered by the function ippsFIR.
numIters	Parameter associated with the number of samples to be filtered by the function ippsFIR. For single-rate filters, the *numIters* samples in the input array are filtered and the resulting *numIters* samples are stored in the output array. For multi-rate filters, the (*numIters* * *downFactor*) samples in the input array are filtered and the resulting (*numIters* * *upFactor*) samples are stored in the output array.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIR is declared in the ipps.h file. This function filters an input array *pSrc* or *pSrcDst* through a single-rate or multi-rate filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.

For single-rate filters, the *numIters* samples in the array *pSrc* or *pSrcDst* are filtered, and the resulting *numIters* samples are stored in the array *pDst* or *pSrcDst*. The results are identical to *numIters* consecutive calls to ippsFIROne.

In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters$$

After the function has performed calculations, it updates the delay line values stored in the state. For a single-rate filter, the *numIters* parameter defines the length of the source and destination arrays.

For multi-rate filters, ippsFIR filters (*numIters*downFactor*) input samples and stores the resulting (*numIters*upFactor*) samples in the output array. The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps. Thus, the function does not create an internal (*upFactor*srcLen*)-size buffer to store the upsampling result.

Before calling ippsFIR, initialize the filter state by calling either ippsFIRInitAlloc or ippsFIRMRInitAlloc. Specify the number of taps *tapsLen*, the tap values in *pTaps*, and the delay line values in *pDlyLine* beforehand.

Example 6-3 illustrates single-rate filtering with the function ippsFIR_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsSizeErr	Indicates an error when *numIters* is less or equal to 0.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

Example 6-3 Single-Rate Filtering with the ippsFIR Function

```
IppStatus fir(void) {
#undef NUMITERS
#define NUMITERS 150
     int n;
     IppStatus status;
     IppsIIRState_32f *ictx;
     IppsFIRState_32f *fctx;
     Ipp32f *x = ippsMalloc_32f(NUMITERS),
            *y = ippsMalloc_32f(NUMITERS),
            *z = ippsMalloc_32f(NUMITERS);
     const float taps[] = {
         0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
         0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
     };
     for (n =0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *0.2);
     ippsIIRInitAlloc_32f( &ictx, taps, 10, NULL );
     ippsFIRInitAlloc_32f( &fctx, taps, 11, NULL );
     status = ippsIIR_32f( x, y, NUMITERS, ictx);
     printf_32f("IIR 32f output + 120 =", y+120, 5, status);
     ippsIIRFree_32f(ictx);
     status = ippsFIR_32f( x, z, NUMITERS, fctx );
     printf_32f("FIR 32f output + 120 =", z+120, 5, status);
     ippsFIRFree_32f(fctx);
     ippsFree(z);
     ippsFree(y);
     ippsFree(x);
     return status;
}
Output:
    IIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
    FIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
Matlab* Analog:
    >> F = 0.2; N = 150; n = 0:N-1; x = sin(2*pi*n*F);
    y = filter(fir1(10,0.15),1,x); y(121:125)
```

FIROne_Direct

Directly filters a single sample through a FIR filter.

```
IppStatus ippsFIROne_Direct_32f(Ipp32f src, Ipp32f* pDstVal,
      const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64f(Ipp64f src, Ipp64f* pDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal,
      const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64fc(Ipp64fc src, Ipp64fc* pDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIROne64f_Direct_32f(Ipp32f src, Ipp32f* pDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIROne64fc_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
      const Ipp16s* pTapsQ15, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
      const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIROne32fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
      const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64fc_Direct_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);


IppStatus ippsFIROne32s_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
      const Ipp32s* pTaps, int tapsLen, int tapsFactor,
      Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32sc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
      const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
      Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);


IppStatus ippsFIROne_Direct_32f_I(Ipp32f* pSrcDstVal,
      const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_64f_I(Ipp64f* pSrcDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_32fc_I(Ipp32fc* pSrcDstVal,
      const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_64fc_I(Ipp64fc* pSrcDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIROne64f_Direct_32f_I(Ipp32f* pSrcDstVal,
      const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
      int* pDlyLineIndex);

IppStatus ippsFIROne64fc_Direct_32fc_I(Ipp32fc* pSrcDstVal,
      const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
      int* pDlyLineIndex);
```

```
IppStatus ippsFIROne_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
     const Ipp16s* pTaps, int tapsLen, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32f_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
     const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
     const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_32s_ISfs(Ipp32s* pSrcDstVal,
     const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
     const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
     const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDstVal,
     const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32s_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
     const Ipp32s* pTaps, int tapsLen, int tapsFactor,
     Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
     const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
     Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

Arguments

src	Input sample to be filtered by the function.
pDstVal	Pointer to the output sample filtered by the function.
pSrcDstVal	Pointer to the input and output sample for in-place operation.
pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.

pTapsQ15	Pointer to the array containing the tap values, represented in Q0.15 format. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *2*tapsLen*. Note that the delay line length is different than that for FIR filters using state structure.
pDlyLineIndex	Pointer to the current delay line index.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIROne_Direct is declared in the ipps.h file. This function directly filters a single sample *src* or *pSrcDstVal* through a single-rate filter and stores the result in *pDstVal* or *pSrcDstVal*.
The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps, the *tapsFactor* value is used.
The set of *tapsLen* input samples is copied twice to the *2*tapsLen*-length array *pDlyLine*. Double length of the delay line in direct FIR filters is used to improve filter performance by decreasing the number of sample copyings. The current delay line index is specified in the *pDlyLineIndex*.
The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)* and the taps are denoted *h(i)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Return Value

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

FIR_Direct

Directly filters a block of samples through a single-rate FIR filter.

```
IppStatus ippsFIR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, const Ipp32f* pTaps, int tapsLen,
      Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen,
      Ipp64f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, const Ipp32fc* pTaps, int tapsLen,
      Ipp32fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen,
      Ipp64fc* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen,
      Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen,
      Ipp32fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIR_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
      int numIters, const Ipp16s* pTapsQ15, int tapsLen, Ipp16s*
      pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int numIters, const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int numIters, const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst,
     int numIters, const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
     int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
     int* pDlyLineIndex, int scaleFactor);


IppStatus ippsFIR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     int numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor,
     Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
     Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);


IppStatus ippsFIR_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
     const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
     int* pDlyLineIndex);

IppStatus ippsFIR_Direct_64f_I(Ipp64f* pSrcDst, int numIters,
     const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
     int* pDlyLineIndex);

IppStatus ippsFIR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
     const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
     int* pDlyLineIndex);

IppStatus ippsFIR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters,
     const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
     int* pDlyLineIndex);
```

```
IppStatus ippsFIR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
      int* pDlyLineIndex);
IppStatus ippsFIR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
      int* pDlyLineIndex);


IppStatus ippsFIR_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp16s* pTapsQ15, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
      const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);


IppStatus ippsFIR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
      const Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine,
      int* pDlyLineIndex, int scaleFactor);
```

Arguments

pSrc	Pointer to the input array to be filtered.
pDst	Pointer to the output array.
pSrcDst	Pointer to the input and output array for the in-place operation.
numIters	Number of samples in the input array to be filtered.
pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
pTapsQ15	Pointer to the array containing the tap values, represented in Q0.15 format. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *2*tapsLen*. Note that the delay line length is different than that for FIR filters using state structure.
pDlyLineIndex	Pointer to the current delay line index.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIR_Direct is declared in the ipps.h file. This function filters an input array *pSrc* or *pSrcDst* containing *numIters* samples through a single-rate filter, and stores the resulting *numIters* samples in *pDst* or *pSrcDst*, respectively. The results are identical to *numIters* consecutive calls to ippsFIROne_Direct.
The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used.
The set of *tapsLen* input samples is copied to the *2*tapsLen*-length array *pDlyLine*. Double length of the delay line in direct FIR filters is used to improve filter performance by decreasing the number of sample copyings. The current delay line

index is specified in the *pDlyLineIndex*.
The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

In the following definition of the FIR filter, the sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*.

The return value *y(n)* is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than or equal to 0.
ippStsSizeErr	Indicates an error when *numIters* is less than or equal to 0.

FIRMR_Direct

Directly filters a block of samples through a multi-rate FIR filter.

```
IppStatus ippsFIRMR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, const Ipp32f* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp64f* pDlyLine);
IppStatus ippsFIRMR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp32fc* pDlyLine);
IppStatus ippsFIRMR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp64fc* pDlyLine);
```

```
IppStatus ippsFIRMR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int numIters, const Ipp64f* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp32f* pDlyLine);

IppStatus ippsFIRMR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      int numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase, Ipp32fc* pDlyLine);


IppStatus ippsFIRMR32f_Direct_16s_Sfs(const Ipp16s* pSrc,
      Ipp16s* pDst, int numIters, const Ipp32f* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64f_Direct_16s_Sfs(const Ipp16s* pSrc,
      Ipp16s* pDst, int numIters, const Ipp64f* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64f_Direct_32s_Sfs(const Ipp32s* pSrc,
      Ipp32s* pDst, int numIters, const Ipp64f* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp32s* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
      Ipp16sc* pDst, int numIters, const Ipp32fc* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
      Ipp16sc* pDst, int numIters, const Ipp64fc* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc,
      Ipp32sc* pDst, int numIters, const Ipp64fc* pTaps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp32sc* pDlyLine, int scaleFactor);


IppStatus ippsFIRMR32s_Direct_16s_Sfs(const Ipp16s* pSrc,
      Ipp16s* pDst, int numIters, const Ipp32s* pTaps, int tapsLen,
      int tapsFactor, int upFactor, int upPhase, int downFactor,
      int downPhase, Ipp16s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
      Ipp16sc* pDst, int numIters, const Ipp32sc* pTaps, int tapsLen,
      int tapsFactor, int upFactor, int upPhase, int downFactor,
      int downPhase, Ipp16sc* pDlyLine, int scaleFactor)
```

```
IppStatus ippsFIRMR_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
      const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR_Direct_64f_I(Ipp64f* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp64f* pDlyLine);;)
IppStatus ippsFIRMR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
      const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32fc* pDlyLine);
IppStatus ippsFIRMR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp64fc* pDlyLine);
IppStatus ippsFIRMR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters,
      const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
      const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
      const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
      int downFactor, int downPhase, Ipp32sc* pDlyLine, int scaleFactor);


IppStatus ippsFIRMR32s_Direct_16s_ISfs(Ipp16s* pSrcDst,
      int numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst,
      int numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
      int upFactor, int upPhase, int downFactor, int downPhase,
      Ipp16sc* pDlyLine, int scaleFactor);
```

Arguments

pSrc	Pointer to the input array to be filtered.
pDst	Pointer to the output array.
pSrcDst	Pointer to the input and output array for the in-place operation.
numIters	Parameter associated with the number of samples to be filtered by the function. The (*numIters* * *downFactor*) samples of the input array are filtered and the resulting (*numIters* * *upFactor*) samples are stored in the output array
pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is (*tapsLen* + *upFactor* - 1) / *upFactor*.

upFactor	Factor for upsampling the multi-rate signals.
downFactor	Factor for downsampling the multi-rate signals.
upPhase	Phase for upsampling the multi-rate signals.
downPhase	Phase for downsampling the multi-rate signals.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsFIRMR_Direct is declared in the ipps.h file. This function filters an input array *pSrc* or *pSrcDst* through a multi-rate filter, and stores the resulting samples in *pDst* or *pSrcDst*, respectively. The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used. The array *pDlyLine* specifies the delay line values. The input array contains (*numIters*downFactor*) samples, and the output array stores the resulting (*numIters*upFactor*) samples.
The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps.

The argument *upFactor* is the factor by which the filtered signal is internally upsampled (see ippsSampleUp function on page 5-139). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The argument *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The argument *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see ippsSampleDown function on page 5-141). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The argument *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response. The length of the delay line array *pDlyLine* is defined as (*tapsLen + upFactor* - 1) / *upFactor*.
The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsFIRLenErr`	Indicates an error when *`tapsLen`* is less than or equal to 0.
`ippStsSizeErr`	Indicates an error when *`numIters`* is less than or equal to 0.
`ippStsFIRMRFactorErr`	Indicates an error when *`upFactor`* (*`downFactor`*) is less than or equal to 0.
`ippStsFIRMRPhaseErr`	Indicates an error when *`upPhase`* (*`downPhase`*) is negative, or greater than or equal to *`upFactor`* (*`downFactor`*).

FIR Filter Coefficient Generating Functions

The functions described in this section compute coefficients (tap values) for different FIR filters by windowing the ideal infinite filter coefficients.

FIRGenLowpass

Computes lowpass FIR filter coefficients.

```
IppStatus ippsFIRGenLowpass_64f(Ipp64f rFreq, Ipp64f* taps,
      int tapsLen, IppWinType winType, IppBool doNormal);
```

Arguments

`rFreq`	Normalized cut-off frequency, should be in the range (0, 0,5).
`pTaps`	Pointer to the array where computed tap values are stored. The number of elements in the array is *`tapsLen`*.
`tapsLen`	Number of elements in the array containing the tap values; should be equal or greater than 5.

winType Values of this argument specify what type of window will be used in computations. The *winType* must have one of the following values:

ippWinBartlett	Bartlett window;
ippWinBlackman	Blackman window;
ippWinHamming	Hamming window;
ippWinHann	Hann window.

doNormal Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The *doNormal* must have one of the following values:

ippTrue	The function computes normalized sequence of coefficients.
ippFalse	The function computes non-normalized sequence of coefficients.

Discussion

The function ippsFIRGenLowpass is declared in the ipps.h file. This function computes *tapsLen* coefficients for lowpass FIR filter with the cut-off frequency *rFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pTaps* pointer is NULL.
ippStsSizeErr	Indicates an error when the *tapsLen* is less than 5, or *rFreq* is out of range.

FIRGenHighpass

Computes highpass FIR filter coefficients.

```
IppStatus ippsFIRGenHighpass_64f(Ipp64f rFreq, Ipp64f* taps,
      int tapsLen, IppWinType winType, IppBool doNormal);
```

Arguments

rFreq	Normalized cut-off frequency, should be in the range (0, 0,5).
pTaps	Pointer to the array where computed tap values are stored. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values; should be equal or greater than 5.
winType	Values of this argument specify what type of window will be used in computations. The *winType* must have one of the following values:
	`ippWinBartlett` Bartlett window;
	`ippWinBlackman` Blackman window;
	`ippWinHamming` Hamming window;
	`ippWinHann` Hann window.
doNormal	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The *doNormal* must have one of the following values:
	`ippTrue` The function computes normalized sequence of coefficients.
	`ippFalse` The function computes non-normalized sequence of coefficients.

Discussion

The function ippsFIRGenHighpass is declared in the ipps.h file. This function computes *tapsLen* coefficients for highpass FIR filter the cut-off frequency *rFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pTaps* pointer is NULL.
ippStsSizeErr	Indicates an error when the *tapsLen* is less than 5, or *rFreq* is out of the range.

FIRGenBandpass

Computes bandpass FIR filter coefficients.

```
IppStatus ippsFIRGenBandpass_64f(Ipp64f rLowFreq, Ipp64f rHighFreq,
      Ipp64f* pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

Arguments

rLowFreq	Normalized low cut-off frequency, should be in the range (0, 0,5) and less than *rHighFreq*.
rHighFreq	Normalized high cut-off frequency, should be in the range (0, 0,5) and greater than *rLowFreq*.
pTaps	Pointer to the array where computed tap values are stored. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values; should be equal or greater than 5.

winType Values of this argument specify what type of window will be used in computations. The *winType* must have one of the following values:

ippWinBartlett	Bartlett window;
ippWinBlackman	Blackman window;
ippWinHamming	Hamming window;
ippWinHann	Hann window.

doNormal Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The *doNormal* must have one of the following values:

ippTrue	The function computes normalized sequence of coefficients.
ippFalse	The function computes non-normalized sequence of coefficients.

Discussion

The function ippsFIRGenBandpass is declared in the ipps.h file. This function computes *tapsLen* coefficients for bandpass FIR filter with the cut-off frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pTaps* pointer is NULL.
ippStsSizeErr	Indicates an error when the *tapsLen* is less than 5, or *rLowFreq* is greater than or equal to *rHighFreq*, or one of the frequency parameters *rLowFreq* and *rHighFreq* is out of the range.

FIRGenBandstop

Computes bandstop FIR filter coefficients.

```
IppStatus ippsFIRGenBandstop_64f(Ipp64f rLowFreq, Ipp64f rHighFreq,
      Ipp64f* pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

Arguments

rLowFreq	Normalized low cut-off frequency, should be in the range (0, 0,5) and less than *rHighFreq*.
rHighFreq	Normalized high cut-off frequency, should be in the range (0, 0,5) and greater than *rLowFreq*.
pTaps	Pointer to the array where computed tap values are stored. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values, should be equal or greater than 5.
winType	Values of this argument specify what type of window will be used in computations. The *winType* must have one of the following values:

`ippWinBartlett`	Bartlett window;
`ippWinBlackman`	Blackman window;
`ippWinHamming`	Hamming window;
`ippWinHann`	Hann window.

doNormal	Values of this argument specify whether normalized or non-normalized sequence of the filter coefficients will be computed. The *doNormal* must have one of the following values:

`ippTrue`	The function computes normalized sequence of coefficients.
`ippFalse`	The function computes non-normalized sequence of coefficients.

Discussion

The function ippsFIRGenBandstop is declared in the ipps.h file. This function computes *tapsLen* coefficients for bandstop FIR filter with the cut-off frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pTaps* pointer is NULL.
ippStsSizeErr	Indicates an error when the *tapsLen* is less than 5, or *rLowFreq* is greater than or equal to *rHighFreq*, or one of the frequency parameters *rLowFreq* and *rHighFreq* is out of the range.

Single-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a single-rate FIR LMS filter
- get and set the delay line values
- get the filter coefficients (taps) values
- perform filtering
- free dynamic memory allocated for the functions state

To use the single-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call ippsFIRLMSInitAlloc to allocate memory and initialize a single-rate FIR LMS filter.
2. Call ippsFIRLMSOne_Direct to make one iteration of FIR filter taps fitting with one input sample and/or call ippsFIRLMS to fit taps with a block of consecutive input samples.

3. Call ippsFIRLMSGetTaps to get the filter coefficients (taps). Call ippsFIRLMSGetDlyLine and ippsFIRLMSSetDlyLine to get and set the values in the delay line.
4. Call ippsFIRLMSFree to release dynamic memory associated with the FIR LMS filter.

FIRLMSInitAlloc

Allocates memory and initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.

```
IppStatus ippsFIRLMSInitAlloc_32f(IppsFIRLMSState_32f** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp32f* pDlyLine, int
     dlyLineIndex);
IppStatus ippsFIRLMSInitAlloc32f_16s(IppsFIRLMSState32f_16s** pState,
     const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine, int
     dlyLineIndex);
```

Arguments

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is 2**tapsLen*.
dlyLineIndex	Current index of the delay line.
pState	Address of the pointer to the state structure to be created.

Discussion

The function ippsFIRLMSInitAlloc is declared in the ipps.h file. This function allocates memory and initializes a single-rate FIR LMS filter state. The function ippsFIRLMSInitAlloc copies the taps from the *tapsLen* -length array *pTaps* into the

state structure *pState*. The 2**tapsLen* -length array *pDlyLine* specifies the delay line values. The current index of the delay line *pDlyLine* is defined by *dlyLineIndex*. If the pointer to the array *pDlyLine* or *pTaps* is NULL, then the corresponding values of the state structure are initialized to 0.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSFree

Closes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.

```
IppStatus ippsFIRLMSFree_32f(IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMSFree32f_16s(IppsFIRLMSState32f_16s* pState);
```

Arguments

pState	Pointer to the FIR LMS filter state structure to be closed.

Discussion

The function ippsFIRLMSFree is declared in the ipps.h file. This function closes the FIR LMS filter state by freeing all memory associated with a filter state created by ippsFIRLMSInitAlloc. Call ippsFIRLMSFree after filtering is completed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSGetTaps

Gets the taps of a FIR LMS filter.

```
IppStatus ippsFIRLMSGetTaps_32f(const IppsFIRLMSState_32f* pState,
     Ipp32f* pOutTaps);
IppStatus ippsFIRLMSGetTaps32f_16s(const IppsFIRLMSState32f_16s*
     pState, Ipp32f* pOutTaps);
```

Arguments

pState	Pointer to the FIR LMS filter state structure.
pOutTaps	Pointer to the array holding copies of the taps.

Discussion

The function ippsFIRLMSGetTaps is declared in the ipps.h file. This function copies the taps from the state structure *pState* to the *tapsLen* -length array *pOutTaps*. To set new taps in the state structure, create a new state using the function ippsFIRLMSInitAlloc.

Before calling the function ippsFIRLMSGetTaps, initialize the filter state by calling ippsFIRLMSInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSGetDlyLine, FIRLMSSetDlyLine

Gets and sets the delay line contents of a FIR LMS filter.

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState,
     Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s*
     pState, Ipp16s* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine_32f(IppsFIRLMSState_32f* pState,
     const Ipp32f* pDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine32f_16s(IppsFIRLMSState32f_16s* pState,
     const Ipp16s* pDlyLine, int dlyLineIndex);
```

Arguments

pState	Pointer to the FIR LMS filter state structure.
pDlyLine	Pointer to the *tapsLen* -length array holding the delay line values.
pDlyLineIndex	Pointer to the array to store the current delay line index copied from *pState* by the function ippsFIRLMSGetDlyLine.
dlyLineIndex	Initial index of the delay line to be stored in *pState* by the function ippsFIRLMSSetDlyLine.

Discussion

The functions ippsFIRLMSGetDlyLine and ippsFIRLMSSetDlyLine are declared in the ipps.h file. These functions get and set the delay line values of a FIR LMS filter state.

ippsFIRLMSGetDlyLine. The function ippsFIRLMSGetDlyLine copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pDlyLine* and *pDlyLineIndex*, respectively.

ippsFIRLMSSetDlyLine. The function ippsFIRLMSSetDlyLine copies the delay line values from *pDlyLine* and the current delay line index from *dlyLineIndex*, and stores them into the state structure *pState*.

Before calling either ippsFIRLMSGetDlyLine or ippsFIRLMSSetDlyLine, initialize the filter state by calling the function ippsFIRLMSInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMS

Filters an array through a FIR LMS filter.

```
IppStatus ippsFIRLMS_32f(const Ipp32f* pSrc, const Ipp32f* pRef,
     Ipp32f* pDst, int len, float mu, IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMS32f_16s(const Ipp16s* pSrc, const Ipp16s* pRef,
     Ipp16s* pDst, int len, float mu, IppsFIRLMSState32f_16s* pState);
```

Arguments

pState	Pointer to the FIR LMS filter state structure.
pSrc	Pointer to the input sample to be filtered.
pRef	Pointer to the reference signal
pDst	Pointer to the output signal
len	Number of elements in the array.
mu	Adaptation step.

Discussion

The function ippsFIRLMS is declared in the ipps.h file. This function filters an input array *pSrc* using an adaptive FIR LMS filter.

Each of *len* iterations performed by the function consists of two main procedures. First, ippsLMS filters the current sample of the input signal *pSrc* and stores the result in *pDst*. Next, the function updates the current taps using the reference signal *pRef*, the computed result signal *pDst*, and the adaptation step *mu*.

The filtering procedure can be described as a FIR filter operation:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Here the input sample to be filtered is denoted by *x(n)*, the taps are denoted by *h(i)*, and *y(n)* is the return value.

The function updates the filter coefficients that are stored in the filter state structure *pState*. Updated filter coefficients are defined as

$$h_{n+1}(i) = h_n(i) + 2 \cdot mu \cdot errVal \cdot x(n-i)$$

where $h_{n+1}(i)$ denotes new taps, $h_n(i)$ denotes initial taps, *mu* and *errVal* are the adaptation step and adaptation error value, respectively. An adaptation error value *errVal* is computed inside the function as the difference between the output and reference signals.

Before using ippsFIRLMS, initialize the *pState* structure by calling the ippsFIRLMSInitAlloc function.

Example 6-4 illustrates the use of the function ippsFIRLMS_32f to filter a signal sample.

Example 6-4 Filtering with the ippsFIRLMS Function

```
IppStatus firlms(void) {
     IppStatus st;
     Ipp32f taps = 0, x[LEN], y[LEN], mu = 0.03f;
     IppsFIRLMSState_32f* ctx;
     int i;
     /// no taps and no delay line from outside
     ippsFIRLMSInitAlloc_32f( &ctx, 0, 1, 0, 0 );
     /// make a const signal of amplitude 1 and noise it
     for(i=0; i<LEN; ++i) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
     st = ippsFIRLMS_32f( x, x+1, y, LEN-1, mu, ctx);
     /// get FIR LMS tap, it must be near to 1
     ippsFIRLMSGetTaps_32f(ctx, &taps);
     ippsFIRLMSFree_32f(ctx);
     printf_32f("FIR LMS tap fitted =", &taps, 1, st);
     return st;
}
Output:
    FIR LMS tap adapted = 0.986842
```

Return Value

ippStsNoErr	Indicates 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.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSOne_Direct

Filters a single sample through a FIR LMS filter.

```
IppStatus ippsFIRLMSOne_Direct_32f(Ipp32f src, Ipp32f refVal,
     Ipp32f* pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu,
     Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSOne_Direct32f_16s(Ipp16s src, Ipp16s refVal,
     Ipp16s* pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu,
     Ipp16s* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSOne_DirectQ15_16s(Ipp16s src, Ipp16s refVal,
     Ipp16s* pDstVal, Ipp32s* pTapsInv, int tapsLen, int muQ15,
     Ipp16s* pDlyLine, int* pDlyLineIndex);
```

Arguments

src	Input sample to be filtered.
pDstVal	Pointer to the output sample.
refVal	Reference signal sample.
pTapsInv	Pointer to the array containing the FIR filter taps to be adapted. The tap values are stored in the array in the inverse order.
tapsLen	Number of elements in the array containing the tap values.
pDlyLine	Pointer to the array holding the delay line values.
pDlyLineIndex	Pointer to the current index of the delay line.
mu	Adaptation step.
muQ15	Integer version adaptation step.

Discussion

The function ippsFIRLMSOne_Direct is declared in the ipps.h file. This function performs directly a single iteration of FIR filter taps adaptation. The *tapsLen* -length array *pTapsInv* contains the FIR filter taps in the inverse order. The 2**tapsLen* -length array *pDlyLine* specifies the delay line values. The *pDlyLineIndex* array specifies the current index of the delay line. The output signal is stored in *pDstVal*.

NOTE. *The adaptation error value can be computed as follows:* *err*[*n*] = *refVal*[*n*] - **pDstVal*.

The function ippsFIRLMSOne_Direct performs a single iteration of FIR filter taps adaptation with the *mu* step value. The taps are floating-point numbers.

Set the taps to zero or to values close to calculated to speed up the process.

The function ippsLMSOne_Direct is to be called within cycle with the number of iterations equal to the number of input samples. You can decide what data is to be saved as a result of adaptation: either the filtered output signal or the adaptation error.

The function ippsFIRLMSOne_DirectQ15 performs a single iteration of FIR filter taps adaptation with the *muQ15* step value. The taps are integer numbers. The adaptation step *muQ15* can be computed as follows:

muQ15 = (int)(*mu* *(1<<15)+0.5f)

Example 6-5 illustrates the use of the function ippsFIRLMSOne_Direct to adapt the FIR filter taps. After adaptation the taps are close to 1.0.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsSizeErr	Indicates an error when *tapsLen* is less or equal to 0.

Example 6-5 Usage of the ippsLMSOne_Direct Function

```
IppStatus firlmsone(void) {
#define LEN 200
      IppStatus st = ippStsNoErr;
      Ipp32f taps = 0, dly[2] = {0};
      Ipp32f x[LEN], y[LEN], mu = 0.05f;
      int i, indx = 0;
      /// make a const signal of amplitude 1 and noise it
      for( i=0; i<LEN; ++i ) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
      for( i=0; i<LEN-1 && ippStsNoErr==st; ++i )
      st=ippsFIRLMSOne_Direct_32f( x[i], x[i+1], y+1+i, &taps, 1, mu,
          dly, &indx );
      printf_32f("FIRLMSOne tap adapted =", &taps, 1, st );
      return st;
}
Output:
    FIRLMSOne tap adapted = 0.993872
```

Multi-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a multi-rate FIR LMS filter
- get and set the delay line values
- get and set the filter coefficients (taps) values
- set the adaptation step value
- perform filtering
- update the filter coefficients using the result of the filter operation
- free dynamic memory allocated for the functions state

To use the multi-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call `ippsFIRLMSMRInitAlloc` to initialize a multi-rate FIR LMS filter.
2. Call `ippsFIRLMSMRPutVal` a required number of times to place the input values in the delay line.
3. Call `ippsFIRLMSMROne` to filter the samples in the delay line.
4. Call ipps`FIRLMSMRUpdateTaps` to update the taps using the value of adaptation error that is based on comparison between filtered and reference signals.
5. Call `ippsFIRLMSMRGetTaps` and `ippsFIRLMSMRSetTaps` to get and set the filter coefficients (taps). Call `ippsFIRLMSMRGetDlyLine` and `ippsFIRLMSMRSetDlyLine` to get and set the values in the delay line.
6. Call `ippsFIRLMSMRFree` to release dynamic memory associated with the FIR LMS filter.

FIRLMSMRInitAlloc

Allocates memory and initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.

```
IppStatus ippsFIRLMSMRInitAlloc32s_16s(IppsFIRLMSMRState32s_16s**
     pState, const Ipp32s* pTaps, int tapsLen, const Ipp16s* pDlyLine,
     int dlyLineIndex, int dlyStep, int updateDly, int mu);
IppStatus ippsFIRLMSMRInitAlloc32sc_16sc(IppsFIRLMSMRState32sc_16sc**
     pState, const Ipp32sc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
     int dlyLineIndex, int dlyStep, int updateDly, int mu);
```

Arguments

pTaps	Pointer to the array containing the tap values. The number of elements in the array is *tapsLen*.
tapsLen	Number of elements in the array containing the tap values.

pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is *tapsLen*dlyStep + updateDly*.
dlyLineIndex	Current index of the delay line.
dlyStep	Multi-rate down factor applied to delay line values.
updateDly	Value of adaptation delay in samples.
mu	Adaptation step.
pState	Address of the pointer to the filter state structure to be created.

Discussion

The function ippsFIRLMSMRInitAlloc is declared in the ipps.h file. This function allocates memory and initializes a multi-rate FIR LMS filter state. The function copies the filter coefficients from *tapsLen*-length array *pTaps* into the state structure *pState*. The array *pDlyLine* specifies the delay line values. The structure is initialized by the downsampling factor over the delay line *dlyStep*, by the adaptation delay value *updateDly*, and by the adaptation step value *mu*. The function returns the pointer in the output parameter *pState* and also the operation status value.

The function uses copies of the tap values during the filtering procedure. The initial values passed to the function may be obtained from the previous filtering procedure.If the pointer *pTaps* is NULL, then the filter coefficient values are set to zero.

Copies of the input samples are used in the filtering procedure. Values in the delay line represent data in the same format as the input data to be filtered. If the pointer *pDlyLine* is NULL, then the filter delay line values are set to zero.

Return Value

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

FIRLMSMRFree

Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.

```
IppStatus ippsFIRLMSMRFree32s_16s(IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRFree32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the multi-rate FIR LMS filter state structure to be closed.

Discussion

The function `ippsFIRLMSMRFree` is declared in the `ipps.h` file. This function closes the multi-rate FIR LMS filter state by freeing all memory associated with a filter state created by `ippsFIRLMSMRInitAlloc`. Call `ippsFIRLMSMRFree` after filtering is completed.

Return Value

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

FIRLMSMRSetMu

Sets the adaptation step.

```
IppStatus ippsFIRLMSMRSetMu32s_16s(IppsFIRLMSMRState32s_16s* pState,
     const int mu);
IppStatus ippsFIRLMSMRSetMu32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
     const int mu);
```

Arguments

mu	New adaptation step
pState	Pointer to the filter state structure.

Discussion

The function ippsFIRLMSMRSetMu is declared in the ipps.h file. This function updates the adaptation step stored in *pState* with the new value *mu*.

Before calling the function ippsFIRLMSMRSetMu, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

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

FIRLMSMRUpdateTaps

Updates the filter coefficients using the adaptation error value.

```
IppStatus ippsFIRLMSMRUpdateTaps32s_16s(Ipp32s errVal,
     IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRUpdateTaps32sc_16sc(Ipp32sc errVal,
     IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the filter state structure.
errVal	Adaptation error value.

Discussion

The function ippsFIRLMSMRUpdateTaps is declared in the ipps.h file. This function updates the filter coefficients that are stored in the filter state structure *pState*. It is assumed that the filter operation was performed and the adaptation error value *errVal* was computed before calling this function. The adaptation error value is computed outside the function as the difference between output and reference signals. Updated filter coefficients are defined as

$$h_{n+1}(i) = h_n(i) + mu \cdot errVal \cdot x(n - (i \cdot dlyStep) - updateDly)$$

where $h_{n+1}(i)$ denotes new taps, $h_n(i)$ denotes initial taps, and *mu*, *errVal* and *updateDly* are the adaptation step, adaptation error value and adaptation delay, respectively.
Before calling the function ippsFIRLMSMRUpdateTaps, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

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

FIRLMSMRGetTaps, FIRLMSMRSetTaps

Gets and sets taps of a multi-rate FIR LMS filter.

```
IppStatus ippsFIRLMSMRGetTaps32s_16s(
     IppsFIRLMSMRState32s_16s* pState, Ipp32s* pOutTaps);
IppStatus ippsFIRLMSMRGetTaps32sc_16sc(
     IppsFIRLMSMRState32sc_16sc* pState, Ipp32sc* pOutTaps);
IppStatus ippsFIRLMSMRSetTaps32s_16s(
     IppsFIRLMSMRState32s_16s* pState, const Ipp32s* pInTaps);
```

```
IppStatus ippsFIRLMSMRSetTaps32sc_16sc(
     IppsFIRLMSMRState32sc_16sc* pState, const Ipp32sc* pInTaps);
```

Arguments

pState	Pointer to the filter state structure.
pOutTaps	Pointer to the array holding copies of the taps.
pInTaps	Pointer to the array holding new tap values.

Discussion

The functions ippsFIRLMSMRGetTaps and ippsFIRLMSMRSetTaps are declared in the ipps.h file. These functions get and set the filter coefficients.

ippsFIRLMSMRGetTaps. The ippsFIRLMSMRGetTaps function copies the values of the filter coefficients stored in the filter state structure *pState* to the array pointed by the *pOutTaps* pointer.

ippsFIRLMSMRSetTaps. The ippsFIRLMSMRSetTaps function sets the the filter coefficients stored in the filter state structure *pState* to the new values stored in an array pointed by the *pInTaps* pointer. If the pointer is NULL, then the filter coefficients values are set to zero.

Before calling either ippsFIRLMSMRGetTaps or ippsFIRLMSMRSetTaps, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

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

FIRLMSMRGetTapsPointer

Returns the pointer to the filter coefficients.

```
IppStatus ippsFIRLMSMRGetTapsPointer32s_16s(
     IppsFIRLMSMRState32s_16s* pState, Ipp32s** pTaps);
IppStatus ippsFIRLMSMRGetTapsPointer32sc_16sc(
     IppsFIRLMSMRState32sc_16sc* pState, Ipp32sc** pTaps);
```

Arguments

pState	Pointer to the filter state structure.
pTaps	Pointer to the variable that contains the pointer to the tap values.

Discussion

The function ippsFIRLMSMRGetTapsPointer is declared in the ipps.h file. This function writes the pointer to the filter coefficients stored in the filter state structure *pState* to the variable pointed by *pTaps*.

CAUTION. *To get the pointer to tap values directly is faster than to copy them using the* ippsFIRLMSMRGetTaps *function, but this operation may be error-prone.*

Before calling the function ippsFIRLMSMRGetTapsPointer, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

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

FIRLMSMRGetDlyLine, FIRLMSMRSetDlyLine

Gets and sets the delay line contents of a multi-rate FIR LMS filter state.

```
IppStatus ippsFIRLMSMRGetDlyLine32s_16s(IppsFIRLMSMRState32s_16s*
     pState, Ipp16s* pOutDlyLine, int* pOutDlyLineIndex);
IppStatus ippsFIRLMSMRGetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc*
     pState, Ipp16sc* pOutDlyLine, int* pOutDlyLineIndex);
IppStatus ippsFIRLMSMRSetDlyLine32s_16s(IppsFIRLMSMRState32s_16s*
     pState, const Ipp16s* pInDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSMRSetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc*
     pState, const Ipp16sc* pInDlyLine, int dlyLineIndex);
```

Arguments

pState	Pointer to the filter state structure.
pOutDlyLine	Pointer to the array containing the copies of delay line values. The number of elements in the array is *tapsLen*dlyStep+updateDly*.
pInDlyLine	Pointer to the array containing the new delay line values.The number of elements in the array is *tapsLen*dlyStep+updateDly*.
pOutDlyLineIndex	Pointer to the array to store the current delay line index copied from *pState*.
dlyLineIndex	Initial index of the delay line to be stored in *pState* .

Discussion

The function `ippsFIRLMSMRGetDlyLine` and `ippsFIRLMSMSetDlyLine` are declared in the `ipps.h` file. These functions get and set the delay line values of a multi-rate FIR LMS filter state.

ippsFIRLMSMRGetDlyLine. The ippsFIRLMSMRGetDlyLine function copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pOutDlyLine* and *pOutDlyLineIndex*, respectively.

ippsFIRLMSMRSetDlyLine. The ippsFIRLMSMRSetDlyLine function copies the new values stored in the *pInDlyLine* and corresponding delay line index from *dlyLineIndex* and stores them into the state structure *pState*. If the pointer *pInDlyLine* is NULL, the delay line values are set to zero.

Before calling either ippsFIRLMSMRGetDlyLine or ippsFIRLMSMRSetDlyLine, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the *pState*, *pOutDlyLine*, *pOutDlyLineIndex* pointers is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMRGetDlyVal

Gets one delay line values from the specified position.

```
IppStatus ippsFIRLMSMRGetDlyVal32s_16s(IppsFIRLMSMRState32s_16s*
     pState, Ipp16s* pOutVal, int index);
IppStatus ippsFIRLMSMRGetDlyVal32sc_16sc(IppsFIRLMSMRState32sc_16sc*
     pState, Ipp16sc* pOutVal, int index);
```

Arguments

pState	Pointer to the filter state structure.
pOutVal	Pointer to the copied delay line value.
index	Index of the required delay line value.

Discussion

The function ippsFIRLMSMRGetDlyVal is declared in the ipps.h file. This function copies from the filter state structure *pState* one value to the *pOurVal*. The position of this sample in the delay line is specified by *index* (which means that *index* iterations ago the sample was placed into the delay line).

Before calling the function ippsFIRLMSMRGetDlyVal, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

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

FIRLMSMRPutVal

Places the input value in the delay line.

```
IppStatus ippsFIRLMSMRPutVal32s_16s(Ipp16s val,
     IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRPutVal32sc_16sc(Ipp16sc val,
     IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the filter state structure.
val	Value of the input sample.

Discussion

The function ippsFIRLMSMRPutVal is declared in the ipps.h file. This function places the value of the input sample *val* into the delay line, thus preparing the filter with the state structure *pState* for the filtering procedure.

Before calling the ippsFIRLMSMRPutVal, initialize the filter state by calling the function ippsFIRLMSMRInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSMROne

Filters data placed in the delay line.

```
IppStatus ippsFIRLMSMROne32s_16s(Ipp32s* pDstVal,
     IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMROne32sc_16sc(Ipp32sc* pDstVal,
     IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the filter state structure.
pDstVal	Pointer to the output signal value.

Discussion

The function FIRLSMMROne is declared in the ipps.h file. This function filters the samples placed in the delay line using the filter coefficients stored in the filter state structure *pState*. The resulting value is placed in the *pDstVal*. The downsampling factor *dlyStep* defines the number of samples that are filtered. The filter coefficients are not updated.

The filtering procedure can be described as a FIR filter operation (here the input sample to be filtered is denoted *x(n)*, the taps are denoted *h(i)*, and the return value is *y(n)*):

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-(i \cdot dlyStep))$$

Note that the function operates with values stored in the delay line that are copies of the input samples.
Before calling the function `ippsFIRLMSMROne`, initialize the filter state by calling `ippsFIRLMSMRInitAlloc`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState* or *pDstVal* pointers are `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSMROneVal

Filters one input value.

```
IppStatus ippsFIRLMSMROneVal32s_16s(Ipp16s val, Ipp32s* pDstVal,
     IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMROneVal32sc_16sc(Ipp16sc val, Ipp32sc* pDstVal,
     IppsFIRLMSMRState32sc_16sc* pState);
```

Arguments

pState	Pointer to the filter state structure.
pDstVal	Pointer to the output signal value.
val	Value of the input signal sample.

Discussion

The function ippsFIRLMSMROneVal is declared in the ipps.h file. This function places one input sample *val* into the delay line and filters it using the filter coefficients specified in the filter state structure *pState*. The result value is stored in the *pDstVal*.

Before calling the function ippsFIRLMSMROneVal, initialize the filter state by calling ippsFIRLMSMRInitAlloc.

Return Value

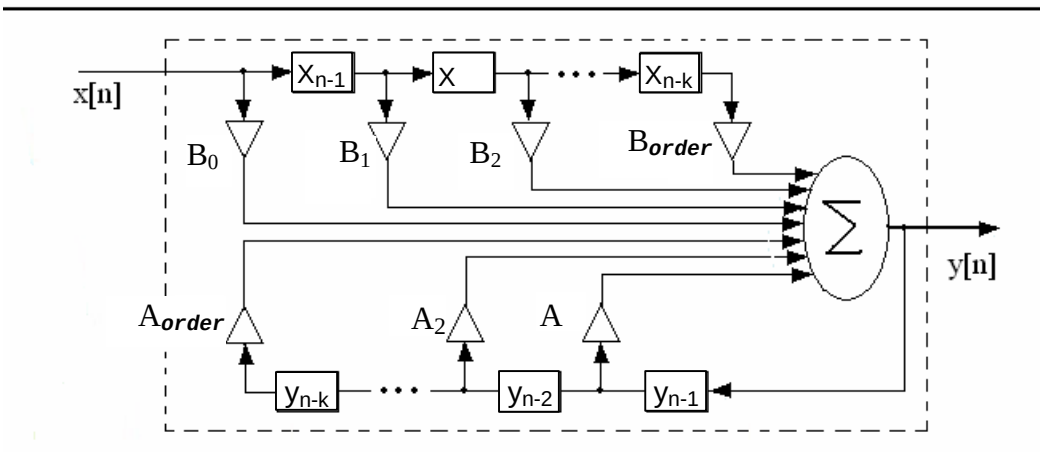
ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pState* or *pDstVal* pointers are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

IIR Filter Functions

The functions described in this section initialize an infinite impulse response (IIR) filter and perform filtering. Intel IPP supports two types of filters: arbitrary order filter and biquad filter.

Figure 6-1 shows the structure of an arbitrary order IIR filter.

Figure 6-1 Structure of an Arbitrary Order Filter



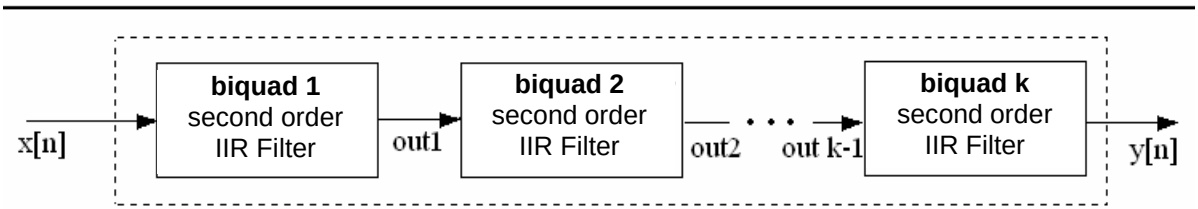
Here `x[n]` is a sample of the input signal, `y[n]` is a sample of the output signal, *order* is the filter order, and $B_0, B_1, \ldots, B_{order}$, $A_1, \ldots, A_{order}$ are the constant coefficients (taps).

The output signal is computed by the following formula:

$$y[n] = \sum_{k=0}^{order} B_k \cdot x(n-k) + \sum_{k=1}^{order} A_k \cdot x(n-k)$$

A biquad IIR filter is a cascade of second-order filters. The Figure 6-2 illustrates the structure of the biquad filter with **k** cascades of second-order filters.

Figure 6-2 Structure of a Biquad IIR Filter



To initialize and use an IIR filter, follow this general scheme:

1. Call `ippsIIRInitAlloc` to allocate memory and initialize the filter as an arbitrary order IIR filter, or call `ippsIIRInitAlloc_BiQuad` to allocate memory and initialize the filter as a cascade of biquads.
   Or alternatively call either `ippsIIRInit` to initialize the filter as an arbitrary order IIR filter in the external buffer, or `ippsIIRInitBiQuad` to initialize the filter as a cascade of biquads in the external buffer. Size of the buffer can be computed by calling the functions `ippsIIRGetStateSize` or `ippsIIRMRGetStateSize_BiQuad`, respectively.
2. Call `ippsIIROne` repeatedly to filter a single sample through an IIR filter or call `ippsIIR` to filter consecutive samples at once.
3. Call `ippsIIRGetDlyLine` and `ippsIIRSetDlyLine` to get and set the delay line values in the IIR state structure.
4. Call `ippsIIRSetTaps` to set new tap values in the previously initialized filter state structure.
5. After all filtering is complete, call `ippsIIRFree` to release dynamic memory associated with the filter state structure created by `ippsIIRInitAlloc` or `ippsIIRInitAlloc_BiQuad`.

Alternatively, you may use the direct version of the functions. These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

IIRInitAlloc, IIRInitAlloc_BiQuad

Allocates memory and initializes an infinite impulse response filter state.

```
IppStatus ippsIIRInitAlloc_32f(IppsIIRState_32f** pState,
     const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc_32fc(IppsIIRState_32fc** pState,
     const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc_64f(IppsIIRState_64f** pState,
     const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_64fc(IppsIIRState_64fc** pState,
     const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc32s_16s(IppsIIRState32s_16s** pState,
     const Ipp32s* pTaps, int order, int tapsFactor, const Ipp32s*
     pDlyLine);
IppStatus ippsIIRInitAlloc32s_16s32f(IppsIIRState32s_16s** pState,
     const Ipp32f* pTaps, int order, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_16sc(IppsIIRState32sc_16sc** pState,
     const Ipp32sc* pTaps, int order, int tapsFactor,
     const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_16sc32fc(IppsIIRState32sc_16sc** pState,
     const Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32f_16s(IppsIIRState32f_16s** pState,
     const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc32fc_16sc(IppsIIRState32fc_16sc** pState,
     const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc64f_16s(IppsIIRState64f_16s** pState,
     const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32s(IppsIIRState64f_32s** pState,
     const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32f(IppsIIRState64f_32f** pState,
     const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_16sc(IppsIIRState64fc_16sc** pState,
     const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32sc(IppsIIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32fc(IppsIIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc_BiQuad_32f(IppsIIRState_32f** pState,
     const Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_32fc(IppsIIRState_32fc** pState,
     const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_64f(IppsIIRState_64f** pState,
     const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_64fc(IppsIIRState_64fc** pState,
     const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc32s_BiQuad_16s(IppsIIRState32s_16s** pState,
     const Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32s_BiQuad_16s32f(IppsIIRState32s_16s**
     pState, const Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc(IppsIIRState32sc_16sc**
     pState, const Ipp32sc* pTaps, int numBq, int tapsFactor,
     const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc**
     pState, const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32f_BiQuad_16s(IppsIIRState32f_16s** pState,
     const Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc32fc_BiQuad_16sc(IppsIIRState32fc_16sc**
     pState, const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);


IppStatus ippsIIRInitAlloc64f_BiQuad_16s(IppsIIRState64f_16s** pState,
     const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_BiQuad_32s(IppsIIRState64f_32s** pState,
     const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_BiQuad_32f(IppsIIRState64f_32f** pState,
     const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_16sc(IppsIIRState64fc_16sc**
     pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_32sc(IppsIIRState64fc_32sc**
     pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_32fc(IppsIIRState64fc_32fc**
     pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

Arguments

pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1) for arbitrary filters and 6**numBq* for BQ filters.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only)
numBq	Number of cascades of biquads. This argument is used for BQ filters.
order	Order of the IIR filter. This argument is used for arbitrary filters.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters.
pState	Pointer to the IIR state structure to be created.

Discussion

The functions ippsIIRInitAlloc and ippsIIRInitAlloc_BiQuad are declared in the ipps.h file. These functions allocate memory and initialize an arbitrary or biquad (BQ) IIR filter state, respectively. The initialization functions copy the taps from the array *pTaps* into the state structure *pState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

If the state is not created, the initialization function returns an error status.

The initialization functions with the 32s_32f suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. Example 6-7 shows how to convert floating-point taps into integer data type.

ippsIIRInitAlloc. The function ippsIIRInitAlloc initializes the taps and the delay line in the state structure of an arbitrary order IIR filter. The *order* -length array *pDlyLine* specifies the delay line values. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The 2*(*order* + 1) -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

$A_0 \neq 0$

ippsIIRInitAlloc_BiQuad. The function ippsIIRInitAlloc_BiQuad initializes the taps and the delay line in the state structure of a biquad IIR filter; that is, defined by a cascade of biquads. The 2**numBq* -length array *pDlyLine* specifies the delay line values. The number of cascades of biquads is defined by the *numBq* value. The 6**numBq* -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \ldots A_{numBq\text{-}1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

Return Value

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory allocated.

`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsIIROrderErr`	Indicates an error when *`order`* is less than 0, or *`numBq`* is less than 1.
`ippStsDivByZeroErr`	Indicates an error when A_0, $A_{n,0}$ or $B_{n,0}$ is equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

IIRFree

Closes an IIR filter state.

```
IppStatus ippsIIRFree_32f(IppsIIRState_32f* pState);
IppStatus ippsIIRFree_64f(IppsIIRState_64f* pState);
IppStatus ippsIIRFree_32fc(IppsIIRState_32fc* pState);
IppStatus ippsIIRFree_64fc(IppsIIRState_64fc* pState);
IppStatus ippsIIRFree32s_16s(IppsIIRState32s_16s* pState);
IppStatus ippsIIRFree32sc_16sc(IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRFree32f_16s(IppsIIRState32f_16s* pState);
IppStatus ippsIIRFree32fc_16sc(IppsIIRState32fc_16sc* pState);
IppStatus ippsIIRFree64f_16s(IppsIIRState64f_16s* pState);
IppStatus ippsIIRFree64f_32s(IppsIIRState64f_32s* pState);
IppStatus ippsIIRFree64f_32f(IppsIIRState64f_32f* pState);
IppStatus ippsIIRFree64fc_16sc(IppsIIRState64fc_16sc* pState);
IppStatus ippsIIRFree64fc_32sc(IppsIIRState64fc_32sc* pState);
IppStatus ippsIIRFree64fc_32fc(IppsIIRState64fc_32fc* pState);
```

Arguments

`pState`	Pointer to an IIR filter state structure to be closed.

Discussion

The function ippsIIRFree is declared in the ipps.h file. This function closes the IIR filter state by freeing all memory associated with a filter state created by ippsIIRInitAlloc or ippsIIRInitAlloc_BiQuad. Call ippsIIRFree after filtering is completed.

Return Value

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

IIRInit, IIRInit_BiQuad

Initializes an IIR filter state.

```
IppStatus ippsIIRInit_32f(IppsIIRState_32f** pState, const Ipp32f*
     pTaps, int order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_32fc(IppsIIRState_32fc** pState, const Ipp32fc*
     pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_64f(IppsIIRState_64f** pState, const Ipp64f*
     pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_64fc(IppsIIRState_64fc** pState, const Ipp64fc*
     pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32s_16s(IppsIIRState32s_16s** pState, const
     Ipp32s* pTaps, int order, int tapsFactor, const Ipp32s* pDlyLine,
     Ipp8u* pBuffer);
IppStatus ippsIIRInit32s_16s32f(IppsIIRState32s_16s** pState, const
     Ipp32f* pTaps, int order, const Ipp32s* pDlyLine, Ipp8u*
     pBuffer);
```

```
IppStatus ippsIIRInit32sc_16sc(IppsIIRState32sc_16sc** pState, const
     Ipp32sc* pTaps, int order, int tapsFactor, const Ipp32sc*
     pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32sc_16sc32fc(IppsIIRState32sc_16sc** pState,
     const Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine, Ipp8u*
     pBuffer);

IppStatus ippsIIRInit32f_16s(IppsIIRState32f_16s** pState, const
     Ipp32f* pTaps, int order, const Ipp32f* pDlyLine, Ipp8u*
     pBuffer);
IppStatus ippsIIRInit32fc_16sc(IppsIIRState32fc_16sc** pState, const
     Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u*
     pBuffer);
IppStatus ippsIIRInit64f_16s(IppsIIRState64f_16s** pState, const
     Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u*
     pBuffer);
IppStatus ippsIIRInit64f_32s(IppsIIRState64f_32s** pState, const
     Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u*
     pBuffer);
IppStatus ippsIIRInit64f_32f(IppsIIRState64f_32f** pState, const
     Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u*
     pBuffer);

IppStatus ippsIIRInit64fc_16sc(IppsIIRState64fc_16sc** pState, const
     Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_32sc(IppsIIRState64fc_32sc** pState, const
     Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_32fc(IppsIIRState64fc_32fc** pState, const
     Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_32f(IppsIIRState_32f** pState, const
     Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_BiQuad_32fc(IppsIIRState_32fc** pState, const
     Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_BiQuad_64f(IppsIIRState_64f** pState, const
     Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsIIRInit_BiQuad_64fc(IppsIIRState_64fc** pState, const
      Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32s_BiQuad_16s(IppsIIRState32s_16s** pState,
      const Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine,
      Ipp8u* pBuffer);
IppStatus ippsIIRInit32sc_BiQuad_16sc(IppsIIRState32sc_16sc** pState,
      const Ipp32sc* pTaps, int numBq, int tapsFactor, const Ipp32sc*
      pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32s_BiQuad_16s32f(IppsIIRState32s_16s** pState,
      const Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine, Ipp8u*
      pBuffer);
IppStatus ippsIIRInit32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc**
      pState, const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine,
      Ipp8u* pBuffer);

IppStatus ippsIIRInit32f_BiQuad_16s(IppsIIRState32f_16s** pState,
      const Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u*
      pBuffer);
IppStatus ippsIIRInit32fc_BiQuad_16sc(IppsIIRState32fc_16sc** pState,
      const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u*
      pBuffer);

IppStatus ippsIIRInit64f_BiQuad_16s(IppsIIRState64f_16s** pState,
      const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine,
      Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_BiQuad_32s(IppsIIRState64f_32s** pState,
      const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine,
      Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_BiQuad_32f(IppsIIRState64f_32f** pState,
      const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u*
      pBuffer);
IppStatus ippsIIRInit64fc_BiQuad_16sc(IppsIIRState64fc_16sc** pState,
      const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine,
      Ipp8u* pBuffer);
```

```
IppStatus ippsIIRInit64fc_BiQuad_32sc(IppsIIRState64fc_32sc** pState,
     const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u*
     pBuffer);
IppStatus ippsIIRInit64fc_BiQuad_32fc(IppsIIRState64fc_32fc** pState,
     const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u*
     pBuffer);
```

Arguments

pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1) for arbitrary filters and 6**numBq* for BQ filters.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).
numBq	Number of cascades of biquads. This argument is used for BQ filters.
order	Order of the IIR filter. This argument is used for arbitrary filters.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters.
pState	Pointer to the IIR state structure to be created.
pBuffer	Pointer to the external buffer for the IIR state structure.

Discussion

The functions ippsIIRInit and ippsIIRInit_BiQuad are declared in the ipps.h file. These functions initialize an arbitrary or biquad (BQ) IIR filter state, respectively, in the external buffer. The size of this buffer should be computed previously by calling the functions IIRGetStateSize and IIRGetStateSize_BiQuad. The initialization functions copy the taps from the array *pTaps* into the state structure *pState*. To scale integer taps, use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. Example 6-7 shows how to convert floating-point taps into integer data type.

`ippsIIRInit`. The function `ippsIIRInit` initializes the taps and the delay line in the state structure of an arbitrary order IIR filter. The *`order`* -length array *`pDlyLine`* specifies the delay line values. The filter order is defined by the *`order`* value which is equal to 0 for zero-order filters. The 2*(*`order`* + 1) -length array *`pTaps`* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

$A_0 \neq 0$

`ippsIIRInit_BiQuad`. The function `ippsIIRInit_BiQuad` initializes the taps and the delay line in the state structure of a biquad IIR filter; that is, defined by a cascade of biquads. The 2*`numBq` -length array *`pDlyLine`* specifies the delay line values. The number of cascades of biquads is defined by the *`numBq`* value. The 6*`numBq` -length array *`pTaps`* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \ldots A_{numBq\text{-}1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsIIROrderErr`	Indicates an error when *`order`* is less than or equal to 0, or *`numBq`* is less than 1.

IIRGetStateSize, IIRGetStateSize_BiQuad

Returns the length of the IIR filter state structure.

```
IppStatus ippsIIRGetStateSize_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64fc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize32s_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32s_16s32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_16sc32fc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize32f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32fc_16sc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize64f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32fc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64fc(int numBq, int* pBufferSize);
```

```
IppStatus ippsIIRGetStateSize32s_BiQuad_16s(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize32s_BiQuad_16s32f(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_BiQuad_16sc(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_BiQuad_16sc32fc(int numBq,
     int* pBufferSize);

IppStatus ippsIIRGetStateSize32f_BiQuad_16s(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize32fc_BiQuad_16sc(int numBq,
     int* pBufferSize);

IppStatus ippsIIRGetStateSize64f_BiQuad_16s(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32s(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32f(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_16sc(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32sc(int numBq,
     int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32fc(int numBq,
     int* pBufferSize);
```

Arguments

order	Order of the IIR filter. This argument is used for arbitrary filters.
numBq	Number of cascades of biquads. This argument is used for BQ filters.
pBuffer	Pointer to the external buffer for the IIR state structure.

Discussion

The functions `ippsIIRGetStateSize` and `ippsIIRGetStateSize_BiQuad` are declared in the `ipps.h` file. These functions compute the size of the external buffer for an arbitrary or biquad IIR filter state, respectively, and store the result in *pBufferSize*.

To compute a size of the buffer for an arbitrary IIR filter, the filter order argument *order* should be specified.

To compute a size of the buffer for a biquad IIR filter, the number of cascades of biquads *numBq* should be specified.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pStateSize* is `NULL`.
`ippStsIIROrderErr`	Indicates an error when *order* is less than 0, or *numBq* is less than 1.

IIRSetTaps

Sets the taps in an IIR filter state.

```
IppStatus ippsIIRSetTaps_32f(const Ipp32f* pTaps,
     IppsIIRState_32f* pState);
IppStatus ippsIIRSetTaps_32fc(const Ipp32fc* pTaps,
     IppsIIRState_32fc* pState);
IppStatus ippsIIRSetTaps_64f(const Ipp64f* pTaps,
     IppsIIRState_64f* pState);
IppStatus ippsIIRSetTaps_64fc(const Ipp64fc* pTaps,
     IppsIIRState_64fc* pState);
IppStatus ippsIIRSetTaps32s_16s(const Ipp32s* pTaps,
     IppsIIRState32s_16s* pState, int tapsFactor);
IppStatus ippsIIRSetTaps32s_16s32f(const Ipp32f* pTaps,
     IppsIIRState32s_16s* pState);
```

```
IppStatus ippsIIRSetTaps32sc_16sc(const Ipp32sc* pTaps,
     IppsIIRState32sc_16sc* pState, int tapsFactor);
IppStatus ippsIIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps,
     IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRSetTaps32f_16s(const Ipp32f* pTaps,
     IppsIIRState32f_16s* pState);
IppStatus ippsIIRSetTaps32fc_16sc(const Ipp32fc* pTaps,
     IppsIIRState32fc_16sc* pState);
IppStatus ippsIIRSetTaps64f_16s(const Ipp64f* pTaps,
     IppsIIRState64f_16s* pState);
IppStatus ippsIIRSetTaps64f_32s(const Ipp64f* pTaps,
     IppsIIRState64f_32s* pState);
IppStatus ippsIIRSetTaps64f_32f(const Ipp64f* pTaps,
     IppsIIRState64f_32f* pState);
IppStatus ippsIIRSetTaps64fc_16sc(const Ipp64fc* pTaps,
     IppsIIRState64fc_16sc* pState);
IppStatus ippsIIRSetTaps64fc_32sc(const Ipp64fc* pTaps,
     IppsIIRState64fc_32sc* pState);
IppStatus ippsIIRSetTaps64fc_32fc(const Ipp64fc* pTaps,
     IppsIIRState64fc_32fc* pState);
```

Arguments

pTaps	Pointer to the array containing the tap values.
pState	Pointer to the IIR filter state structure.
tapsFactor	Scale factor for the taps of Ipp32s data type (for integer versions only).

Discussion

The function ippsIIRSetTaps is declared in the ipps.h file. This function sets new tap values in the previously initialized IIR filter state structure *pState*. New tap values must be specified in the array *pTaps*. To scale integer taps use the *tapsFactor* value.

The filter state must be initialized before calling the function ippsIIRSetTaps. The length of the array *pTaps* should be equal to the *tapsLen* parameter of the initialized filter state.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

IIRGetDlyLine, IIRSetDlyLine

Gets and sets the delay line contents of an IIR filter state.

```
IppStatus ippsIIRGetDlyLine_32f(const IppsIIRState_32f* pState,
     Ipp32f* pDlyLine);
IppStatus ippsIIRGetDlyLine_64f(const IppsIIRState_64f* pState,
     Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine_32fc(const IppsIIRState_32fc* pState,
     Ipp32fc* pDlyLine);
IppStatus ippsIIRGetDlyLine_64fc(const IppsIIRState_64fc* pState,
     Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine32f_16s(const IppsIIRState32f_16s* pState,
     Ipp32f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_16s(const IppsIIRState64f_16s* pState,
     Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32s(const IppsIIRState64f_32s* pState,
     Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32f(const IppsIIRState64f_32f* pState,
     Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine32s_16s(const IppsIIRState32s_16s* pState,
     Ipp32s* pDlyLine);
IppStatus ippsIIRGetDlyLine32sc_16sc(const IppsIIRState32sc_16sc*
     pState, Ipp32sc* pDlyLine);
IppStatus ippsIIRGetDlyLine32fc_16sc(const IppsIIRState32fc_16sc*
     pState, Ipp32fc* pDlyLine);
```

```
IppStatus ippsIIRGetDlyLine64fc_16sc(const IppsIIRState64fc_16sc*
     pState, Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32sc(const IppsIIRState64fc_32sc*
     pState, Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32fc(const IppsIIRState64fc_32fc*
     pState, Ipp64fc* pDlyLine);


IppStatus ippsIIRSetDlyLine_32f(IppsIIRState_32f* pState,
     const Ipp32f* pDlyLine);
IppStatus ippsIIRSetDlyLine_64f(IppsIIRState_64f* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine_32fc(IppsIIRState_32fc* pState,
     const Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine_64fc(IppsIIRState_64fc* pState,
     const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine32s_16s(IppsIIRState32s_16s* pState,
     const Ipp32s* pDlyLine);
IppStatus ippsIIRSetDlyLine32f_16s(IppsIIRState32f_16s* pState,
     const Ipp32f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_16s(IppsIIRState64f_16s* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32s(IppsIIRState64f_32s* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32f(IppsIIRState64f_32f* pState,
     const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState,
     const Ipp32sc* pDlyLine);
IppStatus ippsIIRSetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState,
     const Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState,
     const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState,
     const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState,
     const Ipp64fc* pDlyLine);
```

Arguments

pState	Pointer to the IIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters. If the pointer is NULL, then the delay line values in the state structure are initialized to zero.

Discussion

The functions ippsIIRGetDlyLine and ippsIIRSetDlyLine are declared in the ipps.h file. These functions get and set the delay line values of a IIR filter state.

ippsIIRGetDlyLine. The function ippsIIRGetDlyLine copies the delay line values from the state structure *pState* and stores them into the arrays *pDlyLine*.

ippsIIRSetDlyLine. The function ippsIIRSetDlyLine copies the delay line values from *pDlyLine* and stores them into the state structure *pState*.

Before calling either ippsIIRGetDlyLine or ippsIIRSetDlyLine, initialize the filter state by calling the function ippsIIRInitAlloc or ippsIIPBQInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

IIROne

Filters a single sample through an IIR filter.

```
IppStatus ippsIIROne_32f(Ipp32f src, Ipp32f* pDstVal,
     IppsIIRState_32f* pState);
IppStatus ippsIIROne_64f(Ipp64f src, Ipp64f* pDstVal,
     IppsIIRState_64f* pState);
```

```
IppStatus ippsIIROne64f_32f(Ipp32f src, Ipp32f* pDstVal,
     IppsIIRState64f_32f* pState);
IppStatus ippsIIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     IppsIIRState_32fc* pState);
IppStatus ippsIIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal,
     IppsIIRState_64fc* pState);
IppStatus ippsIIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal,
     IppsIIRState64fc_32fc* pState);

IppStatus ippsIIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
     IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
     IppsIIRState64f_32s* pState, int scaleFactor);

IppStatus ippsIIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
     IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
     IppsIIRState64fc_32sc* pState, int scaleFactor);
```

Arguments

pState	Pointer to the IIR filter state structure.
src	Input sample to be filtered by the function ippsIIROne.
pDstVal	Pointer to the output sample filtered by the function ippsIIROne.
scaleFactor	Refer to “Integer Scaling” in Chapter 2.

Discussion

The function ippsIIROne is declared in the ipps.h file. This function filters a single sample *src* through an IIR filter with real taps, and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

Before calling either the ippsIIR or ippsFIROne function, initialize the filter state by calling ippsIIRInitAlloc or ippsIIRInitAlloc_BiQuad. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

IIR

Filters a block of samples through an IIR filter.

```
IppStatus ippsIIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
     IppsIIRState_32f* pState);
IppStatus ippsIIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
     IppsIIRState_64f* pState);
IppStatus ippsIIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
     IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
     IppsIIRState_64fc* pState);
IppStatus ippsIIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
     IppsIIRState64f_32f* pState);
IppStatus ippsIIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
      IppsIIRState64fc_32fc* pState);
```

```
IppStatus ippsIIR_32f_I(Ipp32f* pSrcDst, int len,
     IppsIIRState_32f* pState);
IppStatus ippsIIR_64f_I(Ipp64f* pSrcDst, int len,
     IppsIIRState_64f* pState);
IppStatus ippsIIR_32fc_I(Ipp32fc* pSrcDst, int len,
     IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc_I(Ipp64fc* pSrcDst, int len,
     IppsIIRState_64fc* pState);
IppStatus ippsIIR64f_32f_I(Ipp32f* pSrcDst, int len,
     IppsIIRState64f_32f* pState);
IppStatus ippsIIR64fc_32fc_I(Ipp32fc* pSrcDst, int len,
     IppsIIRState64fc_32fc* pState);

IppStatus ippsIIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
     IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
     IppsIIRState64f_32s* pState, int scaleFactor);

IppStatus ippsIIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     int len, IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
     int len, IppsIIRState64fc_32sc* pState, int scaleFactor);
IppStatus ippsIIR32s_16s_ISfs(Ipp16s* pSrcDst, int len,
     IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIR32f_16s_ISfs(Ipp16s* pSrcDst, int len,
     IppsIIRState32f_16s* pState, int scaleFactor);
```

```
IppStatus ippsIIR64f_16s_ISfs(Ipp16s* pSrcDst, int len,
     IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_32s_ISfs(Ipp32s* pSrcDst, int len,
     IppsIIRState64f_32s* pState, int scaleFactor);
IppStatus ippsIIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
     IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
     IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
     IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int len,
     IppsIIRState64fc_32sc* pState, int scaleFactor);
```

Arguments

pState	Pointer to the IIR filter state structure.
pSrc	Pointer to the input array to be filtered by the function ippsIIR.
pDst	Pointer to the output array filtered by the function ippsIIR.
pSrcDst	Pointer to the input and output array (for the in-place operation) to be filtered by the function ippsIIR.
len	Number of samples to be filtered by the function ippsIIR.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsIIR is declared in the ipps.h file. This function filters *len* samples in the input array *pSrc* or *pSrcDst* through an IIR filter with real taps, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

Before calling ippsIIR, initialize the filter state by calling ippsIIRInitAlloc or ippsIIRInitAlloc_BiQuad. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

Example 6-6 illustrates using ippsIIR_32f to suppress a 60 Hz signal.

Example 6-7 illustrates using ippsIIR to filter a sample. The function ippsCnvrt_64f32s_Sfs converts floating-point taps into integer data type before calling ippsIIRInitAlloc_32s.

Return Value

ippStsNoErr	Indicates 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.

Example 6-6 Using the ippsIIR_32f Function to Suppress a 60 Hz Signal

```
IppStatus iir( void ) {
#undef NUMITERS
#define NUMITERS 150
     int n;
     IppStatus status;
     IppsIIRState_32f *ctx;
     Ipp32f *x = ippsMalloc_32f( NUMITERS ), *y = ippsMalloc_32f( NUMITERS );
     /// A second-order notch filter having notch freq at 60 Hz
     const float taps[] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
     };
     /// generate a signal having 60 Hz freq sampled with 400 Hz freq
     for(n=0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *60 /400);
     ippsIIRInitAlloc_32f( &ctx, taps, 2, NULL );
     status = ippsIIR_32f( x, y, NUMITERS, ctx );
     printf_32f( " IIR 32f output+120 =", y+120, 5, status );
     ippsIIRFree_32f( ctx );
     ippsFree( y );
     ippsFree( x );
     return status;
}
Output:
    IIR 32f output + 120 =  -0.000094 0.000339 0.000458 0.000208 -0.000173
Matlab* Analog:
    >> B = [0.940809,-1.105987,0.940809]; A = [1,-1.105987,0.881618];
    n = 0:150; x = sin(2*pi*n*60/400); y = filter(B,A,x); y(121:125)
```

Example 6-7 Using the ippsIIR Function to Filter a Sample

```
IppStatus iir16s( void ) {
#undef NUMITERS
#define NUMITERS 150
     int n, tapsfactor = 30;
     IppStatus status;
     IppsIIRState32s_16s *ctx;
     Ipp16s *x = ippsMalloc_16s( NUMITERS ), *y = ippsMalloc_16s( NUMITERS );
     /// A second-order notch filter having notch freq at 60 Hz
     Ipp64f taps[6] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
     };
     Ipp32s taps32s[6];
     Ipp64f tmax, tmp[6];
     ippsAbs_64f( taps, tmp, 6 );
     ippsMax_64f( tmp, 6, &tmax );
     tapsfactor = 0;
     if( tmax > IPP_MAX_32S )
        while( (tmax/=2) > IPP_MAX_32S ) ++tapsfactor;
     else
        while( (tmax*=2) < IPP_MAX_32S ) --tapsfactor;
     if(tapsfactor >0 )
        ippsDivC_64f_I( (float)(1<<(++tapsfactor)), taps, 6 );
     else if( tapsfactor < 0 )
        ippsMulC_64f_I( (float)(1<<(-(tapsfactor))), taps, 6 );
     ippsConvert_64f32s_Sfs( taps, taps32s, 6, ippRndNear, 0 );
     /// generate a signal of 60 Hz freq that is sampled with 400 Hz freq
     for(n=0; n<NUMITERS; ++n) x[n] = (Ipp16s)(1000*sin(IPP_2PI*n*60/400));
     ippsIIRInitAlloc32s_16s( &ctx, taps32s, 2, tapsfactor, NULL );
     status = ippsIIR32s_16s_Sfs( x, y, NUMITERS, ctx, 0);
     printf_16s( " IIR 32s output+120 =", y+120, 5, status );
     ippsIIRFree32s_16s( ctx );
     ippsFree( y );
     ippsFree( x );
     return status;
}
Output:
    IIR 32s output + 120 =  0 0 0 0 0
```

IIROne_Direct, IIROne_BiQuadDirect

Directly filters a single sample through an IIR filter.

```
IppStatus ippsIIROne_Direct_16s(Ipp16s src, Ipp16s* pDtsVal, const Ipp16s*
     pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIROne_BiQuadDirect_16s(Ipp16s src, Ipp16s* pDtsVal,
     const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
IppStatus ippsIIROne_Direct_16s_I(Ipp16s* pSrcDtsVal, const Ipp16s* pTaps,
     int order, Ipp32s* pDlyLine);
IppStatus ippsIIROne_BiQuadDirect_16s_I(Ipp16s* pSrcDtsVal,
     const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
```

Arguments

src	Input sample to be filtered by the function.
pDstVal	Pointer to the output sample.
pSrcDstVal	Pointer to the input and output sample for in-place operation.
pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1) for arbitrary filters, and 6**numBq* for BQ filters.
order	Order of the arbitrary IIR filter.
numBq	Number of cascades of biquads.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order* for arbitrary filters, and 2**numBq* for BQ filters.

Discussion

The functions `ippsIIROne_Direct` and `ippsIIROne_BiQuadDirect` are declared in the `ipps.h` file.

ippsIIROne_Direct. The function ippsIIROne_Direct filters a single sample *src* (*pSrcDstVal* for in-place flavors) through an arbitrary order IIR filter and stores the result in *pDstVal* (*pSrcDstVal*). The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The *order* -length array *pDlyLine* specifies the delay line values. The 2*(*order* + 1) -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

The value $A_0 \geq 0$ holds the scale factor (and not a divisor) for all the other taps.

ippsIIROne_BiQuadDirect. The function ippsIIROne_BiQuadDirect filters a single sample *src* (*pSrcDstVal* for in-place flavors) through a biquad IIR filter and stores the result in *pDstVal* (*pSrcDstVal*). The number of cascades of biquads is defined by the *numBq* value. The 2**numBq* -length array *pDlyLine* specifies the delay line values. The 6**numBq* -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \ldots A_{numBq\text{-}1,2}$

Values $A_{n,0} \geq 0$, $B_{n,0} \geq 0$ hold the scale factors (and not divisors) for all the other taps of each section.

Return Value

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsIIROrderErr	Indicates an error when *order* is negative or *numBq* is less or equal to 0.
ippStsScaleRangeErr	Indicates an error when A_0, or $A_{n,0}$ is less than 0.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect

IIR_Direct, IIR_BiQuadDirect

Directly filters a block of samples through an IIR filter.

```
IppStatus ippsIIR_Direct_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
     len, const Ipp16s* pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIR_BiQuadDirect_16s(const Ipp16s* pSrc, Ipp16s* pDst,
     int len, const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
IppStatus ippsIIR_Direct_16s_I(Ipp16s* pSrcDst, int len, const Ipp16s*
     pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIR_BiQuadDirect_16s_I(Ipp16s* pSrcDst, int len, const
     Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
```

Arguments

pSrc	Pointer to the input array to be filtered.
pDst	Pointer to the output (filtered) array.
pSrcDst	Pointer to the input and output array (for the in-place operation) to be filtered by the function.
len	Number of samples to be filtered.
pTaps	Pointer to the array containing the taps. The number of elements in the array is 2*(*order*+1) for arbitrary filters, and 6**numBq* for BQ filters.
order	Order of the arbitrary IIR filter.
numBq	Number of cascades of biquads.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order* for arbitrary filters, and 2**numBq* for BQ filters.

Discussion

The functions ippsIIR_Direct and ippsIIR_BiQuadDirect are declared in the ipps.h file.

ippsIIR_Direct. The function ippsIIR_Direct filters *len* samples in the input array *pSrc* or *pSrcDst* through an arbitrary IIR filter and stores the results in *pDst* or *pSrcDst*, respectively. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The *order* -length array *pDlyLine* specifies the delay line values. The 2*(*order* + 1) -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

The value $A_0 \geq 0$ holds the scale factor (and not a divisor) for all the other taps.

ippsIIR_BiQuadDirect. The function ippsIIR_BiQuadDirect filters *len* samples in the input array *pSrc* or *pSrcDst* through a biquad IIR filter and stores the result in *pDst* or *pSrcDst*, respectively. The number of cascades of biquads is defined by the *numBq* value. The 2**numBq* -length array *pDlyLine* specifies the delay line values. The 6**numBq* -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \ldots A_{numBq\text{-}1,2}$

Values $A_{n,0} \geq 0$, $B_{n,0} \geq 0$ hold the scale factors (and not divisors) for all the other taps of each section.

Return Value

ippStsNoErr	Indicates no error.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.
ippStsIIROrderErr	Indicates an error when *order* is negative or *numBq* is less or equal to 0.
ippStsScaleRangeErr	Indicates an error when A_0, or $A_{n,0}$ is less than 0.

Median Filter Functions

Median filters are nonlinear rank-order filters based on replacing each element of the input array with the median value, taken over the fixed neighborhood (mask) of the processed element. These filters are extensively used in image and signal processing applications. Median filtering removes impulsive noise, while keeping the signal blurring to the minimum. Typically mask size (or window width) is set to odd value which ensures simple function implementation and low output signal bias. In the Intel IPP median function implementation, the mask is always centered at the input element for which the median value is computed. You can use an even mask size in function calls as well, but internally it will be changed to odd by subtracting 1. Another specific feature of the median function implementation in the Intel IPP is that elements outside the input array, which are needed to determine the median value for "border" elements, are set to be equal to the corresponding edge element of the input array, i.e. are padded by cloning the edge element.

FilterMedian

Computes median values for each input array element.

```
IppStatus ippsFilterMedian_8u(const Ipp8u* pSrc, Ipp8u* pDst,
      int len, int maskSize);
IppStatus ippsFilterMedian_16s(const Ipp16s* pSrc, Ipp16s* pDst,
      int len, int maskSize);
IppStatus ippsFilterMedian_32s(const Ipp32s* pSrc, Ipp32s* pDst,
      int len, int maskSize);
IppStatus ippsFilterMedian_32f(const Ipp32f* pSrc, Ipp32f* pDst,
      int len, int maskSize);
IppStatus ippsFilterMedian_64f(const Ipp64f* pSrc, Ipp64f* pDst,
      int len, int maskSize);

IppStatus ippsFilterMedian_8u_I(Ipp8u* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_16s_I(Ipp16s* pSrcDst, int len, int maskSize);
```

```
IppStatus ippsFilterMedian_32s_I(Ipp32s* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_32f_I(Ipp32f* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_64f_I(Ipp64f* pSrcDst, int len, int maskSize);
```

Arguments

pSrcDst	Pointer to the input and output array (for the in-place operation).
pSrc	Pointer to the input array to be filtered by the function ippsFilterMedian.
pDst	Pointer to the output array filtered by the function ippsFilterMedian.
len	Number of elements in the array.
maskSize	Median mask size, must be a positive integer. If an even value is specified, the function subtracts 1 and uses the odd value of the filter mask for median filtering.

Discussion

The function ippsFilterMedian is declared in the ipps.h file. This function computes median values for each element of the input array *pSrc* or *pSrcDst*, and stores the result in *pDst* or *pSrcDst*, respectively.

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

Example 6-8 illustrates using ippsFilterMedian_16s_I for single-rate filtering.

Return Value

ippStsNoErr	Indicates 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.

`ippStsEvenMedianMaskSize`
Indicates a warning when the median mask length is even.

Example 6-8 Single-Rate Filtering with the ippsFilterMedian Function

```
void median(void) {
      Ipp16s x[8] = {1,2,127,4,5,0,7,8};
      IppStatus status = ippsFilterMedian_16s_I(x, 8, 3);
      printf_16s("median =", x,8, status);
}
Output:
    median =  1 2 4 5 4 5 7 8
Matlab* Analog:
    >> x = [1 2 127 4 5 0 7 8]; medfilt1(x)
```

Transform Functions

7

This chapter describes Intel® IPP functions that perform Fourier and discrete cosine transforms (DCT), as well as Hilbert and wavelet transforms of signals.

The full list of functions in this group is given in Table 7-1.

Table 7-1 Intel IPP Transform Functions

Function Base Name	Operation
Support Functions	
ConjPerm	Converts the data in Perm format to complex data format.
ConjPack	Converts the data in Pack format to complex data format.
ConjCcs	Converts the data in CCS format to complex data format.
MulPack, MulPerm	Multiplies the elements of two vectors stored in Pack or Perm format.
MulPackConj	Multiplies elements of a vector by the elements of a complex conjugate vector stored in Pack format.
Fast Fourier Transform Functions	
FFTInitAlloc_C, FFTInitAlloc_R	Initializes the fast Fourier transform structure for real and complex signals.
FFTFree_C, FFTFree_R	Closes a fast Fourier transform structure for real and complex signals.
FFTGetBufSize_C, FFTGetBufSize_R	Gets the size of the FFT work buffer in bytes.
FFTFwd_CToC, FFTInv_CToC	Computes the forward or inverse fast Fourier transform (FFT) of a complex signal.

Table 7-1 Intel IPP Transform Functions (continued)

Function Base Name	Operation
FFTFwd_RToPerm, FFTInv_PermToR, FFTFwd_RToPack, FFTInv_PackToR, FFTFwd_RToCCS, FFTInv_CCSToR	Computes the forward or inverse fast Fourier transform (FFT) of a real signal.
Discrete Fourier Transform Functions	
DFTInitAlloc_C, DFTInitAlloc_R	Initializes the discrete Fourier transform structure for real and complex signals.
DFTFree_C, DFTFree_R	Closes the discrete Fourier transform structure for real and complex signals.
DFTGetBufSize_C, DFTGetBufSize_R	Gets the size of the discrete Fourier transform work buffer in bytes.
DFTFwd_CToC, DFTInv_CToC	Computes the forward or inverse discrete Fourier transform of a complex signal.
DFTFwd_RToPerm, DFTInv_PermToR, DFTFwd_RToPack, DFTInv_PackToR, DFTFwd_RToCCS, DFTInv_CCSToR	Computes the forward or inverse discrete Fourier transform of a real signal.
DFTOutOrdInitAlloc_C	Initializes the out-of-order discrete Fourier transform structure.
DFTOutOrdFree_C	Closes the out-of-order discrete Fourier transform structure.
DFTOutOrdGetBufSize_C	Computes the size of the work buffer for the out-of-order discrete Fourier transform.
DFTOutOrdFwd_CToC, DFTOutOrdInv_CToC	Computes the forward or inverse out-of-order discrete Fourier transform.
Goertz	Computes the discrete Fourier transform for a given frequency for a single complex signal.
GoertzTwo	Computes two discrete Fourier transforms for a given frequency for a single complex signal.
DCT Functions	
DCTFwdInitAlloc , DCTInvInitAlloc	Initializes the discrete cosine transform structure.

Table 7-1 Intel IPP Transform Functions (continued)

Function Base Name	Operation
DCTFwdFree, DCTInvFree	Closes a discrete cosine transform structure.
DCTFwdGetBufSize, DCTInvGetBufSize	Gets the size of the DCT work buffer in bytes.
DCTFwd, DCTInv	Computes the forward or inverse discrete cosine transform (DCT) of a signal.
Hilbert Transform Functions	
HilbertInitAlloc	Initializes the Hilbert transform structure.
HilbertFree	Closes a Hilbert transform structure.
Hilbert	Computes an analytic signal using the Hilbert transform.
Wavelet Transform Functions	
WTHaarFwd, WTHaarInv	Performs forward or inverse single-level discrete wavelet Haar transforms.
WTFwdInitAlloc, WTInvInitAlloc	Initializes the wavelet transform structure.
WTFwdFree, WTInvFree	Closes a wavelet transform structure.
WTFwd	Computes the forward wavelet transform.
WTFwdSetDlyLine, WTFwdGetDlyLine	Sets and gets the delay lines of the forward wavelet transform.
WTInv	Computes the inverse wavelet transform.
WTInvSetDlyLine, WTInvGetDlyLine	Sets and gets the delay lines of the inverse wavelet transform.

Fourier Transform Functions

This section describes the Fourier and the discrete cosine transform functions in the Intel IPP. The Intel IPP contains functions which perform the discrete Fourier transform (DFT), the fast Fourier transform (FFT), and the discrete cosine transform (DCT) of signal samples. It also includes variations of the basic functions to support different application requirements.

Transform Support Functions

This section describes the flag and hint arguments used by the Fourier and the discrete transform functions, the main packed formats `Perm`, `Pack`, and `CCS`, and the functions performing unpack, conversion, and multiplication of data stored in the above formats.

Flag and Hint Arguments

The Fourier transform functions require you to specify the *flag* and *hint* arguments.

The *flag* argument specifies the result normalization method. Table 7-2 lists the values you can enter for the *flag* argument. Specify one and only one of the represented values in the *flag* argument. The *A* and *B* factors are multipliers used in the DFT computation.

Table 7-2 Flag Arguments for Fourier Transform Functions

Value	A	B	Description
IPP_FFT_DIV_FWD_BY_N	$1/N$	1	Forward transform is done with the $1/N$ normalization.
IPP_FFT_DIV_INV_BY_N	1	$1/N$	Inverse transform is done with the $1/N$ normalization.
IPP_FFT_DIV_BY_SQRTN	$1/N^{1/2}$	$1/N^{1/2}$	Forward and inverse transform is done with the $1/N^{1/2}$ normalization.
IPP_FFT_NODIV_BY_ANY	1	1	Forward or inverse transform is done without the $1/N$ or $1/N^{1/2}$ normalization.

The *hint* argument suggests using special code, faster but less accurate calculation, or more accurate but slower calculation. Table 7-3 lists values you can enter for the *hint* argument.

Note that the *hint* argument is also used for tone generation, see Tone-Generating Functions in Chapter 4, and for logarithmic addition, see Model Evaluation in Chapter 8.

Table 7-3 Hint Arguments for Fourier Transform Functions

Value	Description
ippAlgHintNone	No suggestions. The function selects an algorithm
ippAlgHintFast	Fast algorithm is suggested for computation.
ippAlgHintAccurate	Accurate algorithm is suggested for computation.

Pack Format

The Pack format is a convenient, compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms ("natural" in the sense that bit-reversed order is natural for radix-2 complex FFTs). In Pack format, the output samples of the FFT are arranged as shown in Table 7-4. The output signal can be unpacked to a complex signal using the function ippsConjPack, see "ConjPack" on page 7-8 for more information.

Perm Format

The Perm format stores the values in the order in which the FFT algorithm uses them. This is the most natural way of storing values for the FFT algorithm. The Perm format is an arbitrary permutation of the Pack format. An important characteristic of the Perm format is that the real and imaginary parts of a given sample need not be adjacent.

In Perm format, the output samples of the FFT are arranged as shown in Table 7-4. The output signal can be unpacked to a complex signal using the function ippsConjPerm, see "ConjPerm" on page 7-6 for more information.

CCS Format

The CCS format stores the values of the first half of the output complex signal resulted from the forward FFT.

In CCS format, the output samples of the FFT are arranged as shown in Table 7-4. Note that the signal stored in CCS format is one complex element longer. The output signal can be unpacked to a complex signal using the function ippsConjCcs, see "ConjCcs" on page 7-10 for more information.

Table 7-4 Forward FFT Result Representation in Pack, Perm, and CCS Formats

FFTReal	0	1	2	3	...	N-2	N-1	N	N+1
Pack	R_0	R_1	I_1	R_2	...	$I_{N/2-1}$	$R_{N/2}$		
Perm	R_0	$R_{N/2}$	R_1	I_1	...	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	R_0	0	R_1	I_1	...	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

Unpack of Packed Data

The following functions `ConjPerm`, `ConjPack`, and `ConjCcs` convert data from the packed formats to a usual complex data format using the FFT symmetry property for transforming real data. The output data is complex, the output array length is defined by the number of complex elements in the output vector. Note that the output array size is two times as big as the input array size. The data stored in `CCS` format require a bigger array than the other formats. Even and odd length arrays have some specific features discussed for each function separately.

ConjPerm

Converts the data in `Perm` *format to complex data format.*

```
IppStatus ippsConjPerm_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPerm_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPerm_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPerm_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Arguments

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
lenDst	Number of elements in the vector.

Discussion

The function ippsConjPerm is declared in the ipps.h file. This function converts the data in Perm format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function ippsConjPerm converts the data in Perm format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-5 shows the examples of unpack from the Perm format. The Data column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the Packed column. The output result is the complex data vector in the Extended column. The number of vector elements is in the Length column. Example 7-1 shows how to use the function ippsConjPerm.

Return Value

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

Table 7-5 Examples of Packed Data Obtained by FFT

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -7, -2, 7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-1 Using the ippsConjPerm Function

```
void ConjPerm(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPerm_16sc( x, y, 6 );
      printf_16sc("Perm 6:", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPerm_16sc( x, y, 5 );
      printf_16sc("Perm 5:", y, 5, st );
}
Output:
    Perm 6:  {1,0} {3,5} {6,7} {2,0} {6,-7} {3,-5}
    Perm 5:  {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjPack

Converts the data in `Pack` *format to complex data format.*

```
IppStatus ippsConjPack_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPack_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPack_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPack_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPack_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPack_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Arguments

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

Discussion

The function ippsConjPack is declared in the ipps.h file. This function converts the data in Pack format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function ippsConjPack converts the data in Pack format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-6 shows the examples of unpack from the Pack format. The Data column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the Packed column. The output result is the complex data vector in the Extended column. The number of vector elements is in the Length column. Example 7-2 shows how to use the function ippsConjPack.

Return Value

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

Table 7-6 Examples of Unpack from the Pack Format

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -2, 7, -7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-2 Using the ippsConjPack Function

```
void ConjPack(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPack_16sc( x, y, 6 );
      printf_16sc("pack 6", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjPack_16sc( x, y, 5 );
      printf_16sc("pack 5", y, 5, st );
}
Output:
    Pack 6: {1,0} {2,3} {5,6} {7,0} {5,-6} {2,-3}
    Pack 5: {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjCcs

Converts the data in CCS *format to complex data format.*

```
IppStatus ippsConjCcs_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjCcs_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjCcs_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjCcs_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Arguments

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

Discussion

The function ippsConjCcs is declared in the ipps.h file. This function converts the data in CCS format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function ippsConjCcs converts the data in CCS format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

Table 7-7 shows the examples of unpack from the CCS format. The Data column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the Packed column. The output result is the complex data vector in the Extended column. The number of vector elements is in the Length column.The data stored in CCS format are two real elements longer.Example 7-3 shows how to use the function ippsConjCcs.

Return Value

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

Table 7-7 Examples of Unpack from the CCS Format

Data	Packed	Extended	Length
FFT([1])	1, 0	{1, 0}	1
FFT([1 2])	3, 0, -1, 0	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, 0, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, 0, -2, 7, -7, 0	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-3 Using the ippsConjCcs Function

```
void ConjCcs(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjCcs_16sc( x, y, 6 );
      printf_16sc("CCS 6:", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjCcs_16sc( x, y, 5 );
      printf_16sc("CCS 5:", y, 5, st );
}
Ouput:
    CCS 6:  {1,2} {3,5} {6,7} {8,9} {6,-7} {3,-5}
    CCS 5:  {1,2} {3,5} {6,7} {6,-7} {3,-5}
```

Multiplication of Packed Data

The functions described in this section perform the element-wise complex multiplication of vectors stored in `Pack` or `Perm` formats. These functions are used with the function `ippsFFTFwd_RToPack` and `ippsFFTInv_PackToR` to perform fast convolution on real signals.

The standard vector multiplication function `ippsMul` can not be used to multiply `Pack` or `Perm` format vectors because:

- Two real samples are stored in `Pack` format.
- The `Perm` format might not pair the real parts of a signal with their corresponding imaginary parts.

The argument *order* indicates base-2 logarithm of the length N of the FFT, where $N = 2^{order}$.

MulPack, MulPerm

Multiply the elements of two vectors stored in `Pack` *or* `Perm` *format.*

```
IppStatus ippsMulPack_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int length);
IppStatus ippsMulPack_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
      Ipp64f* pDst, int length);
IppStatus ippsMulPack_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int length, int scaleFactor);
IppStatus ippsMulPack_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
      int length);
IppStatus ippsMulPack_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
      int length);
IppStatus ippsMulPack_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
      int length, int scaleFactor);

IppStatus ippsMulPerm_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
      Ipp32f* pDst, int length);
IppStatus ippsMulPerm_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
      Ipp64f* pDst, int length);
IppStatus ippsMulPerm_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
      Ipp16s* pDst, int length, int scaleFactor);
IppStatus ippsMulPerm_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
      int length);
IppStatus ippsMulPerm_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
      int length);
IppStatus ippsMulPerm_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
      int length, int scaleFactor);
```

Arguments

pSrc1, pSrc2	Pointers to the vectors whose elements are to be multiplied together.

pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc1[n] * pSrc2[n]*.
pSrc	Pointer to the vector whose elements are to be multiplied by the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
length	Number of elements in the vector.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions ippsMulPack and ippsMulPerm are declared in the ipps.h file. These functions multiply the elements of the vector *pSrc1* by the elements of the vector *pSrc2*, and store the result in *pDst*.

The in-place functions ippsMulPack and ippsMulPerm multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The functions multiply the packed data according to their packed format. The data in Perm and Pack packed formats include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

ippsMulPack. The function ippsMulPack performs multiplication of the data stored in Pack format.

ippsMulPerm. The function ippsMulPerm performs multiplication of the data stored in Perm format.

The vectors stored in CCS format can be multiplied using the standard function for complex data multiplication. For the functions with the Sfs suffixes scaling is performed in accordance with the *scaleFactor* value. When the output value exceeds the data range, the result may become saturated.

Example 7-4 shows how to use the function ippsMulPack_32f_I.

Return Value

ippStsNoErr	Indicates no error
ippStsNullPtrErr	Indicates an error when the *pSrcDst*, *pDst*, *pSrc1*, *pSrc2*, or *pSrc* pointer is NULL
ippStsSizeErr	Indicates an error when *length* is less than or equal to 0

Example 7-4 Using the ippsMulPack Function

```
void mulpack( void ) {
     Ipp32f x[8], X[8], h[8]={1.0f/3,1.0f/3,1.0f/3,0,0,0,0,0}, H[8];
     IppStatus st;
     IppsFFTSpec_R_32f* spec;
     st = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
         ippAlgHintNone);
     ippsSet_32f( 3, x, 8 );
     x[3] = 5;
     st = ippsFFTFwd_RToPack_32f( x, X, spec, NULL );
     st = ippsFFTFwd_RToPack_32f( h, H, spec, NULL );
     ippsMulPack_32f_I( H, X, 8 );
     st = ippsFFTInv_PackToR_32f( X, x, spec, NULL );
     printf_32f("filtered =", x, 8, st );
     ippsFFTFree_R_32f( spec );
}
Output:
    filtered =  3.0 3.0 3.0 3.666667 3.666667 3.666667 3.0 3.0
Matlab* analog:
    >> x=3*ones(1,8); x(4)=5;h=zeros(1,8); h(1:3)=1/3;
    real(ifft(fft(x).*fft(h)))
```

MulPackConj

Multiplies elements of a vector by the elements of a complex conjugate vector stored in `Pack` *format.*

```
IppStatus ippsMulPackConj_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
     int length);
IppStatus ippsMulPackConj_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
     int length);
```

Arguments

pSrc	Pointer to the first source vector.
pSrcDst	Pointer to the second source and destination vector.
length	Number of elements in the vector.

Discussion

The function `ippsMulPackConj` is declared in the `ipps.h` file. This function multiplies the elements of a source vector *pSrc* by elements of the vector that is complex conjugate to the source vector *pSrcDst* and stores the results in *pSrcDst*. The function performs only in-place operations on data stored in `Pack` format.

Return Value

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

Fast Fourier Transform Functions

The functions described in this section compute the forward and inverse fast Fourier transform of real and complex signals. The FFT is similar to the discrete Fourier transform (DFT) but is significantly faster. The length of the vector transformed by the FFT must be a power of 2.

To use the FFT functions, initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms. The amount of prior calculations is thus reduced and the overall performance increased.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method. The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in `Perm`, `Pack`, or `CCS` format.

You can speed up the FFT on Intel® processors with SIMD instructions, if assign an external buffer. The external buffer increases performance because it allows to avoid allocation and deallocation of internal buffers and to store data in cache.

FFTInitAlloc_C, FFTInitAlloc_R

Initializes the fast Fourier transform structure for real and complex signals.

```
IppStatus ippsFFTInitAlloc_R_16s(IppsFFTSpec_R_16s** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_16s(IppsFFTSpec_C_16s** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_16sc(IppsFFTSpec_C_16sc** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_R_32f(IppsFFTSpec_R_32f** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
```

```
IppStatus ippsFFTInitAlloc_C_32f(IppsFFTSpec_C_32f** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32fc(IppsFFTSpec_C_32fc** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_R_64f(IppsFFTSpec_R_64f** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64f(IppsFFTSpec_C_64f** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64fc(IppsFFTSpec_C_64fc** pFFTSpec,
     int order, int flag, IppHintAlgorithm hint);
```

Arguments

flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
order	FFT order. The input signal length is $N=2^{order}$.
pFFTSpec	Pointer to the FFT specification structure to be created.

Discussion

The functions ippsFFTInitAlloc_C and ippsFFTInitAlloc_R are declared in the ipps.h file. These functions create and initialize the FFT specification structure *pFFTSpec* with the following parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The *order* argument defines the transform length. Thus the input and output signals are 2^{order}-length arrays.

ippsFFTInitAlloc_C. The function ippsFFTInitAlloc_C initializes the complex FFT specification structure.

ippsFFTInitAlloc_R. The function ippsFFTInitAlloc_R initializes the real FFT specification structure.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pFFTSpec* pointer is `NULL`.
`ippStsFftOrderErr`	Indicates an error when the *order* value is incorrect.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

FFTFree_C, FFTFree_R

Closes a fast Fourier transform structure for real and complex signals.

```
IppStatus ippsFFTFree_R_16s(IppsFFTSpec_R_16s* pFFTSpec);
IppStatus ippsFFTFree_C_16s(IppsFFTSpec_C_16s* pFFTSpec);
IppStatus ippsFFTFree_C_16sc(IppsFFTSpec_C_16sc* pFFTSpec);

IppStatus ippsFFTFree_R_32f(IppsFFTSpec_R_32f* pFFTSpec);
IppStatus ippsFFTFree_C_32f(IppsFFTSpec_C_32f* pFFTSpec);
IppStatus ippsFFTFree_C_32fc(IppsFFTSpec_C_32fc* pFFTSpec);

IppStatus ippsFFTFree_R_64f(IppsFFTSpec_R_64f* pFFTSpec);
IppStatus ippsFFTFree_C_64f(IppsFFTSpec_C_64f* pFFTSpec);
IppStatus ippsFFTFree_C_64fc(IppsFFTSpec_C_64fc* pFFTSpec);
```

Arguments

pFFTSpec	Pointer to the FFT specification structure to be closed.

Discussion

The function ippsFFTFree is declared in the ipps.h file. This function closes the FFT specification structure *pFFTSpec* by freeing all memory associated with the specification created by ippsFFTInit_C or ippsFFTInit_R. Call ippsFFTFree after the transform is completed.

ippsFFTFree_C. The function ippsFFTFree_C closes the complex FFT specification structure.

ippsFFTFree_R. The function ippsFFTFree_R closes the real FFT specification structure.

Return Value

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

FFTGetBufSize_C, FFTGetBufSize_R

Gets the size of the FFT work buffer in bytes.

```
IppStatus ippsFFTGetBufSize_R_16s(const IppsFFTSpec_R_16s* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_16s(const IppsFFTSpec_C_16s* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_16sc(const IppsFFTSpec_C_16sc* pFFTSpec,
          int* pSize);


IppStatus ippsFFTGetBufSize_R_32f(const IppsFFTSpec_R_32f* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_32f(const IppsFFTSpec_C_32f* pFFTSpec,
     int* pSize);
```

```
IppStatus ippsFFTGetBufSize_C_32fc(const IppsFFTSpec_C_32fc* pFFTSpec,
     int* pSize);

IppStatus ippsFFTGetBufSize_R_64f(const IppsFFTSpec_R_64f* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_64f(const IppsFFTSpec_C_64f* pFFTSpec,
     int* pSize);
IppStatus ippsFFTGetBufSize_C_64fc(const IppsFFTSpec_C_64fc* pFFTSpec,
     int* pSize);
```

Arguments

pFFTSpec	Pointer to the FFT specification structure.
pSize	Pointer to the FFT work buffer size value.

Discussion

The function ippsFFTGetBufSize is declared in the ipps.h file. This function gets the work buffer size of the FFT described by the specification structure *pFFTSpec* in bytes and stores in *pSize*.

ippsFFTGetBufSize_C. The function ippsFFTGetBufSize_C gets the size of the complex FFT work buffer.

ippsFFTGetBufSize_R. The function ippsFFTGetBufSize_R gets the size of the real FFT work buffer.

Return Value

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

FFTFwd_CToC, FFTInv_CToC

Computes the forward or inverse fast Fourier transform (FFT) of a complex signal.

```
IppStatus ippsFFTFwd_CToC_32f(const Ipp32f* pSrcRe,
      const Ipp32f* pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm,
      const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32f(const Ipp32f* pSrcRe,
      const Ipp32f* pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm,
      const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64f(const Ipp64f* pSrcRe,
      const Ipp64f* pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm,
      const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64f(const Ipp64f* pSrcRe,
      const Ipp64f* pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm,
      const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      const IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
      const IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      const IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
      const IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe,
      const Ipp16s* pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm,
      const IppsFFTSpec_C_16s* pFFTSpecx, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe,
      const Ipp16s* pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm,
      const IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     const IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     const IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pBuffer	Pointer to the FFT work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions `ippsFFTFwd_CToC` and `ippsFFTInv_CToC` are declared in the `ipps.h` file. These functions compute the forward or inverse FFT of a complex signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, e.g., with the `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with the `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the FFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is same for both consecutive operations.

ippsFFTFwd_CToC. The function ippsFFTFwd_CToC computes a complex forward FFT. The length of the FFT must be a power of 2.

ippsFFTInv_CToC. The function ippsFFTInv_CToC computes a complex inverse FFT. The length of the FFT must be a power of 2.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

FFTFwd_RToPerm, FFTInv_PermToR, FFTFwd_RToPack, FFTInv_PackToR, FFTFwd_RToCCS, FFTInv_CCSToR

Computes the forward or inverse fast Fourier transform (FFT) of a real signal.

```
IppStatus ippsFFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pFFTSpec	Pointer to the FFT specification structure.

pSrc	Pointer to the input array containing real values for the forward transform and packed complex values resulted from the inverse transform.
pDst	Pointer to the output array containing real values for the inverse transform and packed complex values resulted from the forward transform.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

These functions are declared in the `ipps.h` file. They compute the forward or inverse FFT of a real signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*.

The result of the forward transform (i.e., in the frequency-domain) of real signals is represented in several possible packed formats: `Pack`, `Perm`, or `CCS`. The data can be packed due to the symmetry property of the FFT transform of a real signal.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the FFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the FFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

ippsFFTFwd_RToPerm, ippsFFTInv_PermToR. These functions compute the forward or inverse FFT and store the result in `Perm` format. The length of the FFT must be a power of 2.

ippsFFTFwd_RToPack, ippsFFTInv_PackToR. These functions compute the forward or inverse FFT and store the result in `Pack` format. The length of the FFT must be a power of 2.

ippsFFTFwd_RToCCS, ippsFFTInv_CCSToR. These functions compute the forward or inverse FFT and store the result in `CCS` format. The length of the FFT must be a power of 2.

Table 7-8 shows how the output result is represented in each of the packed format: `Pack`, `Perm`, and `CCS`. Example 7-5 shows how to initialize the specification and call `ippsFFTFwd_RToCCS_32f`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

Table 7-8 Forward FFT Result Representation in Pack, Perm, and CCS Formats

FFTReal	**0**	**1**	**2**	**3**	**...**	**N-2**	**N-1**	**N**	**N+1**
Pack	R_0	R_1	I_1	R_2	. . .	$I_{N/2-1}$	$R_{N/2}$		
Perm	R_0	$R_{N/2}$	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	R_0	0	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

Example 7-5 Using the ippsFFTFwd_RToCCS Function

```
IppStatus fft( void ) {
     Ipp32f x[8], X[10];
     int n;
     IppStatus status;
     IppsFFTSpec_R_32f* spec;
     status = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
         ippAlgHintNone );
     for(n=0; n<8; ++n) x[n] = (float)cos(IPP_2PI *n *16/64);
     status = ippsFFTFwd_RToCCS_32f( x, X, spec, NULL );
     ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
     ippsFFTFree_R_32f( spec );
     printf_32f("fft magn =", x, 4, status );
     return status;
}
Output:
    fft magn = 0.000000 0.000000 4.000000 0.000000
Matlab* Analog:
    >> n=0:7; x=sin(2*pi*n*16/64); X=abs(fft(x)); X(1:4)
```

Discrete Fourier Transform Functions

The functions described in this section compute the forward and inverse discrete Fourier transform of real and complex signals. The DFT is less efficient than the fast Fourier transform, however the length of the vector transformed by the DFT can be arbitrary.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method. The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in `Perm`, `Pack`, or `CCS` format.

To use the DFT functions, you should initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms.

For more information about the fast computation of the discrete Fourier transform, see [Mit93], section 8-2, *Fast Computation of the DFT.*

A special set of Intel IPP functions provides the so called “out-of-order” DFT of the complex signal. In this case, the elements in frequency domain for both forward and inverse transforms can be re-ordered to speed-up the computation of the transforms. This re-ordering is hidden from the user and can be different in different implementations of the functions. However, reversibility of each pair of functions for forward/inverse transforms is ensured.

DFTInitAlloc_C, DFTInitAlloc_R

Initializes the discrete Fourier transform structure for real and complex signals.

```
IppStatus ippsDFTInitAlloc_R_16s(IppsDFTSpec_R_16s** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_16s(IppsDFTSpec_C_16s** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_16sc(IppsDFTSpec_C_16sc** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_R_32f(IppsDFTSpec_R_32f** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_32f(IppsDFTSpec_C_32f** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_32fc(IppsDFTSpec_C_32fc** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_R_64f(IppsDFTSpec_R_64f** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_C_64f(IppsDFTSpec_C_64f** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_64fc(IppsDFTSpec_C_64fc** pDFTSpec,
     int length, int flag, IppHintAlgorithm hint);
```

Arguments

flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
length	Length of the DFT transform.
pDFTSpec	Pointer to the DFT specification structure to be created.

Discussion

The functions ippsDFTInitAlloc_C and ippsDFTInitAlloc_R are declared in the ipps.h file. These functions create and initialize the DFT specification structure *pDFTSpec* with the following parameters: the transform *length*, the normalization *flag*, and the specific code *hint*. The *length* argument defines the transform length.

ippsDFTInitAlloc_C. The function ippsDFTInitAlloc_C initializes the complex DFT specification structure.

ippsDFTInitAlloc_R. The function ippsDFTInitAlloc_R initializes the real DFT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDFTSpec* pointer is NULL.
ippStsSizeErr	Indicates an error when *length* is less than or equal to 0.
ippStsFftFlagErr	Indicates an error when the *flag* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DFTFree_C, DFTFree_R

Closes the discrete Fourier transform structure for real and complex signals.

```
IppStatus ippsDFTFree_R_16s(IppsDFTSpec_R_16s* pDFTSpec);
IppStatus ippsDFTFree_C_16s(IppsDFTSpec_C_16s* pDFTSpec);
IppStatus ippsDFTFree_C_16sc(IppsDFTSpec_C_16sc* pDFTSpec);

IppStatus ippsDFTFree_R_32f(IppsDFTSpec_R_32f* pDFTSpec);
IppStatus ippsDFTFree_C_32f(IppsDFTSpec_C_32f* pDFTSpec);
IppStatus ippsDFTFree_C_32fc(IppsDFTSpec_C_32fc* pDFTSpec);

IppStatus ippsDFTFree_R_64f(IppsDFTSpec_R_64f* pDFTSpec);
IppStatus ippsDFTFree_C_64f(IppsDFTSpec_C_64f* pDFTSpec);
IppStatus ippsDFTFree_C_64fc(IppsDFTSpec_C_64fc* pDFTSpec);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure to be closed.

Discussion

The function ippsDFTFree is declared in the ipps.h file. This function closes the DFT specification structure *pDFTSpec* by freeing all memory associated with the specification created by ippsDFTInitAlloc_C or ippsDFTInitAlloc_R. Call ippsDFTFree after the transform is completed.

ippsDFTFree_C. The function ippsDFTFree_C closes the complex DFT specification structure.

ippsDFTFree_R. The function ippsDFTFree_R closes the real DFT specification structure.

Return Value

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

DFTGetBufSize_C, DFTGetBufSize_R

Gets the size of the DFT work buffer in bytes.

```
IppStatus ippsDFTGetBufSize_R_16s(const IppsDFTSpec_R_16s*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_16s(const IppsDFTSpec_C_16s*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_16sc(const IppsDFTSpec_C_16sc* pDFTSpec,
      int* pSize);

IppStatus ippsDFTGetBufSize_R_32f(const IppsDFTSpec_R_32f*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_32f(const IppsDFTSpec_C_32f*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_32fc(const IppsDFTSpec_C_32fc* pDFTSpec,
      int* pSize);

IppStatus ippsDFTGetBufSize_R_64f(const IppsDFTSpec_R_64f*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_64f(const IppsDFTSpec_C_64f*  pDFTSpec,
      int* pSize);
IppStatus ippsDFTGetBufSize_C_64fc(const IppsDFTSpec_C_64fc* pDFTSpec,
      int* pSize);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure.

pSize	Pointer to the DFT work buffer size value.

Discussion

The function ippsDFTGetBufSize is declared in the ipps.h file. This function computes the size in bytes of an external memory buffer for the DFT described by the specification structure *pDFTSpec* and stores it in *pSize*.

To use external buffering, call this function after *pDFTSpec* initialization.

ippsDFTGetBufSize_C. The function ippsDFTGetBufSize_C gets the size of the complex DFT work buffer.

ippsDFTGetBufSize_R. The function ippsDFTGetBufSize_R gets the size of the real DFT work buffer.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDFTSpec* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.

DFTFwd_CToC, DFTInv_CToC

Computes the forward or inverse discrete Fourier transform of a complex signal.

```
IppStatus ippsDFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f*
     pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f*
     pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f*
     pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f*
     pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f*
     pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f*
     pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f*
     pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f*
     pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     const IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     const IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     const IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     const IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
     pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s*
     pDFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsDFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
     pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s*
     pDFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     const IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsDFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
     const IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.

pBuffer	Pointer to the work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions ippsDFTFwd_CToC and ippsDFTInv_CToC are declared in the ipps.h file. These functions compute the forward or inverse DFT of a complex signal according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, e.g., with 32fc suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, e.g., with 32f suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the correspondent function ippsDFTGetBufSize_C prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j 2\pi \frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *length*, and A and B are multipliers defined by *flag*. Also, in the forward direction, $x(n)$ is *pSrc[n]* and $X(k)$ is *pDst[k]*; in the inverse direction, $x(n)$ is *pDst[n]* and $X(k)$ is *pSrc[k]*.

The definition of the inverse discrete Fourier transform is:

$$x(k) = B \sum_{n=0}^{N-1} X(n) \cdot \exp\left(j 2\pi \frac{kn}{N}\right)$$

ippsDFTFwd_CToC. The function ippsDFTFwd_CToC computes the complex forward DFT.

ippsDFTInv_CToC. The function ippsDFTInv_CToC computes the complex inverse DFT.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DFTFwd_RToPerm, DFTInv_PermToR, DFTFwd_RToPack, DFTInv_PackToR, DFTFwd_RToCCS, DFTInv_CCSToR

Computes the forward or inverse discrete Fourier transform of a real signal.

```
IppStatus ippsDFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
     const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing complex values for the forward transform and packed complex values resulted from the inverse transform.

pDst	Pointer to the output array containing real values for the inverse transform and packed complex values resulted from the forward transform.
pBuffer	Pointer to the work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

These functions are declared in the `ipps.h` file. They compute the forward or inverse DFT of a real signal according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

The result of the forward transform (i.e. in the frequency-domain) of real signals is represented in several possible packed formats: `Pack`, `Perm`, or `CCS`. The data can be packed due to the symmetry property of the DFT transform of a real signal.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The *pBuffer* argument provides the DFT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer allows to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the correspondent function `ippsDFTGetBufSize_R` prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j 2\pi \frac{kn}{N}\right),$$

where k is the index of elements in the frequency domain, n is the index of elements in the time domain, N is the input signal *length*, and A and B are multipliers defined by *flag*. Also, in the forward direction, $x(n)$ is *pSrc[n]* and $X(k)$ is *pDst[k]*; in the inverse direction, $x(n)$ is *pDst[n]* and $X(k)$ is *pSrc[k]*.

The definition of the inverse discrete Fourier transform is:

$$x(k) = B \sum_{n=0}^{N-1} X(n) \cdot \exp\left(j 2\pi \frac{kn}{N}\right)$$

ippsDFTFwd_RToPerm, ippsDFTInv_PermToR. These functions compute the forward or inverse DFT and store the result in `Perm` format.

ippsDFTFwd_RToPack, ippsDFTInv_PackToR. These functions compute the forward or inverse DFT and store the result in `Pack` format.

ippsDFTFwd_RToCCS, ippsDFTInv_CCSToR. These functions compute the forward or inverse DFT and store the result in `CCS` format.

Example 7-6 shows how to initialize the specification and call `ippsDFTFwd_RToCCS_32f`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *`pDFTSpec`* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

Example 7-6 Using the ippsDFTFwd_RToCCS Function

```
IppStatus dft( void ) {
     Ipp32f x[7], X[8];
     int n;
     IppStatus status;
     IppsDFTSpec_R_32f* spec;
     status = ippsDFTInitAlloc_R_32f(&spec, 7, IPP_FFT_DIV_INV_BY_N,
          ippAlgHintNone);
     for( n=0; n<7; ++n ) x[n] = (float) cos(IPP_2PI * n * 14 / 49);
     status = ippsDFTFwd_RToCCS_32f( x, X, spec, NULL );
     ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
     ippsDFTFree_R_32f( spec );
     printf_32f("dft magn =", x, 4, status );
     return status;
}
Output:
    dft magn =  0.000000 0.000000 3.500000 0.000000
Matlab* analog:
    >> N=7;F=14/49;n=0:N-1;x=cos(2*pi*n*F);y=abs(fft(x));y(1:4)
```

DFTOutOrdInitAlloc_C

Initializes the out-of-order discrete Fourier transform structure.

```
IppStatus ippsDFTOutOrdInitAlloc_C_32fc(IppsDFTOutOrdSpec_C_32fc**
     pDFTSpec, int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTOutOrdInitAlloc_C_64fc(IppsDFTOutOrdSpec_C_64fc**
     pDFTSpec, int length, int flag, IppHintAlgorithm hint);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure to be created.
flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
length	Length of the DFT transform.

Discussion

The function ippsDFTOutOrdInitAlloc_C is declared in the ipps.h file. This function creates and initializes the specification structure *pDFTSpec* for the out-of-order DFT with the following parameters: the transform *length*, the normalization *flag*, and the specific code *hint*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFFTSpec* pointer is NULL.
ippStsSizeErr	Indicates an error when *length* is less than or equal to 0.
ippStsFftFlagErr	Indicates an error when the *flag* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DFTOutOrdFree_C

Closes the out-of-order discrete Fourier transform structure.

```
IppStatus ippsDFTOutOrdFree_C_32fc(IppsDFTOutOrdSpec_C_32fc* pDFTSpec);
IppStatus ippsDFTOutOrdFree_C_64fc(IppsDFTOutOrdSpec_C_64fc* pDFTSpec);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure to be closed.

Discussion

The function ippsDFTOutOrdFree is declared in the ipps.h file. This function closes the specification structure *pDFTSpec* for the out-of-order DFT by freeing all memory associated with the specification created by ippsDFTOutOrdInitAlloc_C. Call ippsDFTOutOrdFree function after the transform is completed.

Return Value

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

DFTOutOrdGetBufSize_C

Computes the size of the work buffer for out-of-order DFT.

```
IppStatus ippsDFTOutOrdGetBufSize_C_32fc(const
     IppsDFTOutOrdSpec_C_32fc* pDFTSpec, int* pSize);
IppStatus ippsDFTOutOrdGetBufSize_C_64fc(const
     IppsDFTOutOrdSpec_C_64fc* pDFTSpec, int* pSize);
```

Arguments

pDFTSpec	Pointer to the DFT specification structure.
pSize	Pointer to the DFT work buffer size value.

Discussion

The function ippsDFTOutOrdGetBufSize is declared in the ipps.h file. This function computes the size in bytes of an external memory buffer for the out-of-order DFT described by the specification structure *pDFTSpec,* and stores it in *pSize*.

To use external buffering, call this function after *pDFTSpec* initialization.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDFTSpec* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.

DFTOutOrdFwd_CToC, DFTOutOrdInv_CToC

Computes the forward or inverse out-of-order discrete Fourier transform.

```
IppStatus ippsDFTOutOrdFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     const IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     const IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
     const IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
     const IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

Arguments

pSrc	Pointer to the source data.
pDst	Pointer to the output data.
pDFTSpec	Pointer to the DFT specification structure.

pBuffer	Pointer to the work buffer.

Discussion

The functions ippsDFTOutOrdFwd_CToC and ippsDFTOutOrdInv_CToC are declared in the ipps.h file. These functions compute the forward or inverse out-of-order DFT of a complex signal *pSrc* according to the *pDFTSpec* specification parameters: the transform *length*, the normalization *flag*, and the specific code *hint*, and store the result in *pDst*.

These functions are analogous to ippsDFTFwd_CToC and ippsDFTInv_CToC and have the similar functionality. The difference is that the elements in frequency domain (*pDst* for forward transforms and *pSrc* for inverse transforms) can be rearranged if this helps to speed up computations. This procedure is hidden from the user and depends on the specific implementation of the functions.However, the reversibility of each pair of functions for forward/inverse transforms is ensured.

The *pBuffer* argument provides the DFT functions with the necessary workspace memory and allows to avoid memory allocation within the functions. The buffer helps to increase performance if the DFT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations. Required buffer size should be computed by the function ippsDFTOutOrdGetBufSize_C prior to using the DFT computation functions. If a NULL pointer is passed, memory will be allocated by the DFT computation functions internally.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pDFTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DFT for a Given Frequency (Goertzel) Functions

The functions described in this section compute a single or a number of the discrete Fourier transforms for a given frequency. Note that the DFT exists only for the following normalized frequencies: 0, 1/*N*, 2/*N*,... (*N*-1)/*N*, where *N* is the number of time domain samples. Therefore you must select the frequency value from the above set.

These Intel IPP functions use a Goertzel algorithm [Mit98] and are more efficient when a small number of DFT values is needed.

Some of the functions compute two values, not one. The applications computing several values, e.g., the dual-tone multifrequency signal detection, work faster, especially on Intel® processors with SIMD instructions.

Goertz

Computes the discrete Fourier transform for a given frequency for a single complex signal.

```
IppStatus ippsGoertz_32f(const Ipp32f* pSrc, int len, Ipp32fc* pVal,
      Ipp32f freq);
IppStatus ippsGoertz_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pVal,
      Ipp32f freq);
IppStatus ippsGoertz_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pVal,
      Ipp64f freq);
IppStatus ippsGoertz_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16sc* pVal,
      Ipp32f freq, int scaleFactor))
IppStatus ippsGoertz_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pVal,
      Ipp32f freq, int scaleFactor);
```

Arguments

freq	Pointer to the single relative frequency value [0, 1.0).
pSrc	Pointer to the input complex data vector.

len	Number of elements in the vector.
pVal	Pointer to the output DFT value.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsGoertz is declared in the ipps.h file. This function computes a DFT for a complex input *len*-length signal *pSrc* for a given frequency *freq*, and stores the result in *pVal*.

The ippsGoertz functionality can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where *k/N* is the normalized *freq* value for which the DFT is computed.

Example 7-7 illustrates the use of Goertzel functions for selecting the magnitudes of a given frequency when computing DFTs.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsRelFreqErr	Indicates an error when *freq* is out of range.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.

Example 7-7 Using Goertzel Functions for Selecting Magnitudes of a Given Frequency

```
IppStatus goertzel( void ) {
#undef  LEN
#define LEN 100
     IppStatus status;
     Ipp32fc *x = ippsMalloc_32fc( LEN ), y;
     int n;
     ///generate a signal of 60 Hz freq that
     /// is sampled with 400 Hz freq
     for( n=0; n<LEN; ++n) {
        x[n].re =(Ipp32f)sin(IPP_2PI * n * 60 / 400);
        x[n].im = 0;
     }
     status = ippsGoertz_32fc( x, LEN, &y, 60.0f / 400 );
     printf_32fc("goertz =", &y, 1, status );
     ippsFree( x );
     return status;
}
Output:
    goertz =  {0.000090,-50.000008}
Matlab* Analog
    >> N=100;F=60/400;n=0:N-1;x=sin(2*pi*n*F);y=fft(x);n=N*F;y(n+1)
```

GoertzTwo

Computes two discrete Fourier transforms for a given frequency for a single complex signal.

```
IppStatus ippsGoertzTwo_32fc(const Ipp32fc* pSrc, int len,
      Ipp32fc pVal[2], const Ipp32f freq[2]);
IppStatus ippsGoertzTwo_64fc(const Ipp64fc* pSrc, int len,
      Ipp64fc pVal[2], const Ipp64f freq[2]);
IppStatus ippsGoertzTwo_16sc_Sfs(const Ipp16sc* pSrc, int len,
      Ipp16sc pVal[2], const Ipp32f freq[2], int scaleFactor);
```

Arguments

freq	Pointer to two single relative frequency value [0, 1.0).
pSrc	Pointer to the input complex data vector.
len	Number of elements in the vector.
pVal	Pointer to the output DFT values.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsGoertzTwo is declared in the ipps.h file. This function computes two DFTs for a complex input *len* -length signal *pSrc* for two given frequencies *freq*, and stores the result in *pVal*. The computation of two DFTs on an Intel® Pentium® III processor is performed at the same speed as one. Therefore, the applications computing several DFTs are faster.

The ippsGoertz functionality can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where *k/N* is one of the normalized *freq* values for which the DFTs are computed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are NULL.
ippStsRelFreqErr	Indicates an error when *freq* is out of range.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.

Discrete Cosine Transform Functions

This section describes the functions that compute the discrete cosine transform of a signal. DCT functions used in the Intel IPP signal processing data-domain implement the modified computation algorithm proposed in [Rao90].

DCTFwdInitAlloc , DCTInvInitAlloc

Initializes the discrete cosine transform structure.

```
IppStatus ippsDCTFwdInitAlloc_16s(IppsDCTFwdSpec_16s** pDCTSpec,
     int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_16s(IppsDCTInvSpec_16s** pDCTSpec,
     int length, IppHintAlgorithm hint);

IppStatus ippsDCTFwdInitAlloc_32f(IppsDCTFwdSpec_32f** pDCTSpec,
     int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_32f(IppsDCTInvSpec_32f** pDCTSpec,
     int length, IppHintAlgorithm hint);

IppStatus ippsDCTFwdInitAlloc_64f(IppsDCTFwdSpec_64f** pDCTSpec,
     int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_64f(IppsDCTInvSpec_64f** pDCTSpec,
     int length, IppHintAlgorithm hint);
```

Arguments

flag	Specifies the result normalization method. The values for the *flag* argument are described in "Flag and Hint Arguments."
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."
length	Number of samples in the DCT.
pDCTSpec	Pointer to the DCT specification structure to be created.

Discussion

The functions ippsDCTFwdInitAlloc and ippsDCTInvInitAlloc are declared in the ipps.h file. These functions create and initialize the DCT specification structure *pDCTSpec* with the following parameters: the transform *length*, the normalization *flag*, and the specific code *hint*. The *length* argument defines the transform length.

ippsDCTFwdInitAlloc. The function ippsDCTFwdInitAlloc initializes the forward DCT specification structure.

ippsDCTInvInitAlloc. The function ippsDCTInvInitAlloc initializes the inverse DCT specification structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDCTSpec* pointer is NULL.
ippStsSizeErr	Indicates an error when *length* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

DCTFwdFree, DCTInvFree

Closes a discrete cosine transform structure.

```
IppStatus ippsDCTFwdFree_16s(IppsDCTFwdSpec_16s* pDCTSpec);
IppStatus ippsDCTInvFree_16s(IppsDCTInvSpec_16s* pDCTSpec);
IppStatus ippsDCTFwdFree_32f(IppsDCTFwdSpec_32f* pDCTSpec);
IppStatus ippsDCTInvFree_32f(IppsDCTInvSpec_32f* pDCTSpec);
IppStatus ippsDCTFwdFree_64f(IppsDCTFwdSpec_64f* pDCTSpec);
IppStatus ippsDCTInvFree_64f(IppsDCTInvSpec_64f* pDCTSpec);
```

Arguments

pDCTSpec	Pointer to the DCT specification structure to be closed.

Discussion

The functions ippsFFTFwdFree and ippsFFTInvFree are declared in the ipps.h file. These functions close the DCT specification structure *pDCTSpec* by freeing all memory associated with the specification created by ippsDCTFwdInitAlloc or ippsDCTInvInitAlloc. Call either ippsDCTFwdFree or ippsDCTInvFree after the transform is completed.

ippsDCTFwdFree. The function ippsDCTFwdFree closes the forward DCT specification structure.

ippsDCTInvFree. The function ippsDCTInvFree closes the inverse DCT specification structure.

Return Value

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

DCTFwdGetBufSize, DCTInvGetBufSize

Gets the size of the DCT work buffer in bytes.

```
IppStatus ippsDCTFwdGetBufSize_16s(const IppsDCTFwdSpec_16s* pDCTSpec,
     int* pSize);
IppStatus ippsDCTInvGetBufSize_16s(const IppsDCTInvSpec_16s* pDCTSpec,
     int* pSize);

IppStatus ippsDCTFwdGetBufSize_32f(const IppsDCTFwdSpec_32f* pDCTSpec,
     int* pSize);
IppStatus ippsDCTInvGetBufSize_32f(const IppsDCTInvSpec_32f* pDCTSpec,
     int* pSize);

IppStatus ippsDCTFwdGetBufSize_64f(const IppsDCTFwdSpec_64f* pDCTSpec,
     int* pSize);
IppStatus ippsDCTInvGetBufSize_64f(const IppsDCTInvSpec_64f* pDCTSpec,
     int* pSize);
```

Arguments

pDCTSpec	Pointer to the DCT specification structure.
pSize	Pointer to the DCT work buffer size value.

Discussion

The functions `ippsDCTFwdGetBufSize` and `ippsDCTInvGetBufSize` are declared in the `ipps.h` file. These functions get the work buffer size of the DCT described by the specification structure *pDCTSpec* in bytes and stores in *pSize*.

`ippsDCTFwdGetBufSize`. The function `ippsDCTFwdGetBufSize` gets the work buffer size of the forward DCT.

`ippsDCTInvGetBufSize`. The function `ippsDCTInvGetBufSize` gets the work buffer size of the inverse DCT.

Return Value

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

DCTFwd, DCTInv

Computes the forward or inverse discrete cosine transform of a signal.

```
IppStatus ippsDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDCTFwdSpec_32f* pDCTSpec, Ipp8u* pBuffer);
IppStatus ippsDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst,
     const IppsDCTInvSpec_32f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippsDCTFwd_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDCTFwdSpec_64f* pDCTSpec, Ipp8u* pBuffer);
IppStatus ippsDCTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst,
     const IppsDCTInvSpec_64f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippsDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
     IppsDCTFwdSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDCTInv_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
     IppsDCTInvSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

Arguments

pDCTSpec	Pointer to the DCT specification structure.
pSrc	Pointer to the input data array.
pDst	Pointer to the output data array.

pBuffer	Pointer to the DCT work buffer.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions ippsDCTFwd and ippsDCTInv are declared in the ipps.h file. These functions compute the forward and inverse discrete cosine transform (DCT). If *length* is a power of 2, the functions use an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *length*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT, $N = length$,

$C(k) = \frac{1}{\sqrt{N}}$ for $k = 0$, $C(k) = \frac{\sqrt{2}}{\sqrt{N}}$ for $k > 0$;

$x(n)$ is *pSrc[n]* and $y(k)$ is *pDst[k]* for the forward DCT;

$x(n)$ is *pDst[n]* and $y(k)$ is *pSrc[k]* for the inverse DCT.

The forward DCT is defined by the formula:

$$y(k) = C(k) \sum_{n=0}^{N-1} x(n) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

The definition of the inverse discrete cosine transform is:

$$x(n) = \sum_{k=0}^{N-1} C(k)y(k) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained. The *pBuffer* argument provides the DCT functions with the necessary working memory and allows to avoid memory allocation within the functions. The buffer may also increase up performance if the DCT functions use the result of the previous operation stored in cache as an input array. The buffer is the same for both consecutive operations.

ippsDCTFwd. The function ippsDCTFwd computes a forward DCT.

ippsDCTInv. The function ippsDCTInv computes an inverse DCT.

Example 7-8 shows how to use the functions ippsDCTFwd_32f and ippsDCTInv_32f to compress and reconstruct a signal.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pDCTSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDCTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

Example 7-8 Using ippsDCTFwd and ippsDCTInv to Compress and Reconstruct a Signal

```
        void dct( void ) {
        #define LEN 256
             Ipp32f x[LEN], y[LEN];
             int n;
             IppsDCTFwdSpec_32f* fspec;
             IppsDCTInvSpec_32f* ispec;
             IppStatus status;
             /// data: Gaussian function, magn =1 and sigma=N/3
             for(n=0; n<LEN; ++n)
                 x[n] = (float)(exp(-0.5*(LEN/2-n)*(LEN/2-n)/(LEN/3.0)));
             /// get cosine transform coefficients
             status = ippsDCTFwdInitAlloc_32f( &fspec, LEN, ippAlgHintNone );
             status = ippsDCTFwd_32f( x, y, fspec, NULL );
             ippsDCTFwdFree_32f( fspec );
             /// Set 3/4 of these coefficients to zero
             for(n=LEN/4; n<LEN; ++n) y[n] = 0.0f;
             /// restore signal using len/4 values
             status = ippsDCTInvInitAlloc_32f( &ispec, LEN, ippAlgHintNone );
             status = ippsDCTInv_32f( y, x, ispec, NULL );
             ippsDCTInvFree_32f( ispec );
}
```

Hilbert Transform Functions

The functions described in this section compute a discrete-time analytic signal from a real data sequence using the Hilbert transform. The analytic signal is a complex signal whose real part is a replica of the original data, and imaginary part contains the Hilbert transform. That is, the imaginary part is a version of the original real data with a 90 degrees phase shift. The Hilbert transformed data have the same amplitude and frequency content as the original real data, plus the additional phase information.

HilbertInitAlloc

Initializes the Hilbert transform structure.

```
IppStatus ippsHilbertInitAlloc_32f32fc(IppsHilbertSpec_32f32fc** pSpec,
     int length, IppHintAlgorithm hint);
IppStatus ippsHilbertInitAlloc_16s32fc(IppsHilbertSpec_16s32fc** pSpec,
     int length, IppHintAlgorithm hint);
IppStatus ippsHilbertInitAlloc_16s16sc(IppsHilbertSpec_16s16sc** pSpec,
     int length, IppHintAlgorithm hint);
```

Arguments

pSpec	Pointer to pointer to the Hilbert context structure being initialized.
length	Number of samples in the Hilbert transform.
hint	Suggests using specific code for calculation. The values for the *hint* argument are described in "Flag and Hint Arguments."

Discussion

The function ippsHilbertInitAlloc is declared in the ipps.h file. This function creates and initializes the Hilbert specification structure *pSpec* with the following parameters: the length of the transform *length*, and the specific code indicator *hint*. Call this function before actually using the Hilbert transform function ippsHilbert.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSpec* pointer is NULL.
ippStsSizeErr	Indicates an error when *length* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

HilbertFree

Closes a Hilbert transform structure.

```
IppStatus ippsHilbertFree_32f32fc(IppsHilbertSpec_32f32fc* pSpec);
IppStatus ippsHilbertFree_16s32fc(IppsHilbertSpec_16s32fc* pSpec);
IppStatus ippsHilbertFree_16s16sc(IppsHilbertSpec_16s16sc* pSpec);
```

Arguments

pSpec	Pointer to the Hilbert specification structure to be closed.

Discussion

The function ippsHilbertFree is declared in the ipps.h file. This function closes the Hilbert specification structure *pSpec* by freeing all memory associated with the specification created by the function ippsHilbertInitAlloc. Call ippsHilbertFree after the transform is completed.

Return Value

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

Hilbert

Computes an analytic signal using the Hilbert transform.

```
IppStatus ippsHilbert_32f32fc(const Ipp32f* pSrc, Ipp32fc* pDst,
      IppsHilbertSpec_32f32fc* pSpec);
IppStatus ippsHilbert_16s32fc(const Ipp16s* pSrc, Ipp32fc* pDst,
      IppsHilbertSpec_16s32fc* pSpec);
IppStatus ippsHilbert_16s16sc_Sfs(const Ipp16s* pSrc, Ipp16sc* pDst,
      IppsHilbertSpec_16s16sc* pSpec, int scaleFactor);
```

Arguments

pSpec	Pointer to the Hilbert specification structure.
pSrc	Pointer to the vector containing original real data.
pDst	Pointer to the output array containing complex data.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsHilbert is declared in the ipps.h file. This function computes a complex analytic signal *pDst*, which contains the original real signal *pSrc* as its real part and computed Hilbert transform as its imaginary part. The Hilbert transform is performed according to the *pSpec* specification parameters: the number of samples *length*, and the specific code *hint*. The input data is zero-padded or truncated to the size of *length* as appropriate.
For integer data types, the output result is scaled according to the *scaleFactor* value, which ensures that the output signal range and precision are retained.

Example 7-9 shows how to initialize the specification structure and use the function ippsHilbert_32f32fc.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSpec* pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

Example 7-9 Using Hilbert Functions

```
IppStatus hilbert(void) {
    Ipp32f x[10];
    Ipp32fc y[10];
    int n;
    IppStatus status;
    IppsHilbertSpec_32f32fc* spec;

    status = ippsHilbertInitAlloc_32f32fc(&spec, 10, ippAlgHintNone);
    for(n = 0; n < 9; n ++) {
        x[n] = (Ipp32f)cos(IPP_2PI * i * 2 / 9);
    }
    status = ippsHilbert_32f32fc(x, y, spec);
    ippsMagnitude_32fc((Ipp32fc*)y, x, 5);
    ippsHilbertFree_32f32fc(spec);
    printf_32f("hilbert magn =", x, 5, status);
    return status;
}

Output:
hilbert magn = 1.0944 1.1214 1.0413 0.9707 0.9839

Matlab* Analog:
>> n=0:9; x=cos(2*pi*n*2/9); y=abs(hilbert(x)); y(1:5)
```

Wavelet Transform Functions

This section describes the wavelet transform functions implemented in Intel IPP.

In signal processing, signals can represented in both frequency and time-frequency domains. In many cases the wavelet transforms become an alternative to short time Fourier transforms.

The discrete wavelet signal can be considered as a set of the coefficients $a_{i,k}$ with two indices, one of which is a “frequency” characteristic and the other is a time localization. The coefficient value corresponds to the localized wave amplitude or to one of basis transform functions. The “frequency” index shows the time scale of the localized wave. Function bases originated from one local wave by decreasing the wave by 2^n in time are the most widely used. Such transforms can be used for building very efficient implementations called fast wavelet transforms by analogy with fast Fourier transforms. Figure 7-1 shows how the time and frequency plane is divided into areas that correspond to the local wave amplitudes. This kind of transforms is implemented in the Intel IPP and referred to as the discrete wavelet transform (DWT).

The DWT is one of the wavelet analysis methods that stem from the basis functions related to the scale factor 2. Thus, there is a basic common element shared by the DWT and the other packet analysis methods.

Likewise another basic element for signal reconstruction or synthesis can be defined, called the one-level inverse DWT. Figure 7-2 shows the diagram of the forward DWT which allows to switch to time-frequency representation shown in Figure 7-1. The diagram includes three levels of decomposition. Figure 7-3 shows the corresponding procedure of signal reconstruction based on the elementary one-level inverse transform.

The implementation of discrete multi-scale transforms is based on the use of interpolation and decimation filters with the resampling factor 2. The basis of the multi-scale signal decomposition and reconstruction functions uniquely defines the filter parameters. The Intel IPP multi-scale transform functions use filters with finite impulse response.

The Primitives contains two sets of functions.

- Transforms designed for fixed filter banks. These transforms yield the highest performance.
- Transforms that enable the user to work with arbitrary filters. These functions use effective polyphase filtration algorithms. The transform interface gives the option of processing the data in blocks, including in real-time applications.

Figure 7-1 Wavelet Decomposition Coefficients in Time-Frequency Domain

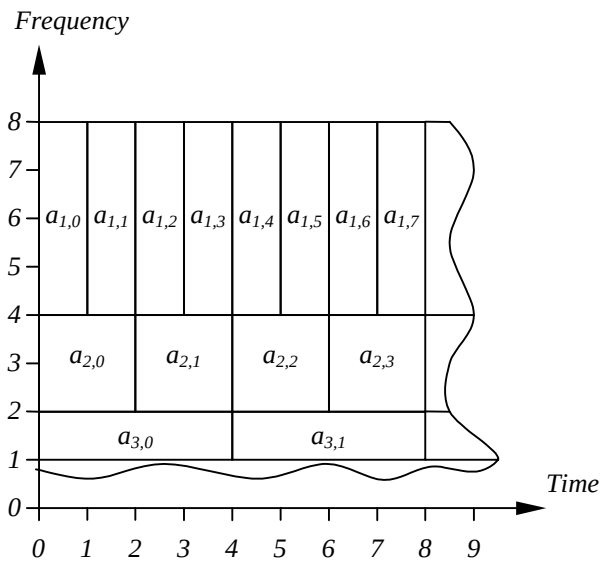


Figure 7-2 Three-Level Discrete Wavelet Decomposition

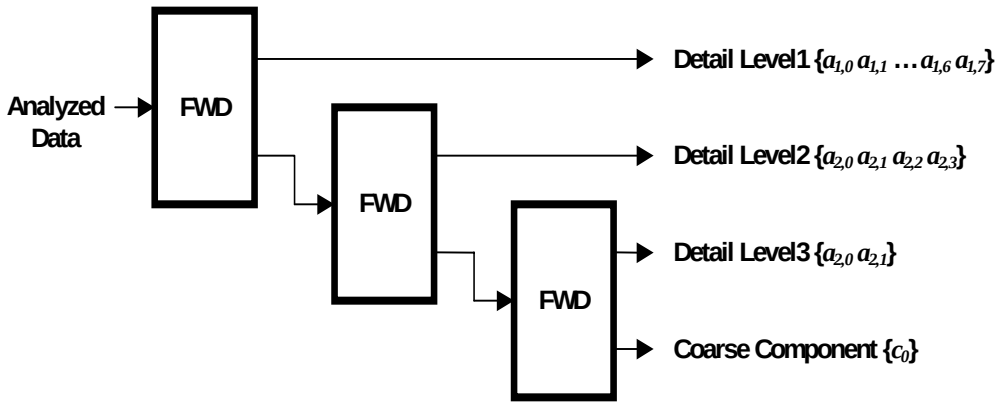
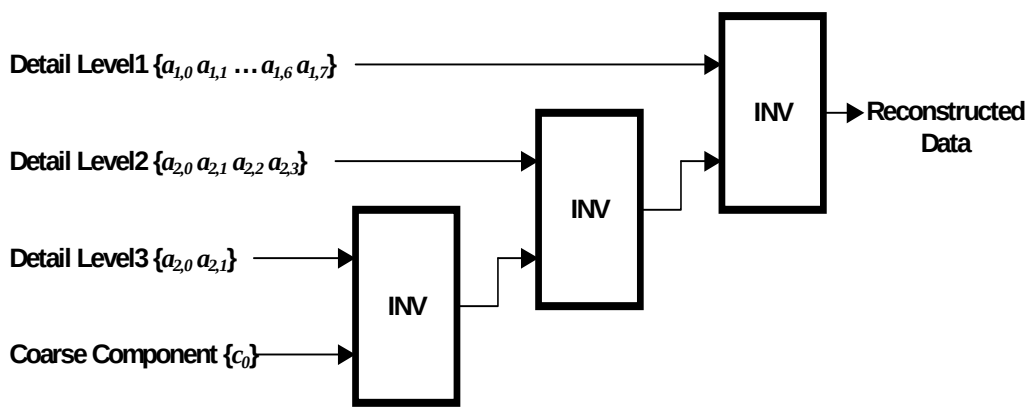


Figure 7-3 Three-Level Discrete Wavelet Reconstruction



Transforms for Fixed Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for fixed filter banks.

WTHaarFwd, WTHaarInv

Performs forward or inverse single-level discrete wavelet Haar transforms.

```
IppStatus ippsWTHaarFwd_8s(const Ipp8s* pSrc, int lenSrc,
      Ipp8s* pDstLow, Ipp8s* pDstHigh);
IppStatus ippsWTHaarFwd_16s(const Ipp16s* pSrc, int lenSrc,
      Ipp16s* pDstLow, Ipp16s* pDstHigh);
IppStatus ippsWTHaarFwd_32s(const Ipp32s* pSrc, int lenSrc,
      Ipp32s* pDstLow, Ipp32s* pDstHigh);
IppStatus ippsWTHaarFwd_64s(const Ipp64s* pSrc, int lenSrc,
      Ipp64s* pDstLow, Ipp64s* pDstHigh);
IppStatus ippsWTHaarFwd_32f(const Ipp32f* pSrc, int lenSrc,
      Ipp32f* pDstLow, Ipp32f* pDstHigh);
IppStatus ippsWTHaarFwd_64f(const Ipp64f* pSrc, int lenSrc,
      Ipp64f* pDstLow, Ipp64f* pDstHigh);
IppStatus ippsWTHaarFwd_8s_Sfs(const Ipp8s* pSrc, int lenSrc,
      Ipp8s* pDstLow, Ipp8s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
      Ipp16s* pDstLow, Ipp16s* pDstHigh, int scaleFactor );
IppStatus ippsWTHaarFwd_32s_Sfs(const Ipp32s* pSrc, int lenSrc,
      Ipp32s* pDstLow, Ipp32s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_64s_Sfs(const Ipp64s* pSrc, int lenSrc,
      Ipp64s* pDstLow, Ipp64s* pDstHigh, int scaleFactor);

IppStatus ippsWTHaarInv_8s(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh,
      Ipp8s* pDst, int lenDst);
```

```
IppStatus ippsWTHaarInv_16s(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh,
     Ipp16s* pDst, int lenDst);
IppStatus ippsWTHaarInv_32s(const Ipp32s* pSrcLow, const Ipp32s* pSrcHigh,
     Ipp32s* pDst, int lenDst);
IppStatus ippsWTHaarInv_64s(const Ipp64s* pSrcLow, const Ipp64s* pSrcHigh,
     Ipp64s* pDst, int lenDst);
IppStatus ippsWTHaarInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
     Ipp32f* pDst, int lenDst);
IppStatus ippsWTHaarInv_64f(const Ipp64f* pSrcLow, const Ipp64f* pSrcHigh,
     Ipp64f* pDst, int lenDst);
IppStatus ippsWTHaarInv_8s_Sfs(const Ipp8s* pSrcLow,
     const Ipp8s* pSrcHigh, Ipp8s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_16s_Sfs(const Ipp16s* pSrcLow,
     const Ipp16s* pSrcHigh, Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_32s_Sfs(const Ipp32s* pSrcLow,
     const Ipp32s* pSrcHigh, Ipp32s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_64s_Sfs(const Ipp64s* pSrcLow,
     const Ipp64s* pSrcHigh, Ipp64s* pDst, int lenDst, int scaleFactor);
```

Arguments

pSrc	Pointer to the array which holds the input signal for the ippsWTHaarFwd function.
lenSrc	Number of elements in the source vector *pSrc*.
pDstLow	Pointer to the array which holds coarse “low frequency” components of the output of the ippsWTHaarFwd function.
pDstHigh	Pointer to the array which holds detail “high frequency” components of the output of the ippsWTHaarFwd function.
pSrcLow	Pointer to the array which holds coarse “low frequency” components of the input to the ippsWTHaarInv function.
pSrcHigh	Pointer to the array which holds detail “high frequency” components of the input to the ippsWTHaarInv function.
pDst	Pointer to the array which holds the output signal of the ippsWTHaarInv function.

lenDst	Number of elements in the destination vector *pDst*
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions ippsWTHaarFwd and ippsWTHaarInv are declared in the ipps.h file. These functions perform forward and inverse single-level discrete Haar transforms. These transforms are orthogonal and reconstruct the original signal perfectly.

The forward transform can be considered as wavelet signal decomposition with lowpass decimation filter coefficients {1/2, 1/2} and highpass decimation filter coefficients {1/2, -1/2}.

The inverse transform is represented as wavelet signal reconstruction with lowpass interpolation filter coefficients {1, 1} and highpass interpolation filter coefficients {-1, 1}.

The decomposition filter coefficients are frequency response normalized to provide the same value range for both input and output signals. Thus, the amplitude of the low pass filter frequency response is 1 for zero-valued frequency, and the amplitude of the high pass filter frequency response is also 1 for the frequency value near to 0.5.

As the absolute values of the interpolation filter coefficients are equal to 1, the reconstruction of the signal requires few operations. It is well suited for usage in data compression applications. As the decomposition filter coefficients are powers of 2, the integer functions perform lossless decomposition with the *scaleFactor* value equal to -1. To avoid saturation, use higher-precision data types.

Note that the filter coefficients can be power spectral response normalized, see [Str96] for more information. Thus, the decomposition filter coefficients are $\{2^{-1/2}, 2^{-1/2}\}$ and $\{2^{-1/2}, -2^{-1/2}\}$; accordingly; the reconstruction filter coefficients are $\{2^{-1/2}, 2^{-1/2}\}$ and $\{-2^{-1/2}, 2^{-1/2}\}$.

In the following definition of the forward single-level discrete Haar transform, $N=lenSrc$. The coarse "low-frequency" component $c(k)$ is *pDstLow[k]* and the detail "high-frequency" component $d(k)$ is *pDstHigh[k]*; also $x(2k)$ and $x(2k+1)$ are even and odd values of the input signal *pSrc*, respectively.

$c(k)=(x(2k)+x(2k+1))/2$

$d(k)=(x(2k+1)-x(2k))/2$

In the inverse direction, $N=lenDst$. The coarse "low-frequency" component $c(k)$ is *pSrcLow[k]* and the detail "high-frequency" component $d(k)$ is *pSrcHigh[k]*; also $y(2i)$ and $y(2i+1)$ are even and odd values of the output signal *pDst*, respectively.

$y(2i)=c(i)-d(i)$

$y(2i+1)=c(i)+d(i)$

For even length N, $0 \le k < N/2$ and $0 \le i < N/2$. Also, "low-frequency" and "high-frequency" components are of size $N/2$ for both original and reconstructed signals. The total length of components is equal to the signal length N.

In case of odd length N, the vector is considered as a vector of the extended length $N+1$ whose two last elements are equal to each other $x[N]=x[N-1]$. The last elements of the coarse and detail components of the decomposed signal are defined as follows:

$c((N+1)/2-1)=x(N-1)$

$d((N+1)/2-1)=0$

Correspondingly, the last element of the reconstructed signal is defined as:

$y(N)=y(N-1)=c((N+1)/2-1)$

For odd length N, $0 \le k < (N-1)/2$ and $0 \le i < (N-1)/2$, assuming that $c((N+1)/2-1)=x(N-1)$ and $y(N-1)=c((N+1)/2-1)$. The "low-frequency" component is of size $(N+1)/2$. The "high-frequency" component is of size $(N-1)/2$, because the last element $d((N+1)/2-1)$ is always equal to 0. The total length of components is also N.

Such an approach applies continuation of boundaries for filters having the symmetry properties, see [Bri94].

When performing block mode transforms, take into consideration that for decomposition and reconstruction of even-length signals no extrapolations at the boundaries is used. In case of odd-length signals, a symmetric continuation of the signal boundary with the last point replica is applied.

When it is necessary to have a continuous set of output blocks, all the input blocks are to be of even length, besides the last one (which can be either of odd or even length). Thus, if the whole amount of elements is odd, only the last block can be of odd length.

ippsWTHaarFwd. The function ippsWTHaarFwd performs the forward single-level discrete Haar transform of a *lenSrc* -length signal *pSrc* and stores the decomposed coarse “low-frequency” components in *pDstLow*, and the detail “high-frequency” components in *pDstHigh*.

ippsWTHaarInv. The function ippsWTHaarInv performs the inverse single-level discrete Haar transform of the coarse “low-frequency” components *pSrcLow* and detail “high-frequency” components *pSrcHigh*, and stores the reconstructed signal in the *lenDst* -length vector *pDst*.

Example 7-10 illustrates the use of the function ippsWTHaarFwd_32f.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 4 for the function ippsWinBlackmanOpt and less than 3 for all other functions of the family.

Example 7-10 Using the ippsWTHaarFwd Function

```
IppStatus wthaar(void) {
      Ipp32f x[8], lo[4], hi[4];
      IppStatus status;
      ippsSet_32f(7, x, 8);  --x[4];
      status = ippsWTHaarFwd_32f(x, 8, lo, hi);
      printf_32f("WT Haar low  =", lo, 4, status);
      printf_32f("WT Haar high =", hi, 4, status);
      return status;
}
Output:
    WT Haar low  =  7.000000 7.000000 6.500000 7.000000
    WT Haar high =  0.000000 0.000000 0.500000 0.000000
```

Related Topics

For more information on wavelet transforms, see: [Str96], pp. 153-157, *Wavelet and Filter Banks*, Wellesley-Cambridge Press; and [Bri94], *Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks*, Los Alamos Report LA-UR-94-1747, 1994. For more information on these references, see the Bibliography at the end of this manual.

Transforms for User Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for user filter banks.

WTFwdInitAlloc, WTInvInitAlloc

Initializes the wavelet transform structure.

```
IppStatus ippsWTFwdInitAlloc_32f(IppsWTFwdState_32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_8s32f(IppsWTFwdState_8s32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_8u32f(IppsWTFwdState_8u32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_16s32f(IppsWTFwdState_16s32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_16u32f(IppsWTFwdState_16u32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);


IppStatus ippsWTInvInitAlloc_32f(IppsWTInvState_32f** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f8s(IppsWTInvState_32f8s** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f8u(IppsWTInvState_32f8u** pState,
      const Ipp32f* pTapsLow, int lenLow, int offsLow,
      const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInitAlloc_32f16s(IppsWTInvState_32f16s** pState,
     const Ipp32f* pTapsLow, int lenLow, int offsLow,
     const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f16u(IppsWTInvState_32f16u** pState,
     const Ipp32f* pTapsLow, int lenLow, int offsLow,
     const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

Arguments

pState	Pointer to the location to store the allocated and initialized state structure.
pTapsLow	Pointer to the vector of lowpass filter taps.
lenLow	Number of taps in the lowpass filter.
offsLow	Additional delay (offset) of the lowpass filter.
pTapsHigh	Pointer to the vector of highpass filter taps.
lenHigh	Number of taps in the highpass filter.
offsHigh	Additional delay (offset) of the highpass filter.

Discussion

The functions ippsWTFwdInitAlloc and ippsWTInvInitAlloc are declared in the ipps.h file. These functions create and initialize the WT state structure *pState* with the following parameters: the lowpass and highpass filter taps *pTapsLow* and *pTapsHigh*, lengths *lenLow* and *lenHigh*, input additional delays *offsLow* and *offsHigh*, respectively.

ippsWTFwdInitAlloc. The function ippsWTFwdInitAlloc initializes the forward WT state structure.

ippsWTInvInitAlloc. The function ippsWTInvInitAlloc initializes the inverse WT state structure.

Application Notes

These functions initialize the wavelet transform state structure, allocate memory for the state structure, initialize it, and returns the *pState* pointer to the state structure. The initialization procedures are implemented separately for forward and inverse

transforms. To perform both forward and inverse wavelet transforms, create two separate state structures. In general, the meanings of initialization parameters of forward and inverse transforms are similar. Each function has parameters describing of a pair of filters. The forward transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of analysis filters. The inverse transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of synthesis filters. Besides lengths and sets of taps the functions allow to specify an additional delay *offsLow* and *offsHigh* for each filter. The adjustable values of delays allow to synchronize:

- group delays for highpass and lowpass filters;
- delays between data of different levels in multilevel decomposition and reconstruction algorithms.

For more information about using these parameters, see "WTFwd" on page 7-74 and "WTInv" on page 7-80. The minimum allowed value of the additional delay for the forward transform is -1. For the inverse transform the delay values must be greater or equal to 0. See "WTFwd" on page 7-74 and "WTInv" on page 7-80 for an example showing how to choose additional delay values. The initialization functions copy filter taps into the state structure *pState*. So all the memory referred to with the pointers can be freed or modified after the functions finished operating. In case of the memory shortage, the function sets a zero pointer to the structure.

Boundaries extrapolation. Typically, reversible wavelet transforms of a bounded signal require data extrapolation towards one or both sides. All internal delay lines are set to zero at the stage of initialization. To set a non-zero signal prehistory, call the function `ippsWTFwdSetDlyLine` on page 7-78. When processed an entire limited data set, data extrapolation may be performed both towards the start and the end of the data vector. For that, the source data and their initial extrapolation are used to form the delay line, the rest of the signal is subdivided into the main block and the signal end. The signal end data and their extrapolation are used to form the last block.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pState*, *pTapsHigh*, or *pTapsLow* pointer is `NULL`.

ippStsSizeErr	Indicates an error when *lenLow* or *lenHigh* is less than or equal to 0.
ippStsWtOffsetErr	Indicates an error when the filter delay *offsLow* or *offsHigh* is less than -1 for the forward transform; and is less than 0 for the inverse transform.

WTFwdFree, WTInvFree

Closes a wavelet transform structure.

```
IppStatus ippsWTFwdFree_32f(IppsWTFwdState_32f* pState);
IppStatus ippsWTFwdFree_8s32f(IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwdFree_8u32f(IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwdFree_16s32f(IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwdFree_16u32f(IppsWTFwdState_16u32f* pState);
IppStatus ippsWTInvFree_32f(IppsWTInvState_32f* pState);
IppStatus ippsWTInvFree_32f8s(IppsWTInvState_32f8s* pState);
IppStatus ippsWTInvFree_32f8u(IppsWTInvState_32f8u* pState);
IppStatus ippsWTInvFree_32f16s (IppsWTInvState_32f16s* pState);
IppStatus ippsWTInvFree_32f16u(IppsWTInvState_32f16u* pState);
```

Arguments

pState	Pointer to the state structure to be closed.

Discussion

The functions ippsWTFwdFree and ippsWTInvFree are declared in the ipps.h file. These functions close the WT state structure *pState* by freeing all the internal memory associated with the state created by ippsWTFwdInitAlloc or ippsWTInvInitAlloc. Call either ippsWTFwdFree or ippsWTInvFree after the transform is completed. If the *pState* pointer is NULL, the function performs no operation and returns the ippStsNullPtrErr status.

ippsWTFwdFree. The function ippsWTFwdFree closes the forward WT state structure.

ippsWTInvFree. The function ippsWTInvFree closes the inverse WT state structure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pState* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

WTFwd

Computes the forward wavelet transform.

```
IppStatus ippsWTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_32f* pState);
IppStatus ippsWTFwd_8s32f(const Ipp8s* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwd_8u32f(const Ipp8u* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwd_16s32f(const Ipp16s* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwd_16u32f(const Ipp16u* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_16u32f* pState);
```

Arguments

pSrc	Pointer to the vector which holds the input signal for decomposition.
pDstLow	Pointer to the vector which holds output coarse “low frequency” components.
pDstHigh	Pointer to the vector which holds output detail “high frequency” components.

dstLen	Number of elements in the vectors *pDstHigh* and *pDstLow*.
pState	Pointer to the state structure.

Discussion

The function ippsWTFwd is declared in the ipps.h file. This function computes the forward wavelet transform. The function transforms the (2 * *dstLen*) -length source data block *pSrc* into "low frequency" components *pDstLow* and "high frequency" components *pDstLow*. The transform parameters are specified in the state structure *pState*.

Application Notes

These functions perform the one-level forward discrete multi-scale transform. An equivalent transform diagram is shown in Figure 7-4. The input signal is divided into the "low frequency" and "high frequency" components. The transfer characteristics of filters are defined by the coefficients set at the initialization stage. The functions are designed for the block processing of data; the transform state structure *pState* contains all needed filter delay lines. Besides these main delay lines each function has an additional delay line for each filter. Adjustable extra delay lines help synchronize group delay times of both highpass and lowpass filters. Moreover, in multilevel systems of signal decomposition delays between different decomposition levels may also be synchronized.

Input and output data block lengths. The functions are designed to decompose signal blocks of even length, therefore, these functions have one parameter only, i.e. the length of input components. The length of the input block must be double the size of each component.

Filter group delays synchronization. Some applications may require synchronization of highpass and lowpass filter time responses. A typical example of this synchronization is synchronizing symmetrical filters of different length.

Below follows an example of biorthogonal set of spline filters of respective length of 6 and 2:

```
static const float decLow[6] =
{
    -6.25000000e-002f,
```

```
      6.25000000e-002f,
      5.00000000e-001f,
      5.00000000e-001f,
      6.25000000e-002f,
      -6.25000000e-002f
};

static const float decHigh[2] =
{
     -5.00000000e-001f,
      5.00000000e-001f
};
```

In this case the lowpass filter gives a delay two samples longer than the highpass filter, which is exactly what the difference between additional initialization function delays should be. The following values must be selected to ensure minimum common signal delay, *offsLow* = -1, *offsHigh* = -1 + 2 = 1. In this case the group times of filter delays are balanced by additional delays. The total delay time is equal to the lowpass filter group delay which has the value of two samples in the decomposition stage in the original signal time frame.

NOTE. *Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, forward transform additional delays must be uniformly even for faultless signal reconstruction.*

Multilevel decomposition algorithm. The implementation of multilevel decomposition algorithms may require synchronization of signal delays across components of different levels.

This is illustrated in the example of the three-level decomposition shown in Figure 7-2. Assume that for transformation the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. Since group delay definitely needs to be synchronized, for the last level select additional filter delays *offsLow3* = -1, *offsHigh3* = 1. Total delay at the last stage of decomposition for this set of filters is two samples. This value corresponds to the time scale of the input of the last stage of decomposition. In order to

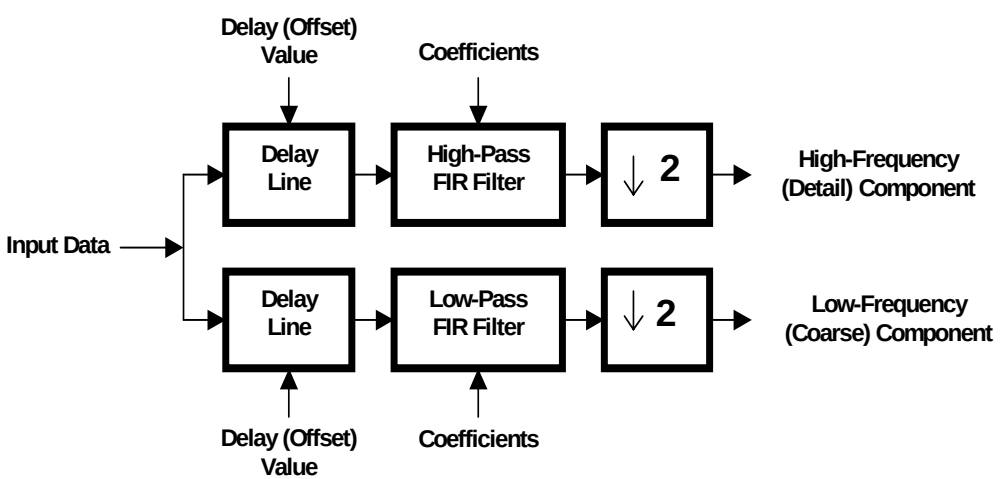
ensure an equivalent delay of the “detail” part on the second level, the delay must be increased by 2*2 samples. Respective values of additional delays for the second level is equal to *offsLow2* = -1, *offsHigh2* = *offsHigh3* + 4 = 5. A greater value of the “high frequency” component delay needs to be selected for the first level of decomposition, *offsLow1* = -1, *offsHigh1* = *offsHigh2* + 2*4 = 13.

Total delay for three levels of decomposition is equal to 12 samples.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data blocks are NULL.
ippStsContextMatchErr	Indicates an error when the state identifier *pState* is incorrect.
ippStsSizeErr	Indicates an error when *dstLen* or *srcLen* is less than or equal to 0.

Figure 7-4 One Level Forward Wavelet Transform



WTFwdSetDlyLine, WTFwdGetDlyLine

Sets and gets the delay lines of the forward wavelet transform.

```
IppStatus ippsWTFwdSetDlyLine_32f(IppsWTFwdState_32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8s32f(IppsWTFwdState_8s32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8u32f(IppsWTFwdState_8u32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_16s32f(IppsWTFwdState_16s32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_16u32f(IppsWTFwdState_16u32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTFwdGetDlyLine_32f(IppsWTFwdState_32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_8s32f(IppsWTFwdState_8s32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_8u32f(IppsWTFwdState_8u32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_16s32f(IppsWTFwdState_16s32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_16u32f(IppsWTFwdState_16u32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
```

Arguments

pState	Pointer to the state structure.
pDlyLow	Pointer to the vector which holds the delay lines for “low frequency” components.

pDlyHigh	Pointer to the vector which holds the delay lines for “high frequency” components.

Discussion

The functions ippsWTFwdSetDlyLine and ippsWTFwdSetDlyLine are declared in the ipps.h file. These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

ippsWTFwdSetDlyLine. The functions ippsWTFwdSetDlyLine sets the delay line values of the forward WT state.

ippsWTFwdGetDlyLine. The functions ippsWTFwdSetDlyLine sets the delay line values of the inverse WT state.

Application Notes

These functions are designed to shape the signal prehistory, save and reconstruct delay lines. Delay lines are implemented separately for highpass and lowpass filters, which gives the option of getting independent signal prehistories for each filter.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the initial signal fed into the forward transform functions, i.e., delay line vectors must be made up of a succession of the signal prehistory counts in the same time frame as the initial signal.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter:

$dlyLowLen = lenLow + offsLow - 1$,

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter:

dlyHighLen = *lenHigh* + *offsHigh* - 1,

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the "high frequency" component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are also described in the section on the transform initialization, see "WTFwdInitAlloc, WTInvInitAlloc" in page 7-70.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDlyLow* or *pDlyHigh* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

WTInv

Computes the inverse wavelet transform.

```
IppStatus ippsWTInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
      int srcLen, Ipp32f* pDst, IppsWTInvState_32f* pState);
IppStatus ippsWTInv_32f8s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
      int srcLen, Ipp8s* pDst, IppsWTInvState_32f8s* pState);
IppStatus ippsWTInv_32f8u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
      int srcLen, Ipp8u* pDst, IppsWTInvState_32f8u* pState);
IppStatus ippsWTInv_32f16s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
      int srcLen, Ipp16s* pDst, IppsWTInvState_32f16s* pState);
IppStatus ippsWTInv_32f16u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
      int srcLen, Ipp16u* pDst, IppsWTInvState_32f16u* pState);
```

Arguments

pSrcLow	Pointer to the vector which holds input coarse "low frequency" components.

pSrcHigh	Pointer to the vector which holds detail “high frequency” components.
srcLen	Number of elements in the vectors *pSrcHigh* and *pSrcLow*.
pDst	Pointer to the vector which holds the output reconstructed signal.
pState	Pointer to the state structure.

Discussion

The function ippsDCTInv is declared in the ipps.h file. This function computes the inverse wavelet transform. The function transforms the “low frequency” components *pSrcLow* and “high frequency” components *pSrcHigh* into the (2 * *srcLen*)-length destination data block *pDst*. The transform parameters are specified in the state structure *pState*.

Application Notes

These functions are used for one level of inverse multiscale transformation which results in reconstructing the original signal from the two “low frequency” and “high frequency” components. Figure 7-5 shows an equivalent transform algorithm. Two interpolation filters are used for signal reconstruction; their coefficients are set at the initialization stage. The inverse transform implementation, similar to forward transform implementation, contains additional delay lines needed to synchronize the group time of filter delays and delays across different levels of data reconstruction.

Input and output data block lengths. These functions are designed to reconstruct the blocks of the even length signal. The signal component length must be the input data. The length of the output block of the reconstructed signal must be double the length of each of the components.

Filter group delay synchronization. In this example consider a biorthogonal set of spline filters of length 2 and 6:

```
static const float recLow[2] =
{
     1.00000000e+000f,
     1.00000000e+000f
};
```

```
static const float recHigh[6] =
{
    -1.25000000e-001f,
    -1.25000000e-001f,
     1.00000000e+000f,
    -1.00000000e+000f,
     1.25000000e-001f,
     1.25000000e-001f
};
```

This set of filters corresponds to the set of filters considered in a similar section of the forward transform, see "WTFwd" on page 7-74.

Unlike the case described above, this time the highpass filter generates a delay greater by two samples compared against the low frequency filter. The two sample difference should also exist between initialization function additional delays. The following parameters of additional delays need to be selected in order to ensure the minimum total delay, *offsLow* = 2, *offsHigh* = 0. In this case the total delay is equal to the highpass filter group delay, which at the decomposition stage is equal to two samples in the original signal time frame.

Total delay of one level of decomposition and reconstruction is equal to 4 samples, considering the decomposition stage delay.

NOTE. *Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, inverse transform additional delays must be uniformly even and opposite to the evenness of the decomposition delays for faultless signal reconstruction.*

Multilevel reconstruction algorithms. An example of a three-level signal reconstruction algorithm is shown in Figure 7-3. The scheme corresponds to the decomposition scheme described in the section on the forward transform, see "WTFwd" on page 7-74. Therefore, for the inverse transform the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. The lowest level filter delays are set to *offsLow3* = 2, *offsHigh3* = 0. The total delay at this stage of reconstruction is equal to two samples. In order to ensure an equivalent delay of the

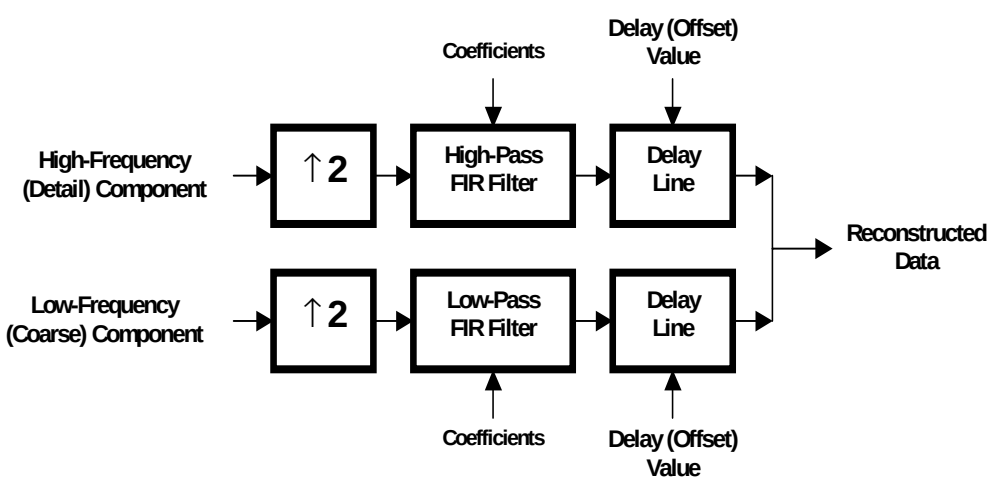
"detail" part in the middle level, the delay must be increased. Respective values of additional delays for the second level are equal to *offsLow2* = 2, *offsHigh2* = *offsHigh3* + 2*2 = 4. A greater value of high frequency component delay needs to be selected for the last level of reconstruction, *offsLow1* = -1, *offsHigh1* = *offsHigh2* + 2*4 = 12.

The total delay for three levels of reconstruction is equal to 12 samples. The total delay of the three-level decomposition and reconstruction cycle is equal to 24 samples.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data blocks are NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.
ippStsSizeErr	Indicates an error when *dstLen* or *srcLen* is less than or equal to 0.

Figure 7-5 One Level Inverse Wavelet Transform



WTInvSetDlyLine, WTInvGetDlyLine

Sets and gets the delay lines of the inverse wavelet transform.

```
IppStatus ippsWTInvSetDlyLine_32f(IppsWTInvState_32f* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8s(IppsWTInvState_32f8s* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8u(IppsWTInvState_32f8u* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f16s(IppsWTInvState_32f16s* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f16u(IppsWTInvState_32f16u* pState,
     const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f(IppsWTInvState_32f* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f8s(IppsWTInvState_32f8s* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f8u(IppsWTInvState_32f8u* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f16s(IppsWTInvState_32f16s* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f16u(IppsWTInvState_32f16u* pState,
     Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
```

Arguments

pState	Pointer to the state structure.
pDlyLow	Pointer to the vector which holds delay lines for “low frequency” components.

pDlyHigh	Pointer to the vector which holds delay lines for “high frequency” components.

Discussion

The functions ippsWTFwdSetDlyLine and ippsWTFwdSetDlyLine are declared in the ipps.h file. These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

ippsWTFwdSetDlyLine. The functions ippsWTFwdSetDlyLine sets the delay line values of the forward WT state.

ippsWTInvSetDlyLine. The functions ippsWTFwdSetDlyLine sets the delay line values of the inverse WT state.

Application Notes

These functions set and read delay lines of inverse multiscale transformation. The functions receive or return filter low and high frequency component delay line vectors. The functions may be used to shape previous history of each of the components. Installation functions and read functions together ensure that delay lines from each filter are saved and reconstructed.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the low and high frequency components at the input of the inverse transform functions. Thus, delay line vectors must be made up of a succession of signal prehistory counts in the same time frame as the input components.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter in terms of the C language (integer division by two is used here for simplicity):

dlyLowLen = (*lenLow* + *offsLow* - 1) / 2,

where *lenLow* and *offsLow* are respectively the length and additional delay of the "low frequency" component filter.

The following expression defines the length of the delay line vector of the "high frequency" component filter in terms of the C language:

dlyHighLen = (*lenHigh* + *offsHigh* - 1) / 2,

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the "high frequency" component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are also described in the section on the transform initialization, see "WTFwdInitAlloc, WTInvInitAlloc" on page 7-70.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDlyLow* or *pDlyHigh* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

Speech Recognition Functions

8

This chapter describes Intel® IPP functions that are designed to be used in speech recognition applications.

The full list of functions in this group is given in Table 8-1.

Table 8-1 Intel IPP Speech Recognition Functions

Function Base Name	Operation
Basic Arithmetics Functions	
AddAllRowSum	Calculates the sums of column vectors in a matrix and adds the sums to a vector.
SumColumn	Calculates sums of column vectors in a matrix.
SumRow	Calculates sums of row vectors in a matrix.
SubRow	Subtracts a vector from all matrix rows.
CopyColumn_Indirect	Copies the input matrix with columns redirection.
BlockDMatrixInitAlloc	Initializes the structure that represents a symmetric block diagonal matrix.
BlockDMatrixFree	Deallocates the block diagonal matrix structure.
NthMaxElement	Searches for the N-th maximal element of a vector.
VecMatMul	Multiplies a vector by a matrix.
MatVecMul	Multiplies a matrix by a vector
Feature Processing Functions	
ZeroMean	Subtracts the mean value from the input vector.
CompensateOffset	Removes the DC offset of the input signals.
SignChangeRate	Counts the zero-cross rate for the input signal.
LinearPrediction	Performs linear prediction analysis on the input vector.
Durbin	Performs Durbin's recursion on an input vector of autocorrelations.
Schur	Calculates reflection coefficients using Schur algorithm.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
LPToSpectrum	Calculates smoothed magnitude spectrum.
LPToCepstrum	Calculates cepstrum coefficients from linear prediction coefficients.
CepstrumToLP	Calculates linear prediction coefficients from cepstrum coefficients.
LPToReflection	Calculates the linear prediction reflection coefficients from the linear prediction coefficients.
ReflectionToLP	Calculates the linear prediction coefficients from the linear prediction reflection coefficients.
ReflectionToAR	Converts reflection coefficients to area ratios.
ReflectionToTilt	Calculates tilt for rise/fall/connection parameters.
PitchmarkToF0	Calculates rise and fall amplitude and duration for tilt.
UnitCurve	Calculates tilt for rise and fall coefficicents.
LPToLSP	Calculates line spectrum pairs vector from linear prediction coefficients.
LSPToLP	Converts line spectrum pairs vector to linear prediction coefficients.
MelToLinear	Converts Mel-scaled values to linear scale values.
LinearToMel	Converts linear-scale values to Mel-scale values.
CopyWithPadding	Copies the input signal to the output with zero-padding.
MelFBankGetSize	Gets the size of the Mel-frequency filter bank structure.
MelFBankInit	Initializes the structure for performing the Mel-frequency filter bank analysis.
MelFBankInitAlloc	Initializes the structure and allocates memory for performing the Mel-frequency filter bank analysis.
MelLinFBankInitAlloc	Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.
EmptyFBankInitAlloc	Initializes an empty filter bank structure.
FBankFree	Destroys the structure for the filter bank analysis.
FBankGetCenters	Retrieves the center frequencies of the triangular filter banks.
FBankSetCenters	Sets the center frequencies of the triangular filter banks.
FBankGetCoeffs	Retrieves the filter bank weight coefficients.
FBankSetCoeffs	Sets the filter bank weight coefficients.
EvalFBank	Performs the filter bank analysis.
DCTLifterGetSize_MulC0	Gets the size of the DCT structure.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
DCTLifterInit_MulC0	Initializes the structure to perform DCT and lift the DCT coefficients.
DCTLifterInitAlloc	Initializes the structure and allocates memory to perform DCT and lift the DCT coefficients.
DCTLifterFree	Destroys the structure used for the DCT and liftering.
DCTLifter	Performs the DCT and lifts the DCT coefficients.
NormEnergy	Normalizes a vector of energy values.
SumMeanVar	Calculates both the sum of a the vector and its square sum.
NewVar	Calculates the variances given the sum and square sum accumulators.
RecSqrt	Calculates square roots of a vector and their reciprocals.
AccCovarianceMatrix	Accumulates covariance matrix.
Derivative Functions	
CopyColumn	Copies the input sequence into the output sequence.
EvalDelta	Calculates the derivatives of feature vectors.
Delta	Copies the base features and calculates the derivatives of feature vectors.
DeltaDelta	Copies the base features and calculates their first and second derivatives.
Pitch Super Resolution Functions	
CrossCorrCoeffDecim	Calculates vector of cross correlation coefficients with decimation.
CrossCorrCoeff	Calculates the cross correlation coefficient.
CrossCorrCoeffInterpolation	Calculates interpolated cross correlation coefficient.
Model Evaluation Functions	
AddNRows	Adds N vectors from a vector array.
ScaleLM	Scales vector elements with thresholding.
LogAdd	Adds two vectors in the logarithmic representation.
LogSub	Subtracts a vector from another vector, in the logarithmic representation.
LogSum	Sums vector elements in the logarithmic representation.
MahDistSingle	Calculates the Mahalanobis distance for a single observation vector.
MahDist	Calculates the Mahalanobis distances for multiple observation vectors.
MahDistMultiMix	Calculates the Mahalanobis distances for multiple means and variances.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
LogGaussSingle	Calculates the observation probability for a single Gaussian with an observation vector.
LogGauss	Calculates the observation probability for a single Gaussian with multiple observation vectors.
LogGaussMultiMix	Calculates the observation probability for multiple Gaussian mixture components.
LogGaussMax	Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.
LogGaussMaxMultiMix	Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.
LogGaussAdd	Calculates the likelihood probability for multiple observation vectors.
LogGaussAddMultiMix	Calculates the likelihood probability for multiple Gaussian mixture components.
LogGaussMixture	Calculates the likelihood probability for the Gaussian mixture.
LogGaussMixtureSelect	Calculates the likelihood probability for the Gaussian mixture using Gaussian selection.
BuildSignTable	Fills sign table for Gaussian mixture calculation.
FillShortlist_Row	Fills row-wise shortlist table for Gaussian selection.
FillShortlist_Column	Fills column-wise shortlist table for Gaussian selection.
DTW	Computes the distance between observation and reference vector sequences using Dynamic Time Warping algorithm.
Model Estimation Functions	
MeanColumn	Computes the mean values for the column elements.
VarColumn	Calculates the variances for the column elements.
MeanVarColumn	Calculates the means and variances for the column elements of a matrix.
WeightedMeanColumn	Computes the weighted mean values for the column elements.
WeightedVarColumn	Computes the weighted variance values for the column elements.
WeightedMeanVarColumn	Computes weighted mean and variance values for the column elements.
NormalizeColumn	Normalizes the matrix columns given the column means and variances.
NormalizeInRange	Normalizes and scales input vector elements.
MeanVarAcc	Accumulates the estimates for the mean and variance re-estimation.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
GaussianDist	Calculates the distance between two Gaussians.
GaussianSplit	Splits a single Gaussian component into two with the same variance.
GaussianMerge	Merges two Gaussian probability distribution functions.
Entropy	Calculates entropy of the input vector.
SinC	Calculates sine divided by its argument.
ExpNegSqr	Calculates exponential of the squared argument taken with the inverted sign.
BhatDist	Calculates the Bhattacharia distance between two Gaussians.
UpdateMean	Updates the mean vector in the EM training algorithm.
UpdateVar	Updates the variance vector in the EM training algorithm.
UpdateWeight	Updates the weight values of Gaussian mixtures in the EM training algorithm.
UpdateGConst	Updates the fixed constant in the Gaussian output probability density function.
OutProbPreCalc	Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.
DcsClustLAccumulate	Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.
DcsClustLCompute	Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.
Model Adaptation Functions	
AddMulColumn	Adds a weighted matrix column to the other column.
AddMulRow	Adds a weighted vector to the other vector.
QRTransColumn	Performs the QR transformation.
DotProdColumn	Calculates the dot product of two matrix columns.
MulColumn	Multiplies a matrix column by a value.
SumColumnAbs	Calculates the absolute sum of matrix column elements.
SumColumnSqr	Calculates the square sums of weighted matrix column elements.
SumRowAbs	Calculates the absolute sum of the vector elements.
SumRowSqr	Calculates the square sum of weighted vector elements.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
SVD	Performs Single Value Decomposition on a matrix.
WeightedSum	Calculates the weighted sums of two input vector elements.
Vector Quantization Functions	
FormVector	Constructs an output vector of multiple streams from codebook entries.
CdbkGetSize	Calculates the size in bytes of the codebook.
CdbkInit	Initializes the structure that contains the codebook.
CdbkInitAlloc	Initializes the codebook structure.
CdbkFree	Destroys the codebook structure.
GetCdbkSize	Retrieves the number of codevectors in the codebook.
GetCodebook	Retrieves the codevectors from the codebook.
VQ	Quantizes the input vectors given a codebook.
SplitVQ	Quantizes a multiple-stream vector given the codebooks.
VQSingle_Sort, VQSingle_Thresh	Quantizes the input vector given a codebook and gets several closest clusters.
FormVectorVQ	Constructs multiple-stream vectors from codebooks, given indexes.
Polyphase Resampling Functions	
ResamplePolyphaseInitAlloc	Initializes the structure for polyphase data resampling.
ResamplePolyphaseFree	Free structure for polyphase data resampling.
ResamplePolyphase	Resamples input data using polyphase filters.
Advanced Aurora Functions	
SmoothedPowerSpectrum_Aurora	Calculates smoothed magnitude of the FFT output.
NoiseSpectrumUpdate_Aurora	Updates the noise spectrum.
WienerFilterDesign_Aurora	Calculates an improved transfer function of the adaptive Wiener filter.
MelFBankInitAlloc_Aurora	Initializes the structure for performing the Mel-frequency filter bank analysis.
TabsCalculation_Aurora	Calculates filter coefficients for residual filter.
ResidualFilter_Aurora	Calculates a denoised waveform signal.
WaveProcessing_Aurora	Processes waveform data after noise reduction.
LowHighFilter_Aurora	Calculates low band and high band filters.

Table 8-1 Intel IPP Speech Recognition Functions (continued)

Function Base Name	Operation
HighBandCoding_Aurora	Codes and decodes the high frequency band energy values.
BlindEqualization_Aurora	Equalizes the cepstral coefficients.
DeltaDelta_Aurora	Calculates the first and second derivatives according to ETSI ES 202 050 standard.
VADGetBufSize_Aurora	Queries the memory size for VAD decision.
VADInit_Aurora	Gets the VAD structure size.
VADDecision_Aurora	Takes the VAD decision.
VADFlush_Aurora	Takes VAD decision for zero input frame.
Ephraim-Malah Noise Suppressor Functions	
FilterUpdateEMNS	Calculates the noise suppression filter coefficients.
FilterUpdateWiener	Calculates the Wiener filter coefficients.
GetSizeMCRA	Calculates the size in bytes required for the state structure.
InitMCRA	Initializes the IppMCRAState state structure.
InitAllocMCRA	Allocates memory and initializes the IppMCRAState state structure.
UpdateNoisePSDMCRA	Re-estimates the noise power spectrum.
Acoustic Echo Canceller Functions	
FilterAECNLMS	Computes the frequency-domain adaptive filter output.
CoefUpdateAECNLMS	Updates the adaptive filter coefficients.
StepSizeUpdateAECNLMS	Computes the adaptive step size.
ControllerGetSizeAEC	Returns the size of the AEC controller state structure.
ControllerInitAEC	Initializes the AEC controller state structure.
ControllerUpdateAEC	Implements the energy-based AEC controller.
Voice Activity Detector Functions	
FindPeaks	Identifies peaks in the input vector.
PeriodicityLSPE	Computes the periodicity of the input speech frame.
Periodicity	Computes the periodicity of the input block.

NOTE. *In the below descriptions of function arguments, when an argument refers to a vector or an array, the expression in square brackets given after the explanation of the argument specifies the dimension of that vector or array.*
In this chapter, unlike other chapters of this manual, arguments that represent step values are measured in elements of the corresponding array, unless explicitly stated to the contrary.

Basic Arithmetics

The functions included in this section perform generic arithmetic operations on vectors and matrices.

AddAllRowSum

Calculates the sums of column vectors in a matrix and adds the sums to a vector.

```
IppStatus ippsAddAllRowSum_32f_D2(const Ipp32f* pSrc, int step,
   int height, Ipp32f* pSrcDst, int width);
IppStatus ippsAddAllRowSum_32f_D2L(const Ipp32f** mSrc, int height,
   Ipp32f* pSrcDst, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in *pSrc*.
height	Number of rows in the input matrix *mSrc*.

pSrcDst	Pointer to the output vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the output vector *pSrcDst*.

Discussion

The function ippsAddAllRowSum is declared in the ippsr.h file. This function calculates sums of column vectors in the input matrix, and adds these sums to the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} pSrc[i \cdot step + j],\ 0 \le j < width .$$

For functions with the D2L suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} mSrc[i][j],\ 0 \le j < width .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

SumColumn

Calculates sums of column vectors in a matrix.

```
IppStatus ippsSumColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int step, int height,
   Ipp32s* pDst, int width, int scaleFactor);
```

```
IppStatus ippsSumColumn_16s32f_D2(const Ipp16s* pSrc, int step, int height,
   Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2(const Ipp32f* pSrc, int step, int height,
   Ipp32f* pDst, int width);
IppStatus ippsSumColumn_64f_D2(const Ipp64f* pSrc, int step, int height,
   Ipp64f* pDst, int width);
IppStatus ippsSumColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
   Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumColumn_16s32f_D2L(const Ipp16s** mSrc, int height,
   Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f* pDst,
   int width);
IppStatus ippsSumColumn_64f_D2L(const Ipp64f** mSrc, int height, Ipp64f* pDst,
   int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
height	Number of rows in the input matrix *mSrc*.
pDst	Pointer to the output vector [*width*].
width	Number of columns in the input matrix *mSrc* and the length of the output vector *pDst*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsSumColumn is declared in the ippsr.h file. This function calculates sums of column vectors in the input matrix, and stores these sums in the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i \cdot step + j],\ 0 \le j < width\ .$$

For functions with the D2L suffix,

$$pDst[j] = \sum_{i=0}^{height-1} mSrc[i][j], \quad 0 \le j < width .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

SumRow

Calculates sums of row vectors in a matrix.

```
IppStatus ippsSumRow_16s32s_D2Sfs(const Ipp16s * pSrc, int height, int step,
   Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2(const Ipp16s* pSrc, int height, int step,
   Ipp32f* pDst, int width);
IppStatus ippsSumRow_32f_D2(const Ipp32f* pSrc, int height, int step,
   Ipp32f* pDst, int width);
IppStatus ippsSumRow_64f_D2(const Ipp64f* pSrc, int height, int step,
   Ipp64f* pDst, int width);
IppStatus ippsSumRow_16s32s_D2LSfs(const Ipp16s** mSrc, int height, Ipp32s*
   pDst, int width, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2L(const Ipp16s** mSrc, int height, Ipp32f* pDst,
   int width);
IppStatus ippsSumRow_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f* pDst,
   int width);
IppStatus ippsSumRow_64f_D2L(const Ipp64f** mSrc, int height, Ipp64f* pDst,
   int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*, and also the length of the output vector *pDst*.
step	Row step in the input vector *pSrc*.
pDst	Pointer to the output vector [*height*].
width	Number of columns in the input matrix *mSrc*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsSumRow is declared in the ippsr.h file. This function calculates sums of row vectors in the input matrix *mSrc* and stores these sums in the output vector *pDst*. The operations are as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[i \cdot step + j], \quad 0 \le i < height .$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mSrc[i][j], \quad 0 \le i < height .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

SubRow

Subtracts a vector from all matrix rows.

```
IppStatus ippsSubRow_16s_D2(const Ipp16s* pSrc, int width, Ipp16s* pSrcDst,
   int dstStep, int height);
IppStatus ippsSubRow_32f_D2(const Ipp32f* pSrc, int width, Ipp32f* pSrcDst,
   int dstStep, int height);
IppStatus ippsSubRow_16s_D2L(const Ipp16s* pSrc, Ipp16s** mSrcDst, int width,
   int height);
IppStatus ippsSubRow_32f_D2L(const Ipp32f* pSrc, Ipp32f** mSrcDst, int width,
   int height);
```

Arguments

pSrc	Pointer to the input vector [*width*].
pSrcDst	Pointer to the source and destination vector [*height*dstStep*].
mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
dstStep	Row step in the vector *pSrcDst*.
height	Number of rows in the matrix *mSrcDst*.

Discussion

The function ippsSubRow is declared in the ippsr.h file. This function subtracts the input vector *pSrc* from all matrix rows. The operations are as follows:

For functions with the D2 suffix,

$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] - pSrc[j]$,

$0 \le i < height$, $0 \le j < width$.

For functions with the D2L suffix,

$mSrcDst[i][j] = mSrcDst[i][j] - pSrc[j]$,

$0 \le i < height$, $0 \le j < width$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pSrcDst*, or *mSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *dstStep* is less than *width*.

CopyColumn_Indirect

Copies the input matrix with columns redirection.

```
IppStatus ippsCopyColumn_Indirect_16s_D2(const Ipp16s* pSrc, int srcLen,
   int srcStep, Ipp16s* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
   height);
IppStatus ippsCopyColumn_Indirect_32f_D2(const Ipp32f* pSrc, int srcLen,
   int srcStep, Ipp32f* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
   height);
IppStatus ippsCopyColumn_Indirect_16s_D2L(const Ipp16s** mSrc, int srcLen,
   Ipp16s** mDst, const Ipp32s* pIndx, int dstLen, int height);
IppStatus ippsCopyColumn_Indirect_32f_D2L(const Ipp32f** mSrc, int srcLen,
   Ipp32f** mDst, const Ipp32s* pIndx, int dstLen, int height);
```

Arguments

pSrc	Pointer to the input vector [*height***srcStep*].
mSrc	Pointer to the input matrix [*height*][*srcLen*].
srcLen	Number of columns in the input matrix *mSrc*.
srcStep	Row step in the vector *pSrc*.
pDst	Pointer to the output vector [*height***dstStep*].
mDst	Pointer to the output matrix [*height*][*dstLen*].

pIndx	Pointer to the redirection vector [*dstLen*].
dstLen	Number of columns in the output matrix *mDst*.
dstStep	Row step in the vector *pDst*.
height	Number of rows in both the input and output matrices.

Discussion

The function ippsCopyColumn_Indirect is declared in the ippsr.h file. This function copies the input matrix *mSrc* to the output matrix *mDst* with the columns redirected by *pIndx*. The operations are as follows:

For functions with the D2 suffix,

$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcStep + pIndx[j]],\ 0 \le i < height,\ 0 \le j < dstLen.$

For functions with the D2L suffix,

$mDst[i][j] = mSrc[i][pIndx[j]],\ 0 \le i < height,\ 0 \le j < dstLen\ .$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pDst*, *mDst*, or *pIndx* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *srcLen*, or *dstLen* is less than or equal to 0, or $pIndx[j] \ge srcLen$ or $pIndx[j] < 0$ for $0 \le j < dstLen$.
ippStsStrideErr	Indicates an error when *srcStep* is less than *srcLen* or *dstStep* is less than *dstLen*.

BlockDMatrixInitAlloc

Initializes the structure that represents a symmetric block diagonal matrix.

```
IppStatus ippsBlockDMatrixInitAlloc_16s(IppsBlockDMatrix_16s** pMatrix,
   const Ipp16s** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_32f(IppsBlockDMatrix_32f** pMatrix,
   const Ipp32f** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_64f(IppsBlockDMatrix_64f** pMatrix,
   const Ipp64f** mSrc, const int* bSize, int nBlocks);
```

Arguments

pMatrix	Pointer to the block diagonal matrix to be created.
mSrc	Pointer to the vector of pointers to matrix rows.
bSize	Pointer to vector of block sizes [*nBlocks*].
nBlocks	Number of blocks in the matrix.

Discussion

The function ippsBlocDMatrixInitAlloc is declared in the ippsr.h file. This function creates the structure that contains a symmetric block diagonal matrix. Matrix elements outside the blocks are assumed to be zero. The block diagonal matrix A is defined as follows:

$A[K_l + i][K_l + j] = A[K_l + j][K_l + i] = mSrc[K_l + i]\,[j]$,

for $i, j = 0 \ldots bSize[l] - 1$ and $l = 0 \ldots nBlocks - 1$,

where

$$K_l = \sum_{k<l} bSize[k]$$

and $mSrc[K_l + i]$ points to the non-zero part of the matrix A row.
The size of the matrix is equal to $K_{nBlocks}$ by $K_{nBlocks}$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMatrix* or *mSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *bSize* or *nBlocks* is less than or equal to 0.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

BlockDMatrixFree

Deallocates the block diagonal matrix structure.

```
IppStatus ippsBlockDMatrixFree_16s(IppsBlockDMatrix_16s* pMatrix);
IppStatus ippsBlockDMatrixFree_32f(IppsBlockDMatrix_32f* pMatrix);
IppStatus ippsBlockDMatrixFree_64f(IppsBlockDMatrix_64f* pMatrix);
```

Arguments

pMatrix	Pointer to the block diagonal matrix.

Discussion

The function ippsBlockDMatrixFree is declared in the ippsr.h file. This function destroys the block diagonal matrix structure, and frees all memory associated with it.

Return Value

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

NthMaxElement

Searches for the N-th maximal element of a vector.

```
IppStatus ippsNthMaxElement_32s(const Ipp32s* pSrc, int len, int N,
   Ipp32s* pRes);
IppStatus ippsNthMaxElement_32f(const Ipp32f* pSrc, int len, int N,
   Ipp32f* pRes);
IppStatus ippsNthMaxElement_64f(const Ipp64f* pSrc, int len, int N,
   Ipp64f* pRes);
```

Arguments

pSrc	Pointer to the input vector [*len*].
len	Number of elements in the input vector *pSrc*.
N	Rank of element to find.
pRes	Pointer to the value of *N*-th maximal element.

Discussion

The function ippsNthMaxElement is declared in the ippsr.h file. This function finds the *N*-th maximal element of the input vector *pSrc*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pRes* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsBadArgErr	Indicates an error when *N* is less then 0, or greater than or equal to *len*.

VecMatMul

Multiplies a vector by a matrix.

```
IppStatus ippsVecMatMul_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pMatr, int
   step, int height, Ipp16s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_16s_D2LSfs(const Ipp16s* pSrc, const Ipp16s** mMatr, int
   height, Ipp16s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMatr, int
   step, int height, Ipp32f* pDst, int width);
IppStatus ippsVecMatMul_32f_D2L(const Ipp32f* pSrc, const Ipp32f** mMatr, int
   height, Ipp32f* pDst, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*].
pMatr	Pointer to the input matrix vector [*height*step*].
mMatr	Pointer to the input matrix [*height*][*width*].
step	Row step in *pMatr* vector (in *pMatr* elements).
width	Length of the result vector and input matrix rows.
height	Number of rows in the input matrix.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsVecMatMul is declared in the ippsr.h file. This function multiplies the input row vector by the given matrix as:

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot pMatr[i \cdot step + j]$$

for the function flavors with D2 suffix,

and

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot mMatr[i][j]$$

for the function flavors with `D2L` suffix,

where $0 \le j < width$.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`*, *`pMatr`*, *`mMatr`* or *`pDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`width`* or *`height`* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *`step`* is less than *`width`*.

MatVecMul

Multiplies a matrix by a vector.

```
IppStatus ippsMatVecMul_16s_D2Sfs(const Ipp16s* pMatr, int step, const Ipp16s*
   pSrc, int width, Ipp16s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_16s_D2LSfs(const Ipp16s** mMatr, const Ipp16s* pSrc,
   int width, Ipp16s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_32f_D2(const Ipp32f* pMatr, int step, const Ipp32f*
   pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMatVecMul_32f_D2L(const Ipp32f** mMatr, const Ipp32f* pSrc, int
   width, Ipp32f* pDst, int height);
```

Arguments

`pSrc`	Pointer to the input vector [*`width`*].
`pMatr`	Pointer to the input matrix vector [*`height*step`*].

mMatr	Pointer to the input matrix [*height*][*width*].
step	Row step in *pMatr* vector (in *pMatr* elements).
width	Length of the input vector and input matrix rows.
height	Number of rows in the input matrix and the length of the result vector.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsVecMatMul is declared in the ippsr.h file. This function multiplies the the given matrix by the input column vector as:

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot pMatr[i \cdot step + j]$$

for the function flavors with D2 suffix,

and

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot mMatr[i][j]$$

for the function flavors with D2L suffix,

where $0 \le i < height$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pMatr*, *mMatr* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

Feature Processing

This section describes functions that pre-process raw speech signals. Feature extraction is the first step in the recognition process. Speech features are a compressed form of the original speech signals. By extracting features, the problem dimension is reduced. To some extent, feature extraction normalizes speaker variations and environmental distortions.
This section also includes functions needed to support both silence/speech detection and also some well-known analysis techniques on speech signals.

ZeroMean

Subtracts the mean value from the input vector.

```
IppStatus ippsZeroMean_16s(Ipp16s* pSrcDst, int len);
```

Arguments

pSrcDst	Pointer to the source and destination vector [*len*].
len	The number of elements in the vector.

Discussion

The function ippsZeroMean is declared in the ippsr.h file. This function calculates the mean value of the *pSrcDst* vector and subtracts it from the vector *pSrcDst*. The resulting values are saturated if they exceed the range [-32768..32767] .
The operations are as follows:

$$pSrcDst[i] = \max(-32768, \min(32767, pSrcDst[i] - \frac{1}{len}\sum_{j=0}^{len-1} pSrcDst[j]))$$

for $0 \le i < len$.

Return Value

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

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

CompensateOffset

Removes the DC offset of the input signals.

```
IppStatus ippsCompensateOffset_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   Ipp16s* pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
   Ipp32f* pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffset_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
   pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f_I(Ipp32f* pSrcDst, int len, Ipp32f*
   pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffsetQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
   len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
IppStatus ippsCompensateOffsetQ15_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
   pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
```

Arguments

pSrc	Pointer to the source vector [*len*].
pDst	Pointer to the destination vector [*len*].
pSrcDst	Pointer to the source and destination vector [*len*] for in-place operations.
pSrcDst0	Pointer to the previous source element.
len	Number of elements in the vector.
dst0	Previous destination element.
val	Constant for offset compensation.
valQ15	Real-valued Q15 format constant for offset compensation ($0.0_{Q31} \le valQ15 < 1.0_{Q31}$).
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsCompensateOffset is declared in the ippsr.h file. This function removes the offset of the input signals. The destination vector is calculated as follows.

For the ippsCompensateOffset function:

$pDst[0] = pSrc[0] - pSrcDst[0] + val \cdot dst0$

$pDst[i] = pSrc[i] - pSrc[i-1] + val \cdot pDst[i-1], \ 1 \le i \le len - 1$

$pSrcDst0[0] = pSrc[len - 1],$

and for ippsCompensateOffset_I function:

$y = pSrcDst[0] - pSrcDst0[0] + val \cdot dst0, x = pSrcDst[0], pSrcDst[0] = y,$

$y = pSrcDst[i] - pSrcDst[i-1] + val \cdot x, x = pSrcDst[i], pSrcDst[i] = y,$

$1 \le i \le len - 1,$

$pSrcDst0[0] = x.$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, *pSrcDst*, or *pSrcDst0* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or when *valQ15* is out of range.

SignChangeRate

Counts the zero-cross rate for the input signal.

```
IppStatus ippsSignChangeRate_16s(const Ipp16s* pSrc, int len, Ipp32s* pRes);
IppStatus ippsSignChangeRate_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
IppStatus ippsSignChangeRateCount0_16s(const Ipp16s* pSrc, int len, Ipp32s*
   pRes);
```

```
IppStatus ippsSignChangeRateCount0_32f(const Ipp32f* pSrc, int len, Ipp32f*
   pRes);
```

Arguments

pSrc	Pointer to the input signal [*len*].
len	Number of elements in the input signal *pSrc*.
pRes	Pointer to the result variable.

Discussion

The function ippsSignChangeRate is declared in the ippsr.h file. This function counts the number of sign changes in the input signal.
This function can be used to detect speech in continuous speech input.

The operations are as follows:

For the ippsSignChangeRate function,

$$pRes[0] = \sum_{i=1}^{len-1} \begin{cases} 1, & if\ \ pSrc[i] \cdot pSrc[i-1] < 0 \\ 0, & otherwise \end{cases}$$

For the ippsSignChangeRateCount0 function,

$$pRes[0] = \frac{1}{2} \sum_{i=1}^{len-1} |sign(pSrc[i]) - sign(pSrc[i-1])| \ , \ sign(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

Return Value

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

LinearPrediction

Performs linear prediction analysis on the input vector.

```
IppStatus ippsLinearPrediction_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
   Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPrediction_Auto_32f(const Ipp32f* pSrc, int lenSrc,
   Ipp32f* pDst, int lenDst);
IppStatus ippsLinearPredictionNeg_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
   Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPredictionNeg_Auto_32f(const Ipp32f* pSrc, int lenSrc,
   Ipp32f* pDst, int lenDst);
IppStatus ippsLinearPrediction_Cov_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
   Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPrediction_Cov_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
   pDst, int lenDst);
```

Arguments

pSrc	Pointer to the input vector [*lenSrc*].
lenSrc	Length of the input vector *pSrc*.
pDst	Pointer to the output LPC coefficients vector [*lenDst*].
lenDst	Length of the output vector *pDst*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLinearPrediction is declared in the ippsr.h file. This function performs linear prediction analysis on the input signal.
For functions with the Auto suffix, the LPC coefficients are calculated by solving the following equations that represent the autocorrelation approach:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = r[k], k = 1,\ldots lenDst,$$

where $r[k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k]$.

For functions with the Neg_Auto suffix, the LPC coefficients are calculated by solving the similar equations with inverted sign of the right-hand side:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = -r[k] \text{ , } k = 1 \ldots lenDst \text{ ,}$$

For functions with the Cov suffix, the LPC coefficients are calculated by solving the following equations that represent the covariance approach:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot c[i][k] = c[0][k] \text{ , } k = 1, \ldots lenDst \text{ ,}$$

where $c[i][k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k-i]$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *lenSrc* or *lenDst* is less than or equal to 0, or *lenDst* is greater or equal than *lenSrc*.
ippStsNoOperation	Indicates no solution to the LPC problem.

Durbin

Performs Durbin's recursion on an input vector of autocorrelations.

```
IppStatus ippsDurbin_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   Ipp32f* pErr, int scaleFactor);
IppStatus ippsDurbin_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
   Ipp32f* pErr);
```

Arguments

pSrc	Pointer to the input vector [*len+1*].
pDst	Pointer to the output LPC coefficients vector [*len*].
len	Length of the output vector.
pErr	Pointer to the result prediction error.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsDurbin is declared in the ippsr.h file. This function performs the Durbin's recursion on the input autocorrelation vector and calculates the linear prediction coefficients and the prediction error as follows:

1) Initialization:

$m = len,\ R_j = pSrc[j],\ j = 0,...m,\ E^0 = R_0$

2) Iterations for $i = 1,...m$:

$$k_i = \left(R_i - \sum_{j=1}^{i-1} y_j^{i-1} \cdot R_{i-j} \right) / E^{i-1},\ E^i = (1 - k_i^2) \cdot E^{i-1},$$

$$y_i^i = k_i,\quad y_j^i = y_j^{i-1} - k_i \cdot y_{i-j}^{i-1},\ j = 1,... i - 1$$

3) Result:

$pDst[i-1] = y_i^m,\ i = 1,...m,\quad pErr[0] = E^m$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pErr* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsNoOperation	Indicates no solution to the LPC problem ($E^i \leq 0$).

Schur

Calculates reflection coefficients using Schur algorithm.

```
IppStatus ippsSchur_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp32f*
   pErr, int scaleFactor);
IppStatus ippsSchur_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f*
   pErr);
```

Arguments

pSrc	Pointer to the input autocorrelations vector [*len+1*].
pDst	Pointer to the output reflection coefficients vector [*len*].
len	Length of the output vector.
pErr	Pointer to the result prediction error.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsSchur is declared in the ippsr.h file. This function uses Schur algorithm to compute reflection coefficients in the following steps:

1) Initialization:

$b_1^0 = pSrc[0]$, $a_j^0 = b_{j+1}^0 = pSrc[j]$, for $j = 1,\ldots len-1$, $a_{len}^0 = pSrc[len]$

2) Iterations for $i = 1,...len$:

$$k_i = -a_i^{i-1} / b_i^{i-1}$$

$$a_j^i = a_j^{i-1} + k_i b_j^{i-1} \quad , \quad b_j^i = b_{j-1}^{i-1} + k_i a_{j-1}^{i-1} \quad , \; j = i+1,... \; len$$

3) Result:

$$pDst[i-1] = k_i \; , \; i = 1,...len$$

$$pErr[0] = b_{len}^{len-1} + k_{len} \cdot a_{len}^{len-1} \; .$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pDst*, or *pErr* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsNoOperation`	Indicates no solution to the LPC problem ($b_i^i = 0$).

LPToSpectrum

Calculates smoothed magnitude spectrum.

```
IppStatus ippsLPToSpectrum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pDst,
   int order, Ipp32s val, int scaleFactor);
IppStatus ippsLPToSpectrum_32f(const Ipp32f* pSrc, int len, Ipp32f* pDst,
   int order, Ipp32f val);
```

Arguments

pSrc	Pointer to the input LPC coefficients vector [*len*].
pDst	Pointer to the output LP spectrum coefficients vector [2^{order}].
len	Number of LPC coefficients.

order	FFT order for spectrum calculation.
val	The value to add to spectrum.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLPToSpectrum is declared in the ippsr.h file. This function computes the first half of a linear prediction magnitude spectrum as follows:

$$pDst[k] = \frac{1}{\left| val - \sum_{i=1}^{len} pSrc[i-1] \cdot e^{-\frac{jik\pi}{N}} \right|}, \; k = 0,\ldots N\text{-}1, \; N = 2^{order}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or when it is greater or equal to $2^{order+1}$.
ippStsFftOrderErr	Indicates an error when the *order* value is incorrect.
ippStsDivByZero	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to +Inf and for integer operations is set to IPP_MAX_16S.

LPToCepstrum

Calculates cepstrum coefficients from linear prediction coefficients.

```
IppStatus ippsLPToCepstrum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   int scaleFactor);
```

```
IppStatus ippsLPToCepstrum_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the linear prediction coefficients [*len*].
pDst	Pointer to the cepstrum coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLPToCepstrum is declared in the ippsr.h file. This function calculates the cepstrum coefficients from the linear prediction coefficients according to the following formula:

$$pDst[k] = -\left(pSrc[k] + \frac{1}{k+1} \sum_{i=1}^{k} (k-i+1) \cdot pSrc[i-1] \cdot pDst[k-i] \right), k = 0 \ldots len-1 .$$

Return Value

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

CepstrumToLP

Calculates linear prediction coefficients from cepstrum coefficients.

```
IppStatus ippsCepstrumToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   int scaleFactor);
IppStatus ippsCepstrumToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the cepstrum coefficients [*len*].
pDst	Pointer to the linear prediction coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsCepstrumToLP is declared in the ippsr.h file. This function calculates the linear prediction coefficients from the cepstrum coefficients according to the following formula:

$$pDst[k] = -\left(pSrc[k] + \frac{1}{k+1} \sum_{i=1}^{k} (k-i+1) \cdot pSrc[k-i] \cdot pDst[i-1] \right), k = 0 \ldots len-1 .$$

Return Value

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

LPToReflection

Calculates the linear prediction reflection coefficients from the linear prediction coefficients.

```
IppStatus ippsLPToReflection_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
   len, int scaleFactor);
IppStatus ippsLPToReflection_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the linear prediction coefficients [*len*].
pDst	Pointer to the linear prediction reflection coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLPToReflection is declared in the ippsr.h file. This function calculates the linear prediction reflection coefficients according to the following formulae:

1) Initialization:

$$n = len\ ,\quad a_i^n = pSrc[i-1]\ ,\quad i = 1,...n$$

2) Iterations for $i = m,...1$:

$$k_i = a_i^i\ ,\quad a_j^{i-1} = \frac{a_j^i - a_i^i \cdot a_{i-j}^i}{1 - k_i^2}\ ,\quad j = 1,...i\text{-}1$$

3) Result:

$$pDst[i-1] = k_i\ ,\quad i = 1,...n\ .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsNoOperation	Indicates that reflection coefficients could not be calculated (that is, $\|k_i\| = 1$ for one of iterations).

ReflectionToLP

Calculates the linear prediction coefficients from the linear prediction reflection coefficients.

```
IppStatus ippsReflectionToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
   len, int scaleFactor);
IppStatus ippsReflectionToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the linear prediction reflection coefficients [*len*].
pDst	Pointer to the linear prediction coefficients [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsReflectionToLP is declared in the ippsr.h file. This function calculates the linear prediction reflection coefficients from the linear prediction coefficients according to the following formulae:

1) Initialization:

$n = len$, $k_i = pSrc[i-1]$, $i = 1,...n$

2) Iterations for $i = 1,...m$:

$a_i^i = k_i$, $a_j^i = a_j^{i-1} - k_i \cdot a_{i-j}^{i-1}$, $j = 1,...k\text{-}1$

3) Result:

$pDst[i-1] = a_i^n$, $i = 1,...n$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

ReflectionToAR

Converts reflection coefficients to area ratios.

```
IppStatus ippsReflectionToAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
   Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsReflectionToLAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
   Ipp16s* pDst, int len, Ipp32f val, int scaleFactor);
IppStatus ippsReflectionToLAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
   Ipp32f val);
IppStatus ippsReflectionToTrueAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
   Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToTrueAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
   len);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.
val	Threshold value (1 > *val* > 0).
srcShiftVal	Scale factor used in ippsReflectionToLAR function only. Refer to "Integer Scaling" in Chapter 2.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

These functions are declared in the ippsr.h file.
The function ippsReflectionToAR calculates approximate area ratios using the following formula:

$pDst[k] = \frac{1 - pSrc[k]}{1 + pSrc[k]}$, $k = 0,\ldots len\text{-}1$

The function ippsReflectionToLAR calculates logarithm of area ratios as given by:

$$pDst[k] = \begin{cases} \ln\frac{1 - val}{1 + val}, & if\ pSrc[k] \geq val \\ \ln\frac{1 - pSrc[k]}{1 + pSrc[k]}, & if\ val > pSrc[k] > -val,\ k = 0,\ldots len - 1 \\ \ln\frac{1 + val}{1 - val}, & if\ -val \geq pSrc[k] \end{cases}$$

The function ippsReflectionToTrueAR calculates area ratios using the formulae:

$pDst[0] = \frac{1 - pSrc[0]}{1 + pSrc[0]}$, $pDst[k] = pDst[k-1] \cdot \frac{1 - pSrc[k]}{1 + pSrc[k]}$, $k = 1,\ldots len\text{-}1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZero	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to: NaN , if the dividend vector element has zero value; +Inf , if the dividend vector element has positive value; -Inf , if the dividend vector element is negative. The value of the destination vector element for integer operations is set to: IPP_MAX_16S , if the dividend vector element is positive; IPP_MIN_16S , if the dividend vector element is negative.
ippStsRangeErr	Indicates an error when the condition $1 > val > 0$ is not met.

ReflectionToTilt

Calculates tilt for rise/fall/connection parameters.

```
IppStatus ippsReflectionToAbsTilt_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
   pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToAbsTilt_32f(const Ipp32f* pSrc1, const Ipp32f*
   pSrc2, Ipp32f* pDst, int len);
IppStatus ippsReflectionToTilt_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
   pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToTilt_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
   Ipp32f* pDst, int len);
```

Arguments

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

These functions are declared in the `ippsr.h` file.

The function `ippsReflectionToAbsTilt` converts rise and fall coefficients to absolute tilt as given by:

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{|pSrc1[k]| + |pSrc2[k]|}, \quad k = 0,\ldots len-1.$$

The function `ippsReflectionToTilt` converts rise and fall coefficients to tilt:

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{pSrc1[k] + pSrc2[k]}, \quad k = 0,\ldots len-1.$$

Return Value

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZero	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to: NaN , if the dividend vector element has zero value; +Inf , if the dividend vector element has positive value; -Inf , if the dividend vector element is negative. The value of the destination vector element for integer operations is set to: IPP_MAX_16S , if the dividend vector element is positive; IPP_MIN_16S , if the dividend vector element is negative.

PitchmarkToF0

Calculates rise and fall amplitude and duration for tilt.

```
IppStatus ippsPitchmarkToF0Cand_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
   len, int scaleFactor);
IppStatus ippsPitchmarkToF0Cand_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
   len);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsPitchmarkToF0Cand is declared in the ippsr.h file. This function calculates F0 candidate values from pitchmarks using the following formula:

$$pDst[0] = \frac{1}{pSrc[0]}, \quad pDst[k] = \frac{1}{pSrc[k] - pSrc[k-1]}, \quad k = 1,\ldots len\text{-}1.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZero	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to +Inf. The value of the destination vector element for integer operations is set to IPP_MAX_16S.

UnitCurve

Calculates tilt for rise and fall coefficicents.

```
IppStatus ippsUnitCurve_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal, Ipp16s*
   pDst, int len, int scaleFactor);
IppStatus ippsUnitCurve_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsUnitCurve_16s_ISfs(Ipp16s* pSrcDst, int srcShiftVal, int len,
   int scaleFactor);
IppStatus ippsUnitCurve_32f_I(Ipp32f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pSrcDst	Pointer to the input and destination vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.

srcShiftVal	Refer to "Integer Scaling" in Chapter 2.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsUnitCurve is declared in the ippsr.h file. This function calculates the destination vector according to the following formula:

$$pDst[k] = \begin{cases} 0, & if\ pSrc[k] < 0 \\ pSrc[k]^2, & if\ 0 \le pSrc[k] < 1 \\ 2-(2-pSrc[k])^2, & if\ 1 \le pSrc[k] \le 2 \\ 2, & if\ 2 < pSrc[k] \end{cases}, \text{ for } k = 0,\ldots len-1.$$

In-place function flavors ovewrite source vector with destination data.

Return Value

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

LPToLSP

Calculates line spectrum pairs vector from linear prediction coefficients.

```
IppStatus ippsLPToLSP_16s_Sfs(const Ipp16s* pSrcLP, int srcShiftVal, Ipp16s*
   pDstLSP, int len, int* nRoots, int nInt, int nDiv, int scaleFactor);
IppStatus ippsLPToLSP_32f(const Ipp32f* pSrcLP, Ipp32f* pDstLSP, int len, int*
   nRoots, int nInt, int nDiv);
```

Arguments

pSrcLP	Pointer to the input LP coefficients vector [*len*].
pDstLSP	Pointer to the output LSP coefficients vector [*len*].

len	Number of LP coefficients.
nRoots	Pointer to the number of LSP values found.
nInt	Number of intervals in zero crossing search.
nDiv	Number of interval bisections during search for roots.
srcShiftVal	Scale factor for *pSrcLP* values.
scaleFactor	Scale factor for *pDstLSP* values, refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLPToLSP is declared in the ippsr.h file. This function converts linear prediction (LP) coefficients to the line spectrum pair (LSP) representation and implements the algorithm generally used in speech coding.

Let $a_k = pSrcLP[k]$, $k = 0,\ldots len\text{-}1$ (or $a_k = pSrcLP[k] \cdot 2^{-srcShiftVal}$ in case of the ippsLPToLSP_16s_Sfs function) be the LP coefficients (see also the formula used in the ippsLinearPredictionNeg_Auto function to compute LP coefficients), and

$$A(z) = 1 - \sum_{k=0}^{len-1} a_k z^{-k} .$$

LSP coefficients are defined as the roots of symmetric and anti-symmetric polynomials of the form

$$F_1(z) = A(z) + z^{-len-1} \cdot A(z^{-1}), \quad F_2(z) = A(z) - z^{-len-1} \cdot A(z^{-1}),$$

with the exception of the roots equal to 1 and -1, which are excluded through the definition of the following polynomials $G_1(z)$ and $G_2(z)$:

$G_1(z) = F_1(z) / (1 + z^{-1})$, $G_2(z) = F_2(z) / (1 - z^{-1})$, if *len* is even;

$G_1(z) = F_1(z)$, $G_2(z) = F_2(z) / (1 - z^{-2})$, if *len* is odd.

These polynomials are symmetric, so that

$$G_i(z) = g_{i,m_i} \cdot z^{-m_i} + \sum_{k=0}^{m_i-1} g_{i,k} \cdot \left(z^{-k} + z^{-(2m_i-k)} \right),$$

and they can be represented in the form

$$G_i(e^{jw}) = e^{-jwm_i} \cdot G_i'(w) = g_{i,m_i} + 2 \cdot \sum_{k=0}^{m_i-1} g_{i,k} \cdot T_{m_i-k}(x) \text{ , for } i = 1,2 \text{ ,}$$

where $x = \cos w$, and $T_k(x) = \cos kw$ are the m-th order Chebyshev polynomials (see [Kab86]).

When len is even, the polynomial degrees are given by $m_1 = m_2 = len/2$, and

$g_{1,0} = g_{2,0} = 1$, $g_{1,k} = a_{k-1} + a_{len-k} - g_{1,k-1}$, $g_{2,k} = a_{k-1} - a_{len-k} + g_{2,k-1}$, for $k = 1,..m_1$.

When len is odd, $m_1 = (len+1)/2$, $m_2 = (len-1)/2$ and

$g_{1,0} = 1$, $g_{1,k} = -a_{k-1} - a_{len-k}$ for $k = 1,..m_1$,

$g_{2,-1} = 0$, $g_{2,0} = 1$, and if $m_2 > 0$ then $g_{2,k} = g_{2,k-2} + a_{len-k} - a_{k-1}$ for $k = 1,..m_2$.

Polynomials $G_1(z)$ and $G_2(z)$ have distinct interlaced roots on the unit circle that are found in cosine domain by checking for polynomials sign change at $nInt$ points equally spaced between 0 and π (zero crossing search) . The sign change interval is then bisected $nDiv$ times and after that the root is located by linear interpolation. The Chebyshev polynomials are used to evaluate $G_1(z)$ and $G_2(z)$. The roots are written to *pDstLSP* vector in ascending order.

The number of roots found is written to *nRoots*[0]. If all len roots were not found, the error status is returned.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcLP*, *pDstLSP* or *nRoots* pointer is NULL.
ippStsSizeErr	Indicates an error when len is less than or equal to 0, or *nInt* is less than *len*, or *nDiv* is less than 0.
ippStsNoRootFoundErr	Indicates that less than len roots were found.

LSPToLP

Converts line spectrum pairs vector to linear prediction coefficients.

```
IppStatus ippsLSPToLP_16s_Sfs(const Ipp16s* pSrcLSP, int srcShiftVal, Ipp16s*
   pDstLP, int len, int scaleFactor);
IppStatus ippsLSPToLP_32f(const Ipp32f* pSrcLSP, Ipp32f* pDstLP, int len);
```

Arguments

pSrcLSP	Pointer to the input LSP coefficients vector [*len*].
pDstLP	Pointer to the output LP coefficients vector [*len*].
len	Number of LSP and LP coefficients.
srcShiftVal	Scale factor for *pSrcLSP* values.
scaleFactor	Scale factor for *pDstLP* values, refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLSPToLP is declared in the ippsr.h file. This function converts line spectrum pair (LSP) values to linear prediction (LP) coefficients using the Kabal's method (see [Kab86]).

The polynomials

$G_1(e^{-jw}) = e^{-jwm_1} \cdot G_1'(w)$ and $G_2(e^{-jw}) = e^{-jwm_2} \cdot G_2'(w)$

are reconstructed from their roots *pSrcLSP*[k], $k = 0,\ldots len-1$, using Chebyshev representation.

Next the polynomials $F_1(z)$ and $F_2(z)$ and are reconstructed as:

$F_1(z) = G_1(z) \cdot (1 + z^{-1})$, $F_2(z) = G_2(z) \cdot (1 - z^{-1})$, if *len* is even;

$F_1(z) = G_1(z)$, $F_2(z) = G_2(z) \cdot (1 - z^{-2})$, if *len* is odd.

Finally, the linear prediction coefficients are determined from

$$A(z) = \frac{F_1(z) + F_2(z)}{2} = 1 - \sum_{k=0}^{len} a_k z^{-k}$$

and

$pDstLP[k] = a_k, k = 0,\ldots len-1.$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLSP* or *pDstLP* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsIncorrectLSP`	Indicates a warning that *pSrcLSP* elements are not valid LSP values (so that condition $-1 < \ldots < pSrcLSP[k+1] < pSrcLSP[k] < \ldots < 1$ is not met).

MelToLinear

Converts Mel-scaled values to linear scale values.

```
IppStatus ippsMelToLinear_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
   Ipp32f melMul, Ipp32f melDiv);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of input and output vectors.
melMul	Multiply factor in the Mel-scale equation.
melDiv	Divide factor in the Mel-scale equation.

Discussion

The function `ippsMelToLinear` is declared in the `ippsr.h` file. This function converts the Mel-frequency scale to the linear frequency scale:

$pDst[i] = melDiv \cdot \left[\exp\left(\frac{pSrc[i]}{melMul}\right) - 1\right]$, $i = 0,\ldots,len\text{-}1$.

Return Value

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

LinearToMel

Converts linear-scale values to Mel-scale values.

```
IppStatus ippsLinearToMel_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
   melMul, Ipp32f melDiv);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the input and output vectors.
melMul	Multiply factor in the Mel-scale equation.
melDiv	Divide factor in the Mel-scale equation.

Discussion

The function ippsLinearToMel is declared in the ippsr.h file. This function converts the linear frequency scale to the Mel-frequency scale:

$pDst[i] = melMul \cdot \ln\left(1 + \frac{pSrc[i]}{melDiv}\right)$, $i = 0,\ldots,len\text{-}1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or *melMul* or *melDiv* is equal to 0.

CopyWithPadding

Copies the input signal to the output with zero-padding.

```
IppStatus ippsCopyWithPadding_16s(const Ipp16s* pSrc, int lenSrc, Ipp16s*
   pDst, int lenDst);
IppStatus ippsCopyWithPadding_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
   pDst, int lenDst);
```

Arguments

pSrc	Pointer to the input vector [*lenSrc*].
pDst	Pointer to the output vector [*lenDst*].
lenSrc	Length of the input vector *pSrc*.
lenDst	Length of the output vector *pDst*.

Discussion

The function ippsCopyWithPadding is declared in the ippsr.h file. This function copies the input vector to the output vector. If the length of the output vector is bigger, the trailing elements are padded with zeroes.

Return Value

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

MelFBankGetSize

Gets the size of the Mel-frequency filter bank structure.

```
IppStatus ippsMelFBankGetSize_32s(int winSize, int nFilter, IppMelMode mode,
   int *pSize);
```

Arguments

winSize	Frame length in samples (32 ≤*winSize* ≤8192).
nFilter	Number of Mel-scale filter banks K (0 < *nFilter* ≤ *winSize*).
mode	Flag that determines the execution mode. Currently only IPP_FBANK_FREQWGT is supported.
pSize	Pointer to the variable to contain the size of the filter bank structure.

Discussion

The function ippsMelFBankGetSize is declared in the ippsr.h file. This function determines the size required for the Mel-frequency filter bank structure and associated storage. It should be called before memory allocation and before the call to ippsMelFBankInit.

Return Value

ippStsNoErr	Indicates no error.
ippStsFBankFlagErr	Indicates an error when the *mode* value is incorrect.
ippStsNullPtrErr	Indicates an error when *pSize* pointer is NULL.
ippStsSizeErr	Indicates an error when *nFilter* is out of range or when *winSize* is less than or equal to 0.

MelFBankInit

Initializes the structure for performing the Mel-frequency filter bank analysis.

```
IppStatus ippsMelFBankInit_32s(IppsFBankState_32s *pFBank, int *pFFTLen, int
   winSize, Ipp32s sampFreq, Ipp32s lowFreq, Ipp32s highFreq, int nFilter,
   Ipp32s melMulQ15, Ipp32s melDivQ15, IppMelMode mode);
```

Arguments

pFBank	Pointer to the Mel-scale filter bank structure to be initialixed.
pFFTLen	Pointer to the length of FFT (N = *pFFTLen* [0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
sampFreq	Input signal sampling frequency f_i (in Hz).
lowFreq	Start frequency f_{low} of the first band-pass filter (in Hz).
highFreq	End frequency f_{high} of the last band-pass filter (in Hz).
nFilter	Number of Mel-scale filter banks K.
melMulQ15	Real-valued mel-scale formula multiply factor in Q15.
melDivQ15	Real-valued mel-scale formula divisor in Q15.
mode	Flags that determine execution mode; can have the following values: IPP_FBANK_MELWGT – the function calculates filter bank weights in Mel-scale; IPP_FBANK_FREQWGT – the function calculates filter bank weights in the frequency space. One of the above two flags should necessarily be set. IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis.

Discussion

The function `ippsMelFBankInit` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis function. Mel filter coefficients are calculated in the same way as for the `ippsMelFBankInitAlloc` functions. Memory for the structure should be previously allocated, the size of this memory can be determined by the `ippsMelFBankGetSize` function.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pFBank`* or *`pFFTLen`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`winSize`*, *`nFilter`*, *`sampFreq`*, or *`lowFreq`* is less than or equal to 0.
`ippStsFBankFreqErr`	Indicates an error when *`highFreq`* is less than *`lowFreq`* or *`highFreq`* is greater than *`sampFreq/2`*.
`ippStsFBankFlagErr`	Indicates an error when the *`mode`* value is incorrect.

MelFBankInitAlloc

Allocates memory and initializes the structure for performing the Mel-frequency filter bank analysis.

```
IppStatus ippsMelFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
   int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f
   highFreq, int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
IppStatus ippsMelFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
   int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f
   highFreq, int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

Arguments

`pFBank`	Pointer to the Mel-scale filter bank structure to be created.

pFFTLen	Pointer to the length of FFT (*N = pFFTLen*[0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
sampFreq	Input signal sampling frequency f_i (in Hz).
lowFreq	Start frequency f_{low} of the first band-pass filter (in Hz).
highFreq	End frequency f_{high} of the last band-pass filter (in Hz).
nFilter	Number of Mel-scale filter banks *K*.
melMul	Mel-scale formula multiply factor.
melDiv	Mel-scale formula divisor.
mode	Flags that determine execution mode; can have the following values:
	`IPP_FBANK_MELWGT` – the function calculates filter bank weights in Mel-scale;
	`IPP_FBANK_FREQWGT` – the function calculates filter bank weights in the frequency space.
	One of the above two flags should necessarily be set.
	`IPP_POWER_SPECTRUM` – indicates that the FFT power spectrum is used during the filter bank analysis.

Discussion

The function `ippsMelFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis. The filter bank analysis is one of the major steps in the Mel-Frequency Cepstrum Coefficients (MFCC) feature calculation.

The Mel-frequency translation is accomplished according to the following equation:

$$mel(f) = melMul \cdot \ln\left(1 + \frac{f}{melDiv}\right)$$

The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

$$c_k = mel(f_{low}) + \frac{k}{K+1} \cdot [mel(f_{high}) - mel(f_{low})] ,$$

$$y_k = round\left\{\frac{N}{f_s} mel^{-1}(c_k)\right\},$$

for $k = 0...K+1$,

where *round*(x) rounds the value of x to the nearest integer (when *mode* is equal to IPP_FBANK_FREQWGT) or to the largest integer that is less than or equal to x (when *mode* is equal to IPP_FBANK_MELWGT), and N is the length of FFT.

If *mode* is IPP_FBANK_FREQWGT, then the filter outputs will be calculated (by the function ippsEvalFBank) according to the following formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}}^{y_k} \frac{i - y_{k-1} + 1}{y_k - y_{k-1} + 1} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{y_{k+1} - i + 1}{y_{k+1} - y_k + 1} \cdot x_i, \text{ for } k = 1...K, \quad (8.1)$$

where x_i is the magnitude of the corresponding FFT spectrums.

If *mode* is IPP_FBANK_MELWGT, the filter outputs will be calculated (by the function ippsEvalFBank) according to the formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}+1}^{y_k} \frac{mel\left(i\frac{f_s}{N}\right) - c_{k-1}}{c_k - c_{k-1}} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{c_{k+1} - mel\left(i\frac{f_s}{N}\right)}{c_{k+1} - c_k} \cdot x_i, k = 1...K \quad (8.2)$$

For the function flavor ippsMelFBankInitAlloc_32s, currently only IPP_FBANK_MELWGT mode is supported.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pFFTLen* pointer is NULL.
ippStsSizeErr	Indicates an error when *winSize*, *nFilter*, *sampFreq*, or *lowFreq* is less than or equal to 0.
ippStsFBankFreqErr	Indicates an error when *highFreq* is less than *lowFreq* or *highFreq* is greater than *sampFreq/2*.
ippStsFBankFlagErr	Indicates an error when the *mode* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

MelLinFBankInitAlloc

Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.

```
IppStatus ippsMelLinFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
   int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
   Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
   Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
IppStatus ippsMelLinFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
   int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
   Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
   Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

Arguments

pFBank	Pointer to the filter bank structure to be created.
pFFTLen	Pointer to the length of FFT (*N* = *pFFTLen*[0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
sampFreq	Input signal sampling frequency f_i (in Hz).
lowFreq	Start frequency f_{low} of the first band-pass filter (in Hz).
highLinFreq	End frequency f_{lin} of the last band-pass filter in linear scale (in Hz).
highFreq	End frequency f_{high} of the last band-pass filter (in Hz).
nFilter	Number of Mel-scale filter banks *K*.
nLinFilter	Number of linear-scale filter banks *L*.
melMul	Mel-scale formula multiply factor.
melDiv	Mel-scale formula divisor.
mode	Flags determining the function's execution mode; can have the following values:

IPP_FBANK_MELWGT – the function calculates filter bank weights in Mel-scale;

IPP_FBANK_FREQWGT – the function calculates filter bank weights in the frequency space.

One of the above two flags should necessarily be set.

IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis.

Discussion

The function ippsMelLinFBankInitAlloc is declared in the ippsr.h file. This function initializes the linear and Mel-frequency triangular filter banks. The first L filters are placed linearly on the frequency band from f_{low} to f_{lin}. The center of each filter bank (c_k in Mel-scale and y_k in FFT domain) is calculated as follows:

If $L = K$, then

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L+1} \cdot k\ ,\quad y_k = round\left\{\frac{N}{f_s} \cdot z_k\right\}\ ,\quad c_k = mel(z_k)\quad ,k = 0,...L+1$$

If $L < K$, then

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L} \cdot k\ ,\quad y_k = round\left\{\frac{N}{f_s} \cdot z_k\right\}\ ,\quad c_k = mel(z_k)\quad ,k = 0,...L$$

and

$$c_k = mel(f_{lin}) + \frac{k-L}{K-L+1} \cdot [mel(f_{high}) - mel(f_{lin})]\ ,$$

$$y_k = round\left\{\frac{N}{f_s} \cdot mel^{-1}(c_k)\right\},\ \text{for } k = L,...K+1\ ,$$

where $round(x)$ rounds the value of x to the nearest integer (when *mode* is equal to IPP_FBANK_FREQWGT) or to the largest integer that is less than or equal to x (when *mode* is IPP_FBANK_MELWGT), and N is the length of FFT.

If *mode* is IPP_FBANK_FREQWGT, the filter outputs are calculated (by the function ippsEvalFBank) according to the formula (8.1) given for the ippsMelFBankInitAlloc function.

If *mode* is IPP_FBANK_MELWGT, the filter output is calculated (by the function ippsEvalFBank) according to the formula (8.2).

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pFFTLen* pointer is NULL.
ippStsSizeErr	Indicates an error when *winSize*, *nFilter*, *sampFreq*, *lowFreq* is less than or equal to 0, or when *nLinFilter* is less than or equal to 1, or *nLinFilter* is greater than *nFilter*.
ippStsFBankFreqErr	Indicates an error when *highLinFreq* is less than *lowFreq*, or *highFreq* is less than *highLinFreq* , or *highFreq* is greater than *sampFreq/2*, or *(highLinFreq > lowFreq)&&(nLinFilter==0)*, or *(highLinFreq < highFreq)&&(nLinFilter==nFilter)* .
ippStsFBankFlagErr	Indicates an error when the *mode* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.

EmptyFBankInitAlloc

Initializes an empty filter bank structure.

```
IppStatus ippsEmptyFBankInitAlloc_16s(IppsFBankState_16s** pFBank, int*
   pFFTLen, int winSize, int nFilter, IppMelMode mode);
IppStatus ippsEmptyFBankInitAlloc_32f(IppsFBankState_32f** pFBank, int*
   pFFTLen, int winSize, int nFilter, IppMelMode mode);
```

Arguments

pFBank	Pointer to the filter bank structure to be created.
pFFTLen	Pointer to the length of FFT (*N* = *pFFTLen*[0]) used for the filter bank evaluation.
winSize	Frame length (in samples).
nFilter	Number of filters banks *K*.
mode	Flag determining the function's execution mode; can have the following value: IPP_POWER_SPECTRUM – indicates that the FFT power spectrum is used during the filter bank analysis.

Discussion

The function ippsEmptyFBankInitAlloc is declared in the ippsr.h file. This function initializes the filter bank structure for the *nFilter* filters. The filter bank center frequencies can be set by the function ippsFBankSetCenters and the filter weights can be set by the function ippsFBankSetCoeffs.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pFFTLen* pointer is NULL.
ippStsSizeErr	Indicates an error when *winSize* or *nFilter* or *pFBank* is less than or equal to 0.

ippStsMemAllocErr	Indicates an error when no memory was allocated.

FBankFree

Destroys the structure for the filter bank analysis.

```
IppStatus ippsFBankFree_16s(IppsFBankState_16s* pFBank);
IppStatus ippsFBankFree_32f(IppsFBankState_32f* pFBank);
```

Arguments

pFBank	Pointer to the filter bank structure.

Discussion

The function ippsFBankFree is declared in the ippsr.h file. This function destroys the filter bank structure and frees all memory associated with it.

Return Value

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

FBankGetCenters

Retrieves the center frequencies of the triangular filter banks.

```
IppStatus ippsFBankGetCenters_16s(const IppsFBankState_16s* pFBank, int*
   pCenters);
IppStatus ippsFBankGetCenters_32f(const IppsFBankState_32f* pFBank, int*
   pCenters);
```

Arguments

pFBank	Pointer to the filter bank structure.
pCenters	Pointer to the output vector that contains the center frequencies.

Discussion

The function ippsGetCenters is declared in the ippsr.h file. This function retrieves the indexes y_k (in FFT domain points) of the filter bank centers. The resulting array *pCenters* is of length *nFilter*+1. The filter bank structure *pFBank* must be initialized by either ippsMelFBankInitAlloc or ippsMelLinFBankInitAlloc function.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pCenters* pointer is NULL.
ippStsFBankErr	Indicates an error when filter centers are not valid after filter bank initialization by ippsEmptyFBankInitAlloc function.

FBankSetCenters

Sets the center frequencies of the triangular filter banks.

```
IppStatus ippsFBankSetCenters_16s(IppsFBankState_16s* pFBank, const int*
   pCenters);
IppStatus ippsFBankSetCenters_32s(IppsFBankState_32s* pFBank, const int*
   pCenters);
```

Arguments

pFBank	Pointer to the filter bank structure.
pCenters	Pointer to the vector that contains center frequencies.

Discussion

The function `ippsSetCenters` is declared in the `ippsr.h` file. This function sets the filter center indexes y_k in *pFBank*.
The indexes must meet the following conditions:

$0 \le ... \le y_k \le y_{k+1} \le ... \le N/2 , \quad i = 0,...K .$

If the *k*-th center is modified, the filter weight coefficients in the (*k*-1)-th, *k*-th and (*k*+1)-th filter banks must be also modified using the `ippsFBankSetCoeffs` function.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pFBank* or *pCenters* pointer is `NULL`.

FBankGetCoeffs

Retrieves the filter bank weight coefficients.

```
IppStatus ippsFBankGetCoeffs_16s(const IppsFBankState_16s* pFBank, int fIdx,
   Ipp32f* pCoeffs);
IppStatus ippsFBankGetCoeffs_32f(const IppsFBankState_32f* pFBank, int fIdx,
   Ipp32f* pCoeffs);
```

Arguments

pFBank	Pointer to the filter bank structure
fIdx	Filter index.
pCoeffs	Pointer to the output vector that contains the filter coefficients.

Discussion

The function ippsFBankGetCoeffs is declared in the ippsr.h file. This function copies the weight coefficients of the filter bank *fIdx* to the array *pCoeffs* which is of length $(y_{k+1} - y_{k-1} + 1)$.

Note that for ippsFBankGetCoeffs_16s function flavor, the filter output weights are represented in floating-point format.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pCoeffs* pointer is NULL.
ippStsSizeErr	Indicates an error when *fIdx* is less then 1 or greater then *nFilter*.
ippStsFBankErr	Indicates an error when *fIdx* filter coefficients are not available or valid.

FBankSetCoeffs

Sets the filter bank weight coefficients.

```
IppStatus ippsFBankSetCoeffs_16s(IppsFBankState_16s* pFBank, int fIdx, const
   Ipp32f* pCoeffs);
IppStatus ippsFBankSetCoeffs_32f(IppsFBankState_32f* pFBank, int fIdx, const
   Ipp32f* pCoeffs);
```

Arguments

pFBank	Pointer to the filter bank structure.
fIdx	Filter index.
pCoeffs	Pointer to the output coefficients vector.

Discussion

The function ippsFBankGetCoeffs is declared in the ippsr.h file. This function sets the weight coefficients of the filter bank *fIdx*. The vector *pCoeffs* contains the weight coefficients from y_{k-1} to y_{k+1}.

Note that for ippsFBankSetCoeffs_16s function flavor, the weight coefficients are represented in floating-point format and are saturated to the [-1,1] interval.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pFBank* or *pCoeffs* pointer is NULL.
ippStsSizeErr	Indicates an error when *fIdx* is less then 1 or greater then *nFilter*.
ippStsFBankErr	Indicates an error when the weight coefficients are not available or valid.

EvalFBank

Performs the filter bank analysis.

```
IppStatus ippsEvalFBank_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
   IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst, const
   IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
   IppsFBankState_32s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
   IppsFBankState_32f* pFBank);
```

Arguments

pSrc	Pointer to the source vector $[2^{pFFTOrder[0]}]$.
pDst	Pointer to the filter bank coefficients vector [*nFilter*].

pFBank	Pointer to the filter bank structure.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsEvalFBank is declared in the ippsr.h file. This function performs the filter bank analysis for the input vector *pSrc*.

The execution mode set during the filter bank structure initialization determines the use of the input vector as follows:

If the IPP_POWER_SPECTRUM flag is set, the source vector is the wave signals. The magnitude of input signal spectrum is calculated for the filter bank analysis as follows:

$$x_k = \left| \left(\sum_{i=0}^{N-1} pSrc[i] \cdot \exp\left(-j \cdot 2\pi \frac{ik}{N}\right) \right) \right| , \quad 0 \le k \le N/2.$$

Otherwise, the input vector is used directly for filter bank analysis:

$x_k = pSrc[i] , \quad 0 \le k \le N/2.$

Depending on the mode, either formula (8.1) or (8.2) is used to obtain the filter bank coefficients. The input vector *pSrc* is destroyed after the analysis.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsFBankErr	Indicates an error when *pFBank* structure is not ready for calculation.

DCTLifterGetSize_MulC0

Gets the size of the DCT structure.

```
IppStatus ippsDCTLifterGetSize_MulC0_16s(int lenDCT, int lenCeps, int *pSize);
```

Arguments

lenDCT	Length of the DCT ($0 < lenDCT \leq 8192$).
lenCeps	Number of the output coefficients not including C0 ($0 < lenCeps \leq lenDCT$).
pSize	Pointer to the variable to contain the size in bytes.

Discussion

The function `ippsDCTLifterGetSize_MulC0` is declared in the `ippsr.h` file. This function computes the size in bytes required for the DCT structure and associated storage.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSize* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error if *lenDCT* or *lenCeps* are out of bounds.

DCTLifterInit_MulC0

Initializes the structure to perform DCT and lift the DCT coefficients.

```
IppStatus ippsDCTLifterInit_MulC0_16s(IppsDCTLifterState_16s* pDCTLifter, int
   lenDCT, const Ipp32s* pLifterQ15, int lenCeps);
```

Arguments

pDCTLifter	Pointer to the structure to be initialized for the DCT calculation and liftering.
lenDCT	Length of the DCT ($0 < lenDCT \leq 8192$).
pLifterQ15	Pointer to the Q15 format liftering coefficients vector ($0.0_{Q15} \leq pLifterQ15[k] < 512.0_{Q15}$).

lenCeps	Number of the output coefficients not including C_0 ($0 < lenCeps \leq lenDCT$).

Discussion

The function ippsDCTLifterInit_MulC0 is declared in the ippsr.h file. This function initializes the structure for the DCT calculation and liftering similar to what is done by the ippsDCTLifterInitAlloc_MulC0 function. However, memory for the structure should be previously allocated, and the size of memory could be determined by the ippsDCTLifterGetSize_MulC0 function.

C_0 is stored as the last element of the output vector.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pLifterQ15* or *pDCTLifter* pointer is NULL.
ippStsSizeErr	Indicates an error when *lenDCT*, *lenCeps*, or *pLifterQ15*[k] is out of range.

DCTLifterInitAlloc

Initializes the structure and allocates memory to perform DCT and lift the DCT coefficients.

```
IppStatus ippsDCTLifterInitAlloc_16s(IppsDCTLifterState_16s** pDCTLifter,
   int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_16s(IppsDCTLifterState_16s** pDCTLifter,
   int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_16s(IppsDCTLifterState_16s** pDCTLifter,
   int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_16s(IppsDCTLifterState_16s**
   pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);
```

```
IppStatus ippsDCTLifterInitAlloc_32f(IppsDCTLifterState_32f** pDCTLifter,
   int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_32f(IppsDCTLifterState_32f** pDCTLifter,
   int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_32f(IppsDCTLifterState_32f** pDCTLifter,
   int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_32f(IppsDCTLifterState_32f**
   pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);
```

Arguments

pDCTLifter	Pointer to the structure to be created for the DCT calculation and liftering.
lenDCT	Length of the DCT.
lenCeps	Number of the output coefficients (not including C_0).
nLifter	Liftering factor.
pLifter	Pointer to the liftering coefficients vector.
val	The scale factor for the output coefficients (except C_0).
val0	The scale factor for C_0.

Discussion

The function `ippsDCTLifterInitAlloc` is declared in the `ippsr.h` file. This function initializes the structure for the DCT calculation and liftering.

The first DCT coefficient C_0 is usually ignored. Therefore, the output of the `ippsDCTLifter` function contains only C_1 to $C_{lenCeps}$ coefficients. However, if the `C0` suffix is specified, C_0 is stored as the last element of the output vector.

The liftering coefficients are calculated according to the below formulae:

For functions without both the `Mul` and `C0` suffixes,

$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val\ ,\ i = 1,\ldots lenCeps$$

For functions with the `C0` suffix,

$$l_0 = val0$$
$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val\ ,\ i = 1,\ldots lenCeps$$

For functions with the `Mul` suffix,

$$l_i = pLifter[i-1]\ ,\ i = 1,\ldots lenCeps$$

For functions that have both the `Mul` and `C0` suffixes:

$$l_0 = pLifter[lenCeps - 1]$$
$$l_i = pLifter[i-1]\ ,\ i = 1,\ldots lenCeps$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pLifter`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`lenDCT`*, *`lenCeps`*, or *`nLifter`* is less than or equal to 0, or when *`lenDCT`* is less than *`lenCeps`*.
`ippStsMemAllocErr`	Indicates an error when no memory was allocated.

DCTLifterFree

Destroys the structure used for the DCT and liftering.

```
IppStatus ippsDCTLifterFree_16s(IppsDCTLifterState_16s* pDCTLifter);
IppStatus ippsDCTLifterFree_32f(IppsDCTLifterState_32f* pDCTLifter);
```

Arguments

pDCTLifter	Pointer to the DCT and liftering structure.

Discussion

The function ippsDCTLifterFree is declared in the ippsr.h file. This function closes the DCT and liftering structure by freeing all memory associated with it.

Return Value

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

DCTLifter

Performs the DCT and lifts the DCT coefficients.

```
IppStatus ippsDCTLifter_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
   IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
IppStatus ippsDCTLifter_32s16s_Sfs (const Ipp32s* pSrc, Ipp16s* pDst, const
   IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
IppStatus ippsDCTLifter_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
   IppsDCTLifterState_32f* pDCTLifter);
```

Arguments

pSrc	Pointer to the source vector [*lenDCT*].
pDst	Pointer to the output vector [*lenCeps*] or [*lenCeps*+1].
pDCTLifter	Pointer to the DCT and liftering structure.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsDCTLifter is declared in the ippsr.h file. This function first performs the DCT and then lifts the DCT coefficients.

The DCT coefficients are calculated according to the formula:

$$y_i = \sum_{j=1}^{lenDCT} pSrc[j-1] \cdot \cos\left(\frac{\pi \cdot i \cdot (j-0.5)}{lenDCT}\right), \quad i = 0,\ldots lenCeps$$

The output coefficients are weighted as follows:

$$pDst[i-1] = l_i \cdot y_i, \quad i = 1,\ldots lenCeps$$

If the C_0 coefficient is required (*pDCTLifter* is initialized by functions with the `C0` suffix), it is stored as the last element of the output vector:

$$pDst[lenCeps] = l_0 \cdot y_0$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *pDst*, or *pFBank* pointer is `NULL`.

The following example shows how the feature processing functions can be used for MFCC feature calculation.

Example 8-1 MFCC feature calculation

```
/* Input: samples[] Input samples
sample_number Number of samples
Output: mfccs[12*(sample_num-240)/160] The resulting MFCC coefficients
*/

void Calc_MFCC (Ipp32f *samples, int sample_num, Ipp32f *mfccs) {
Ipp32f* frame_buffer,fbank_buffer,mfcc_cur;
IppsFBankState_32f *fbank;
IppsDCTLifterState_32f *dctl;
int fft_len,fft_order;
int i,j;
float LogEnergy;

/* Initialize the structures */
```

```
ippsMelFBankInitAlloc_32f(&fbank, /* return the structure pointer */
                  &fft_order, /* return the FFT length */
                  400, /* 25ms window/512 point FFT */
                  16000, /* sample rate */
                  64, /* lowest frequency of interest */
                  8000, /* highest frequency of interest */
                  24, /* number of filter banks */
                  1127.0, /* mel-scale factor 1 */
                  700.0, /* mel-scale factor 2 */
                  IPP_FBANK_MELWGT | IPP_POWER_SPECTRUM);
ippsDCTLifterInitAlloc_32f(&dctl, /* return the structure pointer */
                  24, /* filter bank channels */
                  12, /* number of MFCC coefficients */
                  22, /* liftering */
                  1.0); /* no scaling */
fft_len=1<<fft_order;
frame_buffer=ippsMalloc_32f(fft_len);
fbank_buffer=ippsMalloc_32f(24);
mfcc_cur=mfccs;

/* Calculate MFCC features */
for (i=j=0; i+400<sample_num; i+=160,mfcc_cur+=12,j++) {
       /* Organize the input wave data into a frame */
       ippsCopyWithPadding_32f(&samples[i],400,frame_buffer,fft_len);
       ippsDotProd_32f(frame_buffer,frame_buffer,fft_len,&LogEnergy);
       /* Pre-emphasize the input signal with factor 0.97 */
       ippsPreemphasize_32f(frame_buffer,400,0.97);
       frame_buffer[0]*=(1.0-0.97);
       /* Add the hamming window to the input signal */
       ippsWinHamming_32f_I(frame_buffer,400);
       /* Perform the filter bank analysis */
       ippsEvalFBank_32f(frame_buffer,fbank_buffer,fbank);
       ippsThreshold_LTVal_32f_I(fbank_buffer, 24, 1.0, 1.0);
       ippsLn_32f_I(fbank_buffer, 24);
       /* Perform the DCT analysis and liftering */
       ippsDCTLifter_32f(fbank_buffer,mfcc_cur,dctl);
       mfcc_cur[12] = (float) log ((double) LogEnergy);
```

```
}
/* Normalize log energy */
ippsNormEnergy_32f(mfccs+11,12,(sample_num-240)/160,50.0,1.0);
/* Destroy the structures after calculation */
ippsFree(fbank_buffer);
ippsFree(frame_buffer);
ippsFBankFree_32f(fbank);
ippsDCTLifterFree_32f(dctl);
   }
```

NormEnergy

Normalizes a vector of energy values.

```
IppStatus ippsNormEnergy_32f(Ipp32f* pSrcDst, int step, int height,
   Ipp32f silFloor, Ipp32f enScale);
IppStatus ippsNormEnergy_16s(Ipp16s* pSrcDst, int step, int height,
   Ipp16s silFloor, Ipp16s val, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_32f(Ipp32f* pSrcDst, int step, int height,
   Ipp32f silFloor, Ipp32f maxE, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_16s(Ipp16s* pSrcDst, int step, int height,
   Ipp16s silFloor, Ipp16s maxE, Ipp16s val, Ipp32f enScale);
```

Arguments

pSrcDst	Pointer to the input and output vector [*height*step*].
step	Sample step in the vector *pSrcDst*.
height	Number of samples for the normalization.
silFloor	Silence floor value.
val	Coefficient value.
maxE	Maximum energy value.
enScale	Energy scale.

Discussion

The function ippsNormEnergy is declared in the ippsr.h file. This function normalizes the input vector that contains energy values.
The normalization is performed as follows:

For functions without the RT suffix, the maximum energy value is calculated as

$$maxE = \max_{0 < i < height - 1} pSrcDst[i \cdot step]$$

For all functions, assuming $val = 1$ if not specified,

$minE = maxE - (silFloor \cdot \ln 10)/10$

$pSrcDst[i \cdot step] = val - (maxE - \max(pSrcDst[i \cdot step], minE)) \cdot enScale$,

for $0 \le i < height$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.

SumMeanVar

Calculates both the sum of a the vector and its square sum.

```
IppStatus ippsSumMeanVar_32f(const Ipp32f* pSrc, int srcStep, int height,
   Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_32f_I(const Ipp32f* pSrc, int srcStep, int height,
   Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32f(const Ipp16s* pSrc, int srcStep, int height,
   Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
```

```
IppStatus ippsSumMeanVar_16s32f_I(const Ipp16s* pSrc, int srcStep, int height,
   Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32s_Sfs(const Ipp16s* pSrc, int srcStep, int
   height, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsSumMeanVar_16s32s_ISfs(const Ipp16s* pSrc, int srcStep, int
   height, Ipp32s* pSrcDstMean, Ipp32s* pSrcDstVar, int width, int
   scaleFactor);
```

Arguments

pSrc	Pointer to the source vector [*height***srcStep*].
srcStep	Row step in the vector *pSrc* .
height	Number of rows in *pSrc*
pDstMean	Pointer to the destination vector that contains the sums [*width*].
pDstVar	Pointer to the destination vector that contains the square sums [*width*].
pSrcDstMean	Pointer to the source and destination vector that contains the sums [*width*].
pSrcDstVar	Pointer to the source and destination vector that contains the square sums [*width*].
width	Number of columns in the source vector *pSrc*.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsSumMeanVar is declared in the ippsr.h file. This function calculates both the sums and the square sums of the source vectors as follows:

$$pDstVar[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pDstMean[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j],$$

for $j = 0,...width - 1$.

The function ippsSumMeanVar_I performs the in-place calculation as given by:

$$pSrcDstVar[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pSrcDstMean[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j],$$

for $j = 0,...width - 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDstMean*, *pDstVar*, *pSrcDstMean*, or *pSrcDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcStep*, *width*, or *height* is less than or equal to 0, or when *width* is greater than *srcStep*.

NewVar

Calculates the variances given the sum and square sum accumulators.

```
IppStatus ippsNewVar_32f(const Ipp32f* pSrcMean, const Ipp32f* pSrcVar,
   Ipp32f* pDstVar, int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32f_I(const Ipp32f* pSrcMean, Ipp32f* pSrcDstVar,
   int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32s_Sfs(const Ipp32s* pSrcMean, const Ipp32s* pSrcVar,
   Ipp32s* pDstVar, int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
IppStatus ippsNewVar_32s_ISfs(const Ipp32s* pSrcMean, Ipp32s* pSrcDstVar,
   int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
```

Arguments

pSrcMean	Pointer to the vector that accumulates sums [*width*].
pSrcVar	Pointer to the vector that accumulates square sums[*width*].
pSrcDstVar	Pointer to the square sum accumulators and the resulting variance vector [*width*].
pDstVar	Pointer to the result variance vector [*width*].
width	Length of the variance vector *pDstVar*.
val1, val2	Constant coefficients.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsNewVar is declared in the ippsr.h file. This function calculates the variance values given the sum and
square sum accumulators as follows:

$$pDstVar[i] = (pSrcVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2 \ , \quad i=0...width-1$$

whereas the in-place function ippsNewVar_I uses the following formula:

$$pSrcDstVar[i] = (pSrcDstVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2 \ ,$$

for $i = 0,...width - 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcMean*, *pSrcVar*, *pDstVar*, or *pSrcDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* is less than or equal to 0.

RecSqrt

Calculates square roots of a vector and their reciprocals.

```
IppStatus ippsRecSqrt_32s_Sfs(Ipp32s* pSrcDst, int len, Ipp32s val, int
   scaleFactor);
IppStatus ippsRecSqrt_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
IppStatus ippsRecSqrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
   Ipp32s val, int scaleFactor);
```

Arguments

pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the vector *pSrcDst*.
val	Threshold for processed source values.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsRecSqrt is declared in the ippsr.h file. This function calculates the square root of a vector and then takes the reciprocals. The operations are as follows:

$$pSrcDst[j] = \begin{cases} val & , \ if\ pSrcDst[j] < val \\ \dfrac{1}{\sqrt{pSrcDst[j]}} & , \ otherwise \end{cases}$$

for $j = 0,\ldots,len - 1$.

Return Value

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

ippStsInvZero	Indicates a warning that all *pSrcDst*[i] are less than *val*.

AccCovarianceMatrix

Accumulates covariance matrix.

```
IppStatus ippsAccCovarianceMatrix_16s64f_D2L(const Ipp16s** mSrc, int height,
   const Ipp16s* pMean, Ipp64f** mSrcDst, int width, Ipp64f val);
IppStatus ippsAccCovarianceMatrix_32f64f_D2L(const Ipp32f** mSrc, int height,
   const Ipp32f* pMean, Ipp64f** mSrcDst, int width, Ipp64f val);
IppStatus ippsAccCovarianceMatrix_16s64f_D2(const Ipp16s* pSrc, int srcStep,
   int height, const Ipp16s* pMean, Ipp64f* pSrcDst, int width, int dstStep,
   Ipp64f val);
IppStatus ippsAccCovarianceMatrix_32f64f_D2(const Ipp32f* pSrc, int srcStep,
   int height, const Ipp32f* pMean, Ipp64f* pSrcDst, int width, int dstStep,
   Ipp64f val);
```

Arguments

pSrc	Pointer to the input vector [*height*srcStep*].
mSrc	Pointer to the input matrix [*height*][*width*].
srcStep	Row step in *pSrc*.
pMean	Pointer to the mean vector [*width*].
width	Length of the input matrix row, also length of the mean and variance vectors.
pSrcDst	Pointer to the result vector [*width*dstStep*].
mSrcDst	Pointer to the result matrix [*width*][*width*].
dstStep	Row step in *pSrcDst*.
height	Number of rows in the input matrix.
val	Value to multiply to each distance.

Discussion

The function ippsAccCovarianceMatrix is declared in the ippsr.h file. This function accumulates destination covariance matrix elements according to formulas below.

For functions with the D2 suffix,

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] + val \cdot \sum_{k=0}^{height-1} (pSrc[k \cdot srcStep + i] - pMean[j]) \cdot (pSrc[k \cdot srcStep + j] - pMean[j])$$

$$pSrcDst[j \cdot dstStep + i] = pSrcDst[i \cdot dstStep + j],$$

for $i = 0,\ldots width - 1$, $j = i,\ldots width - 1$.

For functions with the D2L suffix,

$$mSrcDst[i][j] = mSrcDst[i][j] + val \cdot \sum_{k=0}^{height-1} (mSrc[k][i] - pMean[j]) \cdot (mSrc[k][j] - pMean[j])$$

$$mSrcDst[j][i] = mSrcDst[i][j],$$

for $i = 0,\ldots width - 1$, $j = i,\ldots width - 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *mSrc*, *pMean*, *pSrcDst*, or *mSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *srcStep* or *dstStep* is less than *width*.

Derivative Functions

Functions described in this section are used to calculate first and second derivatives of feature vectors. These functions process the sequence of input feature vectors and generate the corresponding output feature vectors that contain derivatives.

The input sequence a_0 ,..., a_{N-1} is stored as a one-dimensional array of length $N \cdot M$, where N is the number of feature vectors and M is the dimension of each feature a_i. Similarly, the output sequence b_0 ,..., b_{N-1} is also stored as a one-dimensional array of length $N \cdot K$, where K is the dimension of each feature b_j.
Certain constraints on values of M and K must be observed, specifically,

$K \geq M$;

$K \geq 2M$, for generating first derivatives;

$K \geq 3M$, for generating both first and second derivatives.

The `ippsCopyColumn` function copies the input sequence to the output sequence, that is, places the base features into the output sequence. The base features are placed in the first M elements of each output vector. The function `ippsEvalDelta` is then used to calculate

the derivatives. While these two functions are used for general derivative calculation, the functions `ippsDelta` and `ippsDeltaDelta` provide combined but specific functionalities.

As the derivative operation accesses feature vectors both in the history and in the future, special treatment is required for the first `winSize` (delta window size) features, as well as for the last `winSize` features. For the first `winSize` features, the first feature vector is usually repeated to provide the history. For the last `winSize` features, the last feature vector is repeated to provide the future information.

CopyColumn

Copies the input sequence into the output sequence.

```
IppStatus ippsCopyColumn_16s_D2(const Ipp16s* pSrc, int srcWidth,
   Ipp16s* pDst, int dstWidth, int height);
IppStatus ippsCopyColumn_32f_D2(const Ipp32f* pSrc, int srcWidth,
   Ipp32f* pDst, int dstWidth, int height);
```

Arguments

pSrc	Pointer to the input feature sequence [*height*srcWidth*].
srcWidth	Length of each input feature vector.
pDst	Pointer to the output feature sequence [*height*dstWidth*].
dstWidth	Length of each output feature vector.
height	Number of features in the sequence.

Discussion

The function ippsCopyColumn is declared in the ippsr.h file. This function copies the input feature sequence into the output sequence as follows:

$pDst[j \cdot dstWidth + i] = pSrc[j \cdot srcWidth + i]$, $0 \le i < srcWidth$, $0 \le j < height$

The unspecified elements in the output sequence remain unchanged.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *srcWidth* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *dstWidth* is less than *srcWidth.*

EvalDelta

Calculates the derivatives of feature vectors.

```
IppStatus ippsEvalDelta_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step, int
   width, int offset, int winSize, Ipp16s val, int scaleFactor);
IppStatus ippsEvalDelta_32f_D2(Ipp32f* pSrcDst, int height, int step, int
   width, int offset, int winSize, Ipp32f val);
IppStatus ippsEvalDeltaMul_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step,
   const Ipp16s* pVal, int width, int offset, int winSize, int scaleFactor);
IppStatus ippsEvalDeltaMul_32f_D2(Ipp32f* pSrcDst, int height, int step, const
   Ipp32f* pVal, int width, int offset, int winSize);
```

Arguments

pSrcDst	Pointer to the input and output sequence [*height*step*].
height	Number of features in *pSrcDst*.
step	Length of each feature in *pSrcDst*.
width	The number of derivatives to be calculated for each feature.
offset	Offset to place the derivative values.
winSize	The delta window size
val	The delta coefficient.
pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsEvalDelta is declared in the ippsr.h file. This function calculates the derivatives for the input feature sequence. The base feature ranges from *offset* to (*offset* + *width* - 1) in each feature vector. The output derivatives are stored in each feature vector next to the base features ranging from (*offset* + *width*) to (*offset* + 2**width* - 1). The operations are detailed as follows:

For ippsEvalDelta functions,

$$pSrcDst[i \cdot step + width + j] = val \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$0 \le i < height, \quad offset \le j < offset + width$.

For ippsEvalDeltaMul functions,

$$pSrcDst[i \cdot step + width + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$0 \le i < height, \quad offset \le j < offset + width$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, or *winSize* is less than or equal to 0; or *offset* is less than 0; or *height* is less than 2**winSize*.
ippStsStrideErr	Indicates an error when *step* is less than *offset*+2**width*.

Delta

Copies the base features and calculates the derivatives of feature vectors.

```
IppStatus ippsDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
   pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);
```

```
IppStatus ippsDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
   pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);
IppStatus ippsDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
   pDst, int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippsDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
   pDst, int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippsDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
   int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
   scaleFactor);
IppStatus ippsDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
   int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
   scaleFactor);
IppStatus ippsDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal, int
   srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
IppStatus ippsDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal, int
   srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Arguments

pSrc	Pointer to the input feature sequence [*height*srcWidth*].
srcWidth	Length of the feature vector in the input sequence *pSrc*.
pDst	Pointer to the output feature sequence
dstStep	Length of the feature vector in the output sequence *pDst*.
height	Number of feature vectors.
val	Delta coefficient.
deltaMode	Execution mode.
pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions ippsDelta and ippsDeltaMul are declared in the ippsr.h file. These functions provide a combined functionality of ippsCopyColumn and ippsEvalDelta. First, the input feature vectors are copied to the output sequence. Then the derivatives are calculated.

In the subsequent discussion, the following conditions hold:

The function suffix `Win1` or `Win2` specifies the delta window size, *winSize* = 1 or 2, respectively;

The function `ippsDelta` implies the following constraints on the delta coefficients:

$pVal[j] \equiv val, \quad \forall j.$

The execution mode *deltaMode* provides additional controls along the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to `IPP_DELTA_BEGIN|IPP_DELTA_END`

   Perform the offline delta feature calculation. All base features are assumed available at the time of the calculation. The base features are copied from the input stream *pSrc* to the output stream *pDst* as follows:

   $$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \quad 0 \le i < height, \quad 0 \le j < srcWidth. \tag{8.4}$$

   Then the derivatives are calculated as given by:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[\min(i+k, height-1) \cdot dstStep + j] - pDst[\max(i-k, 0) \cdot dstStep + j]\}, \tag{8.5}$$

   for $0 \le i < height, \quad 0 \le j < srcWidth$.

2. *deltaMode* is equal to `0`

Perform the online delta feature calculation. The input features are the current segment of a continuous stream. The function `ippsDelta` calculates the partial delta features accordingly. First, the base features are copied as follows:

$$pDst[(i + 2 \cdot winSize) \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \tag{8.6}$$

for $0 \le i < height, \quad 0 \le j < srcWidth$.

Then the derivatives are calculated as given by:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[(i+k) \cdot dstStep + j] - pDst[(i-k) \cdot dstStep + j]\}, \quad (8.7)$$

for $0 \le i - winSize < height$, $0 \le j < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the derivatives in $pDst[k \cdot dstStep + srcWidth + j]$ are assumed available through a previous derivative calculation for $0 \le j < srcWidth$, $0 \le i < 2 \cdot winSize$, and $0 \le k < winSize$.

3. *deltaMode* is equal to `IPP_DELTA_BEGIN`

   Perform the partial online delta feature calculation, where the beginning of the input stream is known. First, the base features are copied as in equation (8.4). Then, the derivatives are calculated as follows:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[(i+k) \cdot dstStep + j] - pDst[\max(i-k, 0) \cdot dstStep + j]\}, \quad (8.8)$$

   for $0 \le i < height - winSize$, $0 \le j < srcWidth$.

4. *deltaMode* is equal to `IPP_DELTA_END`

   Perform the partial online delta feature calculation, where the ending of the input stream is known. First, the base features are copied as in equation (8.6). Then, the derivatives are calculated as follows:

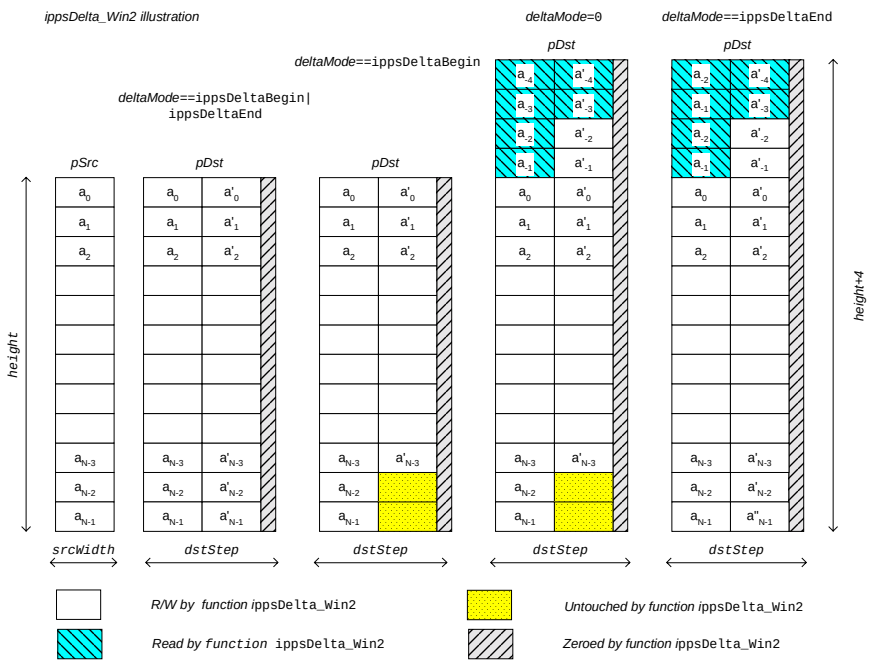
$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{ pDst[\min(i+k, height + 2 \cdot winSize - 1) \cdot dstStep + j] - pDst[(i-k) \cdot dstStep + j]\}$$

for $0 \le i - winSize < height + winSize$, $0 \le j < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the derivatives in $pDst[k \cdot dstStep + srcWidth + j]$ are assumed available through a previous derivative calculation for $0 \le j < srcWidth$, $0 \le i < 2 \cdot winSize$, and $0 \le k < winSize$.

The following figure illustrates the above four delta calculation modes:

Figure 8-1 Execution Modes of ippsDelta_Win2 function



Return Value

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pVal* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcWidth* is less than or equal to 0; or *height* is less than or equal to 0; or *height* is less than or equal to *winSize*+1 when IPP_DELTA_BEGIN is set; or *height* is less than 0 when IPP_DELTA_BEGIN is not set.
ippStsStrideErr	Indicates an error when *dstStep* is less than 2**srcWidth*.

DeltaDelta

Copies the base features and calculates their first and second derivatives.

```
IppStatus ippsDeltaDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
   Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
   deltaMode, int scaleFactor);
IppStatus ippsDeltaDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
   Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
   deltaMode, int scaleFactor);
IppStatus ippsDeltaDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
   pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);
IppStatus ippsDeltaDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
   pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);
IppStatus ippsDeltaDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
   pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
   int scaleFactor);
IppStatus ippsDeltaDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
   pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
   int scaleFactor);
IppStatus ippsDeltaDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f*
   pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
IppStatus ippsDeltaDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f*
   pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Arguments

pSrc	Pointer to the input feature sequence [*height*srcWidth*].
srcWidth	Length of the input feature in the input sequence *pSrc*.
pDst	Pointer to the output feature sequence.
dstStep	Length of the output feature in the output sequence *pDst*.
height	Number of feature vectors.
val1, val2	The first and second delta coefficients.
deltaMode	Execution mode.
pVal	Pointer to the delta coefficients vector [*width*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions `ippsDeltaDelta` and `ippsDeltaDeltaMul` are declared in the `ippsr.h` file. These functions provide a combined functionality of `ippsCopyColumn` and double operations of `ippsEvalDelta`. First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated.

In the subsequent discussion, the following conditions hold:

The function suffix `Win1` or `Win2` specifies the delta window size, *winSize* = 1 or 2, respectively;

The function `ippsDeltaDelta` implies the following constraints on the delta coefficients:

$pVal[j] \equiv val1, \quad \forall j$, for the first derivative calculation and

$pVal[j] \equiv val2, \quad \forall j$, for the second derivative calculation.

The execution mode *deltaMode* provides additional controls along the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to IPP_DELTA_BEGIN|IPP_DELTA_END

   Perform the offline delta-delta feature calculation. All base features are assumed available at the time of the calculation. First, the base features are copied from the input stream *pSrc* to the output stream *pDst* according to (8.4). Then the first derivatives are calculated as given by (8.5) for $0 \le i <$ *height*, $0 \le j <$ *srcWidth*. Finally, the second derivatives are calculated also by formula (8.5) for $0 \le i <$ *height*, $0 \le j -$ *srcWidth* $<$ *srcWidth*.

2. *deltaMode* is equal to 0

   Perform the online delta-delta feature calculation. The input features are the current segment of a continuous stream. The function ippsDeltaDelta calculates the partial delta-delta features accordingly.

   First, the base features are copied as follows:

   $$pDst[(i + 3 \cdot winSize) \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \quad (8.9)$$

   for $0 \le i <$ *height*, $0 \le j <$ *srcWidth*.

   Then the first derivatives are calculated as given by (8.7), for $0 \le i - 2 \cdot winSize <$ *height* and $0 \le j <$ *srcWidth*.

   Finally, the second derivatives are calculated also by formula (8.7) for $0 \le i -$ *winSize* $<$ *height*, $0 \le j -$ *srcWidth* $<$ *srcWidth*.

   In this execution mode, the base features in $pDst[i \cdot dstStep + j]$, the first derivatives in $pDst[k \cdot dstStep + srcWidth + j]$, and the second derivatives in $pDst[l \cdot dstStep + 2 \cdot srcWidth + j]$ are assumed available through a previous delta-delta calculation for $0 \le j <$ *srcWidth*, $0 \le i < 3 \cdot winSize$, $0 \le k < 2 \cdot winSize$, and $0 \le l <$ *winSize*.

3. *deltaMode* is equal to IPP_DELTA_BEGIN

   Perform the partial online delta-delta feature calculation, where the beginning of the input stream is known. Initially, the base features are copied as in equation (8.4). Then, the first derivatives are calculated as given by (8.8), for $0 \le i <$ *height* $-$ *winSize* and $0 \le j <$ *srcWidth*.

   Finally, the second derivatives are calculated also by formula (8.8) for $0 \le i <$ *height* $- 2 \cdot$ *winSize* and $0 \le j -$ *srcWidth* $<$ *srcWidth*.

4. *deltaMode* is equal to IPP_DELTA_END

Perform the partial online delta-delta feature calculation, where the ending of the input stream is known. Initially, the base features are copied as in equation (8.9). Then, the first derivatives are calculated as follows:

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{ \quad (8.10)$$

$$pDst[\min(i+k, height+3 \cdot winSize-1) \cdot dstStep + j] - pDst[(i-k) \cdot dstStep + j]\}$$

for $0 \le i - 2 \cdot winSize < height + winSize$, $0 \le j < srcWidth$.

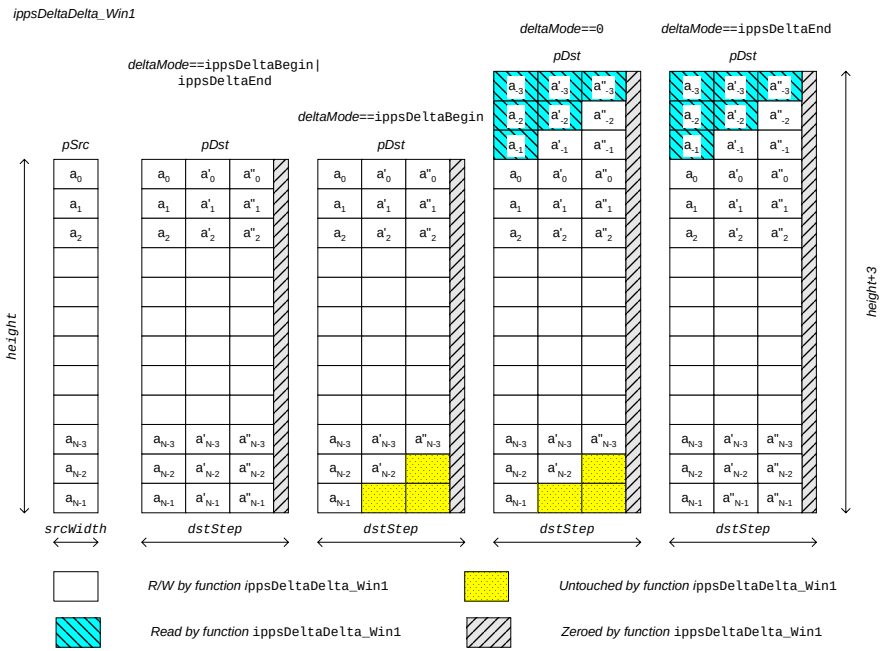
Finally, the second derivatives are calculated also according to (8.10) where the indexes *i* and *j* vary in the following ranges:

$0 \le i - winSize < height + 2 \cdot winSize$, $0 \le j - srcWidth < srcWidth$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$, the first derivatives in $pDst[k \cdot dstStep + srcWidth + j]$, and the second derivatives in $pDst[l \cdot dstStep + 2 \cdot srcWidth + j]$ are assumed available through a previous delta-delta calculation for $0 \le j < srcWidth$, $0 \le i < 3 \cdot winSize$, $0 \le k < 2 \cdot winSize$, and $0 \le l < winSize$.

The following figure illustrates the above four delta-delta calculation modes:

Figure 8-2 Execution Modes of `ippsDeltaDelta_Win1` function



Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`*, *`pDst`* or *`pVal`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`srcWidth`* is less than or equal to 0; or *`height`* is less than or equal to 0; or *`height`* is less than or equal to `2*(`*`winSize`*`+1)` when `IPP_DELTA_BEGIN` is set; or *`height`* is less than 0 when `IPP_DELTA_BEGIN` is not set.
`ippStsStrideErr`	Indicates an error when *`dstStep`* is less than `3*`*`srcWidth`*.

Pitch Super Resolution

This section describes functions that can be used for implementing the pitch resolution algorithm (see [Med91]).

CrossCorrCoeffDecim

Calculates vector of cross correlation coefficients with decimation.

```
IppStatus ippsCrossCorrCoeffDecim_16s32f(const Ipp16s* pSrc1, const Ipp16s*
   pSrc2, int maxLen, int minLen, Ipp32f* pDst, int dec);
```

Arguments

pSrc1	Pointer to the first input vector [*maxLen*].
pSrc2	Pointer to the second input vector [*maxLen*].
pDst	Pointer to the output vector [(*maxLen-minLen*)/*dec*+1].
maxLen	Maximal length of cross correlation.
minLen	Minimal length of cross correlation.
dec	Decimation step.

Discussion

The function ippsCrossCorrCoeffDecim is declared in the ippsr.h file. This function calculates the vector of cross correlation coefficients with lengths from *minLen* to *maxLen* using the decimation step *dec*.
Computations are performed as given by:

$$pDst[k] = \begin{cases} 0, & if\ (x^k, x^k)_{dec} \cdot (y^k, y^k)_{dec} = 0 \\ \dfrac{(x^k, y^k)_{dec}}{\sqrt{(x^k, x^k)_{dec}} \cdot \sqrt{(y^k, y^k)_{dec}}}, & otherwise \end{cases}$$

where

$x_i^k = pSrc1[maxLen - minLen - dec \cdot k + i]$, $y_i^k = pSrc2[i]$,

$i = 0,...minLen + dec \cdot k - 1$, $k = 0,...(maxLen - minLen)/dec$,

and $(a, b)_{dec}$ denotes the decimated dot product of two vectors a and b:

$$(a, b)_{dec} = \sum_{i=0}^{(len-1)/dec} a[i \cdot dec] \times b[i \cdot dec]$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *minLen* or *dec* is less than or equal to 0, or if *maxLen* is less than *minLen*.

CrossCorrCoeff

Calculates the cross correlation coefficient.

```
IppStatus ippsCrossCorrCoeff_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   int len, Ipp32f* pResult);
IppStatus ippsCrossCorrCoeffPartial_16s32f(const Ipp16s* pSrc1, const Ipp16s*
   pSrc2, int len, Ipp32f val, Ipp32f* pResult);
```

Arguments

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*].
len	Length of the input vectors.
pResult	Pointer to the result cross correlation coefficient value.
val	Value used as the squared magnitude of the first vector.

Discussion

The functions ippsCrossCorrCoeff and ippsCrossCorrCoeffPartial are declared in the ippsr.h file. These functions calculate the cross correlation coefficient of two vectors according to the formulas given below.

For the ippsCrossCorrCoeff function:

$$pResult[0] = \begin{cases} 0, & if\ (\boldsymbol{x}, \boldsymbol{x}) \cdot (\boldsymbol{y}, \boldsymbol{y}) = 0 \\ \dfrac{(\boldsymbol{x}, \boldsymbol{y})}{\sqrt{(\boldsymbol{x}, \boldsymbol{x})} \cdot \sqrt{(\boldsymbol{y}, \boldsymbol{y})}}, & otherwise \end{cases}$$

For the ippsCrossCorrCoeffPartial function:

$$pResult[0] = \begin{cases} 0, & if\ val \cdot (\boldsymbol{y}, \boldsymbol{y}) = 0 \\ \dfrac{(\boldsymbol{x}, \boldsymbol{y})}{\sqrt{val} \cdot \sqrt{(\boldsymbol{y}, \boldsymbol{y})}}, & otherwise \end{cases}$$

where

$\boldsymbol{x}_i = pSrc1[i-1]$, $\boldsymbol{y}_i = pSrc2[i-1]$, for $i = 1, \ldots len$,

and $(\boldsymbol{a}, \boldsymbol{b})$ denotes the dot product of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsBadArgErr	Indicates an error when *val* is less than 0.

CrossCorrCoeffInterpolation

Calculates interpolated cross correlation coefficient.

```
IppStatus ippsCrossCorrCoeffInterpolation_16s32f(const Ipp16s* pSrc1, const
   Ipp16s* pSrc2, int len, Ipp32f* pBeta, Ipp32f* pResult);
```

Arguments

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*+1].
len	Length of the input vectors.
pResult	Pointer to the result cross correlation coefficient value.
pBeta	Pointer to the result value of fractional part of the pitch period.

Discussion

The function ippsCrossCorrCoeffInterpolation is declared in the ippsr.h file. This function calculates interpolated cross correlation coefficient *pResult* of two vectors and the fractional part *pBeta* of the pitch period according to the following formulas:

$$pBeta[0] = \beta = \frac{(x, y') \cdot (y, y) - (x, y) \cdot (y, y')}{(x, y') \cdot [(y, y) - (y, y')] + (x, y) \cdot [(y', y') - (y, y')]},$$

and if $0 \le \beta < 1$, then

$$pResult[0] = \frac{(1-\beta) \cdot (x, y) + \beta \cdot (x, y')}{\sqrt{(x, x) \cdot ((1-\beta)^2 \cdot (y, y) + 2\beta \cdot (1-\beta) \cdot (y, y') + \beta^2 \cdot (y', y'))}},$$

where

$x_i = pSrc1[i-1]$, $y_i = pSrc2[i-1]$, $y_i' = pSrc2[i]$, $i = 1, \ldots len$,

and **(*a*, *b*)** denotes the dot product of two vectors ***a*** and ***b***.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, *pBeta* or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *srcStep* or *dstStep* is less than *width*.
ippStsDivByZero	Indicates a warning that the denominator in formula for *pBeta*[0] has zero value. Operation execution is not aborted.

Model Evaluation

This section describes functions that evaluate the acoustic and language models.

AddNRows

Adds N vectors from a vector array.

```
IppStatus ippsAddNRows_32f_D2(Ipp32f* pSrc, int height, int offset,
   int step, Ipp32s* pInd, Ipp16u* pAddInd, int rows, Ipp32f* pDst,
   int width, Ipp32f weight);
```

Arguments

pSrc	Pointer to the vector array [*height*step*].
height	Number of rows in the vector array *pSrc*.
offset	Offset to the vector of interest in the vector array *pSrc*.
step	Row step in the vector array *pSrc*.
pInd	Pointer to the index vector [*rows*].
pAddInd	Pointer to the additional index vector [*rows*].
rows	Number of vectors to be added.
pDst	Pointer to the output vector [*width*].
width	Length of the output vector *pDst*.
weight	Weight value to be added to the output.

Discussion

The function ippsAddNRows is declared in the ippsr.h file. This function calculates the sum of the *rows* number of vectors, according to the sum of the indexing vectors *pInd* and *pAddInd*, from the vector array *pSrc* as follows:

$$pDst[k] = weight + \sum_{i=0}^{rows} pSrc[k + offset + step \cdot (pInd[i] + pAddInd[i])],$$

$0 \le k < width$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pInd*, *pAddInd*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, or *rows* is less than or equal to 0; or (*pInd*[*i*]+*pAddInd*[*i*])or *offset* is less than 0; or (*pInd*[*i*]+*pAddInd*[*i*]) is greater than or equal to *height*.
ippStsStrideErr	Indicates an error when *step* is less than *width* + *offset*.

ScaleLM

Scales vector elements with thresholding.

```
IppStatus ippsScaleLM_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len, Ipp16s
   floor, Ipp16s scale, Ipp32s base);
IppStatus ippsScaleLM_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
   floor, Ipp32f scale, Ipp32f base);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the vector *pSrc* or *pDst*.
floor	Threshold value.
scale	Scaling factor.
base	Additive factor.

Discussion

The function ippsScaleLM is declared in the ippsr.h file. This function sets threshold on the input vector *pSrc* and scales the vector elements as follows:

$$pDst[n] = scale \cdot \max(pSrc[n],\ floor) + base,\ 0 \le n < len$$

Return Value

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

LogAdd

Adds two vectors in the logarithmic representation.

```
IppStatus ippsLogAdd_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len,
   IppHintAlgorithm hint);
IppStatus ippsLogAdd_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
   IppHintAlgorithm hint);
IppStatus ippsLogAdd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
   scaleFactor, IppHintAlgorithm hint);
IppStatus ippsLogAdd_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
   scaleFactor, IppHintAlgorithm hint);
```

Arguments

pSrc	Pointer to the first input vector [*len*].
pSrcDst	Pointer to the second input vector and also the output vector [*len*].
len	Number of elements in the input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.
hint	Suggestion for using specific code for logarithmic addition.

Discussion

The function ippsLogAdd is declared in the ippsr.h file. This function adds the *pSrc* and *pSrcDst* vectors, whose elements are in the logarithmic representation.
For functions with floating point arguments, the output vector *pSrcDst*, also in the logarithmic representation, is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} + e^{pSrcDst[i]}), \quad 0 \le i < len.$$

For functions with integer arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} + e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$0 \le i < len.$

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The values you can enter for the *hint* argument are listed in Table 7-3, "Flag and Hint Arguments".

Return Value

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

LogSub

Subtracts a vector from another vector, in the logarithmic representation.

```
IppStatus ippsLogSub_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsLogSub_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsLogSub_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
   int scaleFactor);
IppStatus ippsLogSub_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len,
   int scaleFactor);
```

Arguments

pSrc	Pointer to the first input vector [*len*].
pSrcDst	Pointer to the second input and the output vector [*len*].
len	Number of elements in the input and output vectors.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLogSub is declared in the ippsr.h file. This function subtracts the vector *pSrcDst* from the vector *pSrc*, both of them in the logarithmic representation. For functions with floating point arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} - e^{pSrcDst[i]}), \quad 0 \le i < len$$

For functions with integer arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} - e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$0 \le i < len$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsRangeErr	Indicates an error when *pSrc*[*i*] is less than or equal to *pSrcDst*[*i*].

LogSum

Sums vector elements in the logarithmic representation.

```
IppStatus ippsLogSum_32f(const Ipp32f* pSrc, Ipp32f* pResult, int len,
   IppHintAlgorithm hint);
IppStatus ippsLogSum_64f(const Ipp64f* pSrc, Ipp64f* pResult, int len,
   IppHintAlgorithm hint);
IppStatus ippsLogSum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pResult, int len,
   int scaleFactor, IppHintAlgorithm hint);
IppStatus ippsLogSum_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pResult, int len,
   int scaleFactor, IppHintAlgorithm hint);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pResult	Pointer to the result value.
len	Number of elements in the input vector.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.
hint	Suggestion for using specific code for logarithmic addition.

Discussion

The function ippsLogSum is declared in the ippsr.h file. This function sums the source vector elements that are taken in logarithmic representation. For functions with floating point arguments, the output value, also in logarithmic representation, is calculated as follows:

$$pResult[0] = \ln \sum_{i=0}^{len-1} e^{pSrc[i]}$$

For functions with integerarguments, the output value is calculated as follows:

$$pResult[0] = 2^{scaleFactor} \cdot \ln\left(\sum_{i=0}^{len-1} e^{pSrc[i] \cdot 2^{-scaleFactor}}\right)$$

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The values you can enter for the *hint* argument are listed in Table 7-3, "Flag and Hint Arguments".

Return Value

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

MahDistSingle

Calculates the Mahalanobis distance for a single observation vector.

```
IppStatus ippsMahDistSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
   pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, int scaleFactor);
IppStatus ippsMahDistSingle_16s32f(const Ipp16s* pSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean, const
   Ipp32f* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean, const
   Ipp64f* pVar, int len, Ipp64f* pResult);
IppStatus ippsMahDistSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
   const Ipp32f* pVar, int len, Ipp64f* pResult);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pMean	Pointer to the mean vector [*len*].
pVar	Pointer to the variance vector [*len*].

len	Number of elements in the input, mean, and variance vectors.
pResult	Pointer to the result.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsMahDistSingle is declared in the ippsr.h file. This function calculates the Mahalanobis distance for the input vector *pSrc*, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows:

$$pResult[0] = \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pMean*, *pVar*, or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

MahDist

Calculates the Mahalanobis distances for multiple observation vectors.

```
IppStatus ippsMahDist_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
   pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
   Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
   pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int height);
```

```
IppStatus ippsMahDist_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, const
   Ipp64f* pVar, int width, Ipp64f* pDst, int height);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pDst	Pointer to the result vector [*height*].
height	Number of rows in the input matrix *mSrc*.

Discussion

The function ippsMahDist is declared in the ippsr.h file. This function calculates the Mahalanobis distances for multiple input vectors, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows:
For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, \; 0 \le i < height.$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, \; 0 \le i < height.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pMean*, *pVar*, or *pDst* pointer is NULL.

ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

MahDistMultiMix

Calculates the Mahalanobis distances for multiple means and variances.

```
IppStatus ippsMahDistMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
   int step, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDistMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
   mVar, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDistMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
   int step, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
IppStatus ippsMahDistMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
   mVar, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
```

Arguments

pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in mean and variance vectors.
pSrc	Pointer to the input vector [*width*].
width	Length of the input vector *pSrc*.
pDst	Pointer to the result vector [*height*].
height	Length of the result vector *pDst*.

Discussion

The function ippsMahDistMultiMix is declared in the ippsr.h file. This function calculates the Mahalanobis distances for a single observation vector but multiple mean and variance pairs as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2, 0 \le i < height.$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2, 0 \le i < height.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussSingle

Calculates the observation probability for a single Gaussian with an observation vector.

Case 1: Operation for the inverse diagonal covariance matrix

IppStatus ippsLogGaussSingle_16s32s_Sfs(const Ipp16s* *pSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *len*, Ipp32s* *pResult*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGaussSingle_Scaled_16s32f(const Ipp16s* *pSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *len*, Ipp32f* *pResult*, Ipp32f *val*, int *scaleFactor*);

IppStatus ippsLogGaussSingle_32f(const Ipp32f* *pSrc*, const Ipp32f* *pMean*, const Ipp32f* *pVar*, int *len*, Ipp32f* *pResult*, Ipp32f *val*);

IppStatus ippsLogGaussSingle_32f64f(const Ipp32f* *pSrc*, const Ipp32f* *pMean*, const Ipp32f* *pVar*, int *len*, Ipp64f* *pResult*, Ipp64f *val*);

IppStatus ippsLogGaussSingle_64f(const Ipp64f* *pSrc*, const Ipp64f* *pMean*, const Ipp64f* *pVar*, int *len*, Ipp64f* *pResult*, Ipp64f *val*);

IppStatus ippsLogGaussSingle_Low_16s32s_Sfs(const Ipp16s* *pSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *len*, Ipp32s* *pResult*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGaussSingle_LowScaled_16s32f(const Ipp16s* *pSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *len*, Ipp32f* *pResult*, Ipp32f *val*, int *scaleFactor*);

Case 2: Operation for the diagonal covariance matrix

IppStatus ippsLogGaussSingle_DirectVar_16s32s_Sfs(const Ipp16s* *pSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *len*, Ipp32s* *pResult*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGaussSingle_DirectVarScaled_16s32f(const Ipp16s* *pSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *len*, Ipp32f* *pResult*, Ipp32f *val*, int *scaleFactor*);

IppStatus ippsLogGaussSingle_DirectVar_32f(const Ipp32f* *pSrc*, const Ipp32f* *pMean*, const Ipp32f* *pVar*, int *len*, Ipp32f* *pResult*, Ipp32f *val*);

IppStatus ippsLogGaussSingle_DirectVar_32f64f(const Ipp32f* *pSrc*, const Ipp32f* *pMean*, const Ipp32f* *pVar*, int *len*, Ipp64f* *pResult*, Ipp64f *val*);

IppStatus ippsLogGaussSingle_DirectVar_64f(const Ipp64f* *pSrc*, const Ipp64f* *pMean*, const Ipp64f* *pVar*, int *len*, Ipp64f* *pResult*, Ipp64f *val*);

Case 3: Operation for the identity covariance matrix

IppStatus ippsLogGaussSingle_IdVar_16s32s_Sfs(const Ipp16s* *pSrc*, const Ipp16s* *pMean*, int *len*, Ipp32s* *pResult*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGaussSingle_IdVarScaled_16s32f(const Ipp16s* *pSrc*, const Ipp16s* *pMean,* int *len*, Ipp32f* *pResult*, Ipp32f *val,* int *scaleFactor*);

```
IppStatus ippsLogGaussSingle_IdVar_32f(const Ipp32f* pSrc, const Ipp32f*
   pMean, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_IdVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
   pMean, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_IdVar_64f(const Ipp64f* pSrc, const Ipp64f*
   pMean, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_IdVarLow_16s32s_Sfs(const Ipp16s* pSrc, const
   Ipp16s* pMean, int len, Ipp32s* pResult, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussSingle_IdVarLowScaled_16s32f(const Ipp16s* pSrc, const
   Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int scaleFactor);
```

Case 4: Operation for the block diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_BlockDVar_16s32s_Sfs(const Ipp16s* pSrc, const
   Ipp16s* pMean, const IppsBlockDMatrix_16s * pBlockVar, int len, Ipp32s*
   pResult, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussSingle_BlockDVarScaled_16s32f(const Ipp16s* pSrc, const
   Ipp16s* pMean, const IppsBlockDMatrix_16s* pBlockVar, int len, Ipp32f*
   pResult, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussSingle_BlockDVar_32f(const Ipp32f* pSrc, const Ipp32f*
   pMean, const IppsBlockDMatrix_32f* pBlockVar, int len, Ipp32f* pResult,
   Ipp32f val);
IppStatus ippsLogGaussSingle_BlockDVar_32f64f(const Ipp32f* pSrc, const
   Ipp32f* pMean, const IppsBlockDMatrix_32f * pBlockVar, int len, Ipp64f*
   pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_BlockDVar_64f(const Ipp64f* pSrc, const Ipp64f*
   pMean, const IppsBlockDMatrix_64f * pBlockVar, int len, Ipp64f* pResult,
   Ipp64f val);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pMean	Pointer to the mean vector [*len*].
pVar	Pointer to the variance vector [*len*].
pBlockVar	Pointer to the block diagonal variance matrix.
len	Number of elements in the input, mean, and variance vectors.
pResult	Pointer to the result.

val	Gaussian constant.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function ippsLogGaussSingle is declared in the ippsr.h file. This function calculates the observation probability for a single observation vector and a Gaussian mixture component. The result *pResult* is in the logarithmic representation.
For functions without the DirectVar, IdVar, or BlockDVar suffix, the covariance matrix is assumed to be inverse diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

For functions with the DirectVar suffix, the covariance matrix is assumed to be diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2 / pVar[i]$$

For functions with the IdVar suffix, the covariance matrix is assumed to be identity:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2$$

For functions with the BlockDVar suffix, the covariance matrix is assumed to be block diagonal:

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} \sum_{j=0}^{len-1} (pSrc[i] - pMean[i]) \cdot V[i][j] \cdot (pSrc[j] - pMean[j])$$

where $V[i][j]$ is the element of the block diagonal matrix *pVar*.

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Value

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

LogGauss

Calculates the observation probability for a single Gaussian with multiple observation vectors.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGauss_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height,
   Ipp32s val, int scaleFactor);
IppStatus ippsLogGauss_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val, int
   scaleFactor);
IppStatus ippsLogGauss_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
   Ipp32f val, int scaleFactor);
IppStatus ippsLogGauss_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
   pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val,
   int scaleFactor);
IppStatus ippsLogGauss_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
   pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);
IppStatus ippsLogGauss_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
   Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);
```

IppStatus ippsLogGauss_64f_D2(const Ipp64f* *pSrc*, int *step*, const Ipp64f* *pMean*, const Ipp64f* *pVar*, int *width*, Ipp64f* *pDst*, int *height*, Ipp64f *val*);

IppStatus ippsLogGauss_64f_D2L(const Ipp64f** *mSrc*, const Ipp64f* *pMean*, const Ipp64f* *pVar*, int *width*, Ipp64f* *pDst*, int *height*, Ipp64f *val*);

IppStatus ippsLogGauss_Low_16s32s_D2Sfs(const Ipp16s* *pSrc*, int *step*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32s* *pDst*, int *height*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGauss_Low_16s32s_D2LSfs(const Ipp16s** *mSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32s* *pDst*, int *height*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGauss_LowScaled_16s32f_D2(const Ipp16s* *pSrc*, int *step*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32f* *pDst*, int *height*, Ipp32f *val*, int *scaleFactor*);

IppStatus ippsLogGauss_LowScaled_16s32f_D2L(const Ipp16s** *mSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32f* *pDst*, int *height*, Ipp32f *val*, int *scaleFactor*);

Case 2: Operation for the identity covariance matrix

IppStatus ippsLogGauss_IdVar_16s32s_D2Sfs(const Ipp16s* *pSrc*, int *step*, const Ipp16s* *pMean*, int *width*, Ipp32s* *pDst*, int *height*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGauss_IdVar_16s32s_D2LSfs(const Ipp16s** *mSrc*, const Ipp16s* *pMean*, int *width*, Ipp32s* *pDst*, int *height*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGauss_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int *step*, const Ipp16s* *pMean*, int *width*, Ipp32f* *pDst*, int *height*, Ipp32f *val,* int *scaleFactor*);

IppStatus ippsLogGauss_IdVarScaled_16s32f_D2L(const Ipp16s** *mSrc*, const Ipp16s* *pMean*, int *width*, Ipp32f* *pDst*, int *height*, Ipp32f *val,* int *scaleFactor*);

IppStatus ippsLogGauss_IdVar_32f_D2(const Ipp32f* *pSrc*, int *step*, const Ipp32f* *pMean*, int *width*, Ipp32f* *pDst*, int *height*, Ipp32f *val*);

IppStatus ippsLogGauss_IdVar_32f_D2L(const Ipp32f** *mSrc*, const Ipp32f* *pMean*, int *width*, Ipp32f* *pDst*, int *height*, Ipp32f *val*);

IppStatus ippsLogGauss_IdVar_64f_D2(const Ipp64f* *pSrc*, int *step*, const Ipp64f* *pMean*, int *width*, Ipp64f* *pDst*, int *height*, Ipp64f *val*);

IppStatus ippsLogGauss_IdVar_64f_D2L(const Ipp64f** *mSrc*, const Ipp64f* *pMean*, int *width*, Ipp64f* *pDst*, int *height*, Ipp64f *val*);

```
IppStatus ippsLogGauss_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
   const Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
   scaleFactor);
IppStatus ippsLogGauss_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc, const
   Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
   scaleFactor);
IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
   const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int
   scaleFactor);
IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
   Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int
   scaleFactor);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
step	Row step in the input vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pDst	Pointer to the result vector [*height*].
height	Number of rows in the input matrix, also the length of the result vector *pDst*.
val	Gaussian constant.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function ippsLogGauss is declared in the ippsr.h file. This function calculates the observation probability for multiple observation vectors for a Gaussian mixture component. The result *pResult* is in the logarithmic representation.

For functions without the IdVar suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional D2 suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, 0 \le i < height$$

For functions with the additional `D2L` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, \ 0 \le i < height.$$

For functions with the `IdVar` suffix, the covariance matrix is assumed to be identity:

For functions with the additional `D2` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2, \ 0 \le i < height$$

For functions with the additional `D2L` suffix,

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2, \ 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *mSrc*, *pMean*, *pVar*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.

ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussMultiMix

Calculates the observation probability for multiple Gaussian mixture components.

```
IppStatus ippsLogGaussMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const Ipp16s*
   pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
   int scaleFactor);

IppStatus ippsLogGaussMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
   Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
   int scaleFactor);

IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
   Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst,
   int height, int scaleFactor);

IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean, const
   Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst, int height,
   int scaleFactor);

IppStatus ippsLogGaussMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
   int step, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
   mVar, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
   int step, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
   mVar, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean, const
   Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int
   height, int scaleFactor);

IppStatus ippsLogGaussMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean, const
   Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
   int scaleFactor);
```

```
IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2(const Ipp16s* pSrc, int
   step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int
   height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, int
   scaleFactor);
```

Arguments

pMean	Pointer to the mean vector [*height*step*].
pVar	Pointer to the variance vector [*height*step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in the mean and variance vectors.
pSrc	Pointer to the input vector [*width*].
pSrcDst	Pointer to the input and destination vector [*height*].
width	Number of columns in the mean matrix *mMean*.
height	Number of rows in the mean matrix *mMean*.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function ippsLogGaussMultiMix is declared in the ippsr.h file. This function calculates the observation probability for multiple Gaussian mixture components. The results are in the logarithmic representation.
The calculations are as follows:

For functions with the D2 suffix,

$$pSrcDst[i] = pSrcDst[i] - 0.5 \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2$$

$0 \le i < height$.

For functions with the D2L suffix,

$$pSrcDst[i] = pSrcDst[i] - 0.5 \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \quad ,$$

$0 \le i < height$.

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

LogGaussMax

Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMax_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
   Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
   pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
   Ipp32s val, int scaleFactor);
```

IppStatus ippsLogGaussMax_Scaled_16s32f_D2(const Ipp16s* *pSrc*, int *step*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32f* *pDst*, int *height*, Ipp32f *val,* int *scaleFactor*);

IppStatus ippsLogGaussMax_Scaled_16s32f_D2L(const Ipp16s** *mSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32f* *pSrcDst*, int *height*, Ipp32f *val,* int *scaleFactor*);

IppStatus ippsLogGaussMax_32f_D2(const Ipp32f* *pSrc*, int *step*, const Ipp32f* *pMean*, const Ipp32f* *pVar*, int *width*, Ipp32f* *pSrcDst*, int *height*, Ipp32f *val*);

IppStatus ippsLogGaussMax_32f_D2L(const Ipp32f** *mSrc*, const Ipp32f* *pMean*, const Ipp32f* *pVar*, int *width*, Ipp32f* *pSrcDst*, int *height*, Ipp32f *val*);

IppStatus ippsLogGaussMax_64f_D2(const Ipp64f* *pSrc*, int *step*, const Ipp64f* *pMean*, const Ipp64f* *pVar*, int *width*, Ipp64f* *pSrcDst*, int *height*, Ipp64f *val*);

IppStatus ippsLogGaussMax_64f_D2L(const Ipp64f** *mSrc*, const Ipp64f* *pMean*, const Ipp64f* *pVar*, int *width*, Ipp64f* *pSrcDst*, int *height*, Ipp64f *val*);

IppStatus ippsLogGaussMax_Low_16s32s_D2Sfs(const Ipp16s* *pSrc*, int *step*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32s* *pSrcDst*, int *height*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGaussMax_Low_16s32s_D2LSfs(const Ipp16s** *mSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32s* *pSrcDst*, int *height*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGaussMax_LowScaled_16s32f_D2(const Ipp16s* *pSrc*, int *step*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32f* *pSrcDst*, int *height*, Ipp32f *val*, int *scaleFactor*);

IppStatus ippsLogGaussMax_LowScaled_16s32f_D2L(const Ipp16s** *mSrc*, const Ipp16s* *pMean*, const Ipp16s* *pVar*, int *width*, Ipp32f* *pSrcDst*, int *height*, Ipp32f *val*, int *scaleFactor*);

Case 2: Operation for the identity covariance matrix

IppStatus ippsLogGaussMax_IdVar_16s32s_D2Sfs(const Ipp16s* *pSrc*, int *step*, const Ipp16s* *pMean*, int *width*, Ipp32s* *pSrcDst*, int *height*, Ipp32s *val*, int *scaleFactor*);

IppStatus ippsLogGaussMax_IdVar_16s32s_D2LSfs(const Ipp16s** *mSrc*, const Ipp16s* *pMean*, int *width*, Ipp32s* *pSrcDst*, int *height*, Ipp32s *val*, int *scaleFactor*);

```
IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
   const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
   int scaleFactor);
IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
   Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
   scaleFactor);
IppStatus ippsLogGaussMax_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
   Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
   pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
   Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussMax_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
   pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
   const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int
   height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
   Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
   step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst,
   int height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
   Ipp32f val, int scaleFactor);
```

Arguments

pSrc	Pointer to the observation vector [*height*step*].
mSrc	Pointer to the observation matrix [*height*][*width*].
step	Row step in the observation vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pSrcDst	Pointer to the likelihood vector [*height*].

height	Length of the vector *pSrcDst*.
val	Gaussian constant.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function ippsLogGaussMax is declared in the ippsr.h file. This function calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector *pSrcDst* using the "maximum" operation.
For functions without the IdVar suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional D2 suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the additional D2L suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the IdVar suffix, the covariance matrix is assumed to be identity:

For functions with the additional D2 suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the additional D2L suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with Low suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pMean*, *pVar*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussMaxMultiMix

Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.

```
IppStatus ippsLogGaussMaxMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const
   Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
   Ipp32s* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
   Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal, Ipp32s*
   pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
   Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
   Ipp32f* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean,
   const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
   Ipp32f* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
   pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
   pSrcDst, int height);

IppStatus ippsLogGaussMaxMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
   mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
   int height);

IppStatus ippsLogGaussMaxMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
   pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
   pSrcDst, int height);

IppStatus ippsLogGaussMaxMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
   mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
   int height);

IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean, const
   Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
   Ipp32s* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean,
   const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
   Ipp32s* pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2(const Ipp16s* pMean,
   const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f*
   pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mMean,
   const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
   Ipp32f* pSrcDst, int height, int scaleFactor);
```

Arguments

pMean	Pointer to the mean vector [*height*step*].
pVar	Pointer to the variance vector [*height*step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
step	Row step in the mean and variance vectors.
pSrc	Pointer to the input vector [*width*].
width	Number of columns in the mean and variance matrices.
pVal	Pointer to the weight constant vector [*height*].
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of rows in the mean and variance matrices.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function ippsLogGaussMaxMultiMix is declared in the ippsr.h file. This function calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst* using the "maximum" operation. The covariance matrix is assumed to be inverse diagonal. For functions with the D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2 \quad ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) \ , \ 0 \le i < height.$$

For functions with the D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \quad ,$$

$pSrcDst[i] = \max(pSrcDst[i], V[i])$, $0 \le i < height$.

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, *pVal*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *width* or *height* is less than or equal to 0.
`ippStsStrideErr`	Indicates an error when *step* is less than *width*.

LogGaussAdd

Calculates the likelihood probability for multiple observation vectors.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussAdd_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f
   val, int scaleFactor);
IppStatus ippsLogGaussAdd_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s* pMean,
   const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
   scaleFactor);
IppStatus ippsLogGaussAdd_32f_D2(const Ipp32f** pSrc, int step, const Ipp32f* pMean,
   const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
   Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
```

```
IppStatus ippsLogGaussAdd_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
   pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f
   val);
IppStatus ippsLogGaussAdd_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
   const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
   const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
   height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
   Ipp32f val, int scaleFactor);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
   const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
   int scaleFactor);
IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
   Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val,
   int scaleFactor);
IppStatus ippsLogGaussAdd_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
   Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
   pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
   Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
   pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
   step, const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f
   val, int scaleFactor);
IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const
   Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
   scaleFactor);
```

Arguments

pSrc	Pointer to the observation vector [*height***step*].
mSrc	Pointer to the observation matrix [*height*][*width*].

step	Row step in the observation vector *pSrc*.
pMean	Pointer to the mean vector [*width*].
pVar	Pointer to the variance vector [*width*].
width	Length of the mean and variance vectors.
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of observation vectors.
val	Weight constant.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function ippsLogGaussAdd is declared in the ippsr.h file. This function calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector *pSrcDst*.

For functions without the IdVar suffix, the covariance matrix is assumed to be inverse diagonal:

For functions with the additional D2 suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the additional D2L suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the IdVar suffix, the covariance matrix is assumed to be identity:

For functions with the additional D2 suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the additional D2L suffix,

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with Low suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pMean*, *pVar*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussAddMultiMix

Calculates the likelihood probability for multiple Gaussian mixture components.

```
IppStatus ippsLogGaussAddMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
   Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f pVal,
   Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean,
   const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
   Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
   pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
   pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
   mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
   int height);
IppStatus ippsLogGaussAddMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
   pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
   pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
   mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
   int height);
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2(const Ipp16s* pMean,
   const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f*
   pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mMean,
   const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
   Ipp32f* pSrcDst, int height, int scaleFactor);
```

Arguments

pMean	Pointer to the mean vector [*height*step*].
pVar	Pointer to the variance vector [*height*step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].

step	Row step in mean and variance vectors (in *pMean* elements).
pSrc	Pointer to the input vector [*width*].
width	Length of the mean and variance matrices.
pVal	Pointer to the weight constant vector [*height*].
pSrcDst	Pointer to the likelihood vector [*height*].
height	Number of Gaussian mixture components.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function ippsLogGaussAddMultiMix is declared in the ippsr.h file. This function calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst*. The covariance matrix is assumed to be inverse diagonal.
For functions with the D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

For functions with the D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \quad ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) \ , \ 0 \le i < height.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with Low suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Value

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when *pSrc*, *mMean*, *mVar*, *pMean*, *pVar*, *pVal*, or *pSrcDst* pointer is `NULL`.

`ippStsSizeErr` Indicates an error when *width* or *height* is less than or equal to 0.

`ippStsStrideErr` Indicates an error when *step* is less than *width*.

LogGaussMixture

Calculates the likelihood probability for the Gaussian mixture.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMixture_Scaled_16s32f_D2(const Ipp16s* pSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int width, const
   Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_Scaled_16s32f_D2L(const Ipp16s* pSrc, const
   Ipp16s** mMean, const Ipp16s** mVar, int height, int width, const Ipp32f*
   pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2(const Ipp16s* pSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int width, const
   Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2L(const Ipp16s* pSrc, const
   Ipp16s** mMean, const Ipp16s** mVar, int height, int width, const Ipp32f*
   pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMean,
   const Ipp32f* pVar, int height, int step, int width, const Ipp32f* pVal,
   Ipp32f* pResult);
IppStatus ippsLogGaussMixture_32f_D2L(const Ipp32f* pSrc, const Ipp32f**
   mMean, const Ipp32f** mVar, int height, int width, const Ipp32f* pVal,
   Ipp32f* pResult);
```

```
IppStatus ippsLogGaussMixture_64f_D2(const Ipp64f* pSrc, const Ipp64f* pMean,
   const Ipp64f* pVar, int height, int step, int width, const Ipp64f* pVal,
   Ipp64f* pResult);
IppStatus ippsLogGaussMixture_64f_D2L(const Ipp64f* pSrc, const Ipp64f**
   mMean, const Ipp64f** mVar, int height, int width, const Ipp64f* pVal,
   Ipp64f* pResult);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, const
   Ipp16s* pMean, int height, int step, int width, const Ipp32f* pVal, Ipp32f*
   pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2L(const Ipp16s* pSrc, const
   Ipp16s** mMean, int height, int width, const Ipp32f* pVal, Ipp32f* pResult,
   int scaleFactor);
IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc,
   const Ipp16s* pMean, int height, int step, int width, const Ipp32f* pVal,
   Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2L(const Ipp16s* pSrc,
   const Ipp16s** mMean, int height, int width, const Ipp32f* pVal, Ipp32f*
   pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVar_32f_D2(const Ipp32f* pSrc, const Ipp32f*
   pMean, int height, int step, int width, const Ipp32f* pVal, Ipp32f*
   pResult);
IppStatus ippsLogGaussMixture_IdVar_32f_D2L(const Ipp32f* pSrc, const Ipp32f**
   mMean, int height, int width, const Ipp32f* pVal, Ipp32f* pResult);
IppStatus ippsLogGaussMixture_IdVar_64f_D2(const Ipp64f* pSrc, const Ipp64f*
   pMean,  int height, int step, int width, const Ipp64f* pVal, Ipp64f*
   pResult);
IppStatus ippsLogGaussMixture_IdVar_64f_D2L(const Ipp64f* pSrc, const Ipp64f**
   mMean, int height, int width, const Ipp64f* pVal, Ipp64f* pResult);
```

Arguments

pSrc	Pointer to the input vector [*width*].
pMean	Pointer to the mean vector [*height*step*].
pVar	Pointer to the variance vector [*height*step*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].

height	Number of Gaussian mixture components.
step	Row step in mean and variance vectors (in *pMean* elements).
width	Length of the mean and variance matrix rows; also length of the *pSrc* vector.
pVal	Pointer to the vector of weight constants [*height*].
pResult	Pointer to the output mixture value.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function ippsLogGaussMixture is declared in the ippsr.h file. This function calculates observation probability for the Gaussian mixture and returns it in *pResult*.

For functions that do not have the IdVar suffix in their names, the covariance matrix is assumed to be inverse diagonal.

For these functions that also have the additional D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2,$$

for $i = 0,\ldots height - 1$, and

$$pResult[0] = \ln \sum_{i=0}^{height-1} e^{V[i]} .$$

For these functions with the D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 ,$$

$i = 0,\ldots height - 1$.

For functions that have the IdVar suffix in their names, the covariance matrix is assumed to be identity.

For these functions that have the additional D2 suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[j] - pMean[i \cdot step + j])^2 \quad ,$$

for $i = 0,\ldots height - 1$, and

$$pResult[0] = \ln \sum_{i=0}^{height-1} e^{V[i]}$$

For these functions with the additional D2L suffix,

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[j] - mMean[i][j])^2 \quad , \quad i = 0,\ldots height - 1.$$

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Functions with Low suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pMean*, *mMean*, *pVar*, *mVar*, *pVal*, or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* or *height* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

LogGaussMixtureSelect

Calculates the likelihood probability for the Gaussian mixture using Gaussian selection.

Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2(const Ipp16s* pSrc, const
   Ipp16s* pMean, const Ipp16s* pVar, int step, int width, const Ipp32s* pVal,
   const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f none,
   int scaleFactor);

IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2L(const Ipp16s** mSrc,
   const Ipp16s** mMean, const Ipp16s** mVar, int width, const Ipp32s* pVal,
   const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f none,
   int scaleFactor);

IppStatus ippsLogGaussMixture_Select_32f_D2(const Ipp32f* pSrc, const Ipp32f*
   pMean, const Ipp32f* pVar, int step, int width, const Ipp32f* pVal, const
   Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);

IppStatus ippsLogGaussMixture_Select_32f_D2L(const Ipp32f** mSrc, const
   Ipp32f** mMean, const Ipp32f** mVar, int width, const Ipp32f* pVal, const
   Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);
```

Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2(const Ipp16s* pSrc,
   const Ipp16s* pMean, int step, int width, const Ipp32s* pVal, const Ipp8u*
   pSign, int height, Ipp32s* pResult, int frames, Ipp32f none, int
   scaleFactor);

IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2L(const Ipp16s**
   mSrc, const Ipp16s** mMean, int width, const Ipp32s* pVal, const Ipp8u*
   pSign, int height, Ipp32s* pResult, int frames, Ipp32f none, int
   scaleFactor);

IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2(const Ipp32f* pSrc, const
   Ipp32f* pMean, int step, int width, const Ipp32f* pVal, const Ipp8u* pSign,
   int height, Ipp32f* pResult, int frames, Ipp32f none);

IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2L(const Ipp32f** mSrc, const
   Ipp32f** mMean, int width, const Ipp32f* pVal, const Ipp8u* pSign, int
   height, Ipp32f* pResult, int frames, Ipp32f none);
```

Arguments

pSrc	Pointer to the input vector [*frames***step*].
pMean	Pointer to the mean vector [*height***step*].
pVar	Pointer to the variance vector [*height***step*].
mSrc	Pointer to the input matrix [*frames*][*width*].
mMean	Pointer to the mean matrix [*height*][*width*].
mVar	Pointer to the variance matrix [*height*][*width*].
pVal	Pointer to the vector of weight constants [*height*].
pSign	Pointer to the vector of Gaussian calculation signs [*frames**(*height*+7)/8] (in bytes).
step	Row step in mean, variance and input vectors (in *pMean* elements).
width	Length of the mean, variance and input matrix rows.
height	Number of Gaussian mixture components.
pResult	Pointer to the vector of output mixture values [*frames*].
frames	Number of input vectors; also the length of *pSign* row.
none	Result value if no Gaussian is calculated.
scaleFactor	Scaling factor for intermediate sums.

Discussion

The function `ippsLogGaussMixture_Select` is declared in the `ippsr.h` file. This function calculates observation probabilities for the Gaussian mixture and *frames* input vectors and returns them in *pResult* vector. Gaussians with corresponding signs equal to zero are not calculated. If no Gaussian is calculated for the input vector, *none* value is returned for this vector.

Let $s(i,k)$ be the k-th bit of the i-th row of *pSign*..

For functions that do not have the `IdVar` suffix in their names, the covariance matrix is assumed to be inverse diagonal.

For these functions that also have the additional `D2` suffix,

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot \left(pSrc[k \cdot step + j] - pMean[i \cdot step + j]\right)^2$$

for k = 0,... `frames` - 1, i = 0,... `height` - 1 and $s(i,k) \neq 0$,

and

$$p\,\mathrm{Re}\,sult[k] = \begin{cases} none, & \text{if } s(i,k) = 0 \text{ for } i = 0,\ldots height-1 \\ \ln \sum_{\substack{i=0 \\ s(i,k)\neq 0}}^{height-1} e^{V[i,k]}, & \text{otherwise} \end{cases}$$

For these functions with the `D2L` suffix,

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot \left(mSrc[k][j] - mMean[i][j]\right)^2$$

for k = 0,... `frames` - 1, i = 0,... `height` - 1 and $s(i,k) \neq 0$,
and the `pResult` vector is computed by the same formula as given above.

For functions that have the `IdVar` suffix in their names, the covariance matrix is assumed to be identity.

For these functions that also have the additional `D2` suffix,

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} \left(pSrc[k \cdot step + j] - pMean[i \cdot step + j]\right)^2$$

for k = 0,... `frames` - 1, i = 0,... `height` - 1 and $s(i,k) \neq 0$,
and the `pResult` vector is computed by the same formula as given above.

For these functions with the `D2L` suffix,

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} \left(mSrc[k][j] - mMean[i][j]\right)^2$$

for $k = 0,\ldots$ *frames* - 1, $i = 0,\ldots$ *height* - 1 and $s(i,k) \neq 0$,
and the *pResult* vector is computed by the same formula as given above.

Function flavors that perform integer scaling (distinguished by the presence of *scaleFactor* argument in their argument list) apply the multiplier $2^{-scaleFactor}$ to intermediate sums in the above formulae, rather than to output results.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pMean*, *pVar*, *mSrc*, *mMean*, *mVar*, *pSign*, *pVal* or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *width*, *height*, or *frames* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.
ippStsNoGaussian	Indicates a warning when no Gaussian is calculated for one of the input vectors.

BuildSignTable

Fills sign table for Gaussian mixture calculation.

```
IppStatus ippsBuildSignTable_8u1u(const Ipp32s* pIndx, int num, const Ipp8u**
   mShortlist, int clust, int width, int shift, Ipp8u* pSign, int frames, int
   comps);
IppStatus ippsBuildSignTable_Var_8u1u(const Ipp32s* pIndx, const int* pNum,
   const Ipp8u** mShortlist, int clust, int width, int shift, Ipp8u* pSign, int
   frames, int comps);
```

Arguments

pIndx	Pointer to the cluster indexes vector ([*frames***num*] or sum of *pNum* elements).
num	Number of clusters for each input vector.
pNum	Number of clusters vector [*frames*].

mShortlist	Pointer to the shortlist matrix [*clust***width*] (in bytes).
clust	Number of rows in shortlist vector (equal to the codebook size).
width	Row length in shortlist matrix in bytes.
shift	First element displacement in shortlist vector rows (in bits).
pSign	Pointer to the output vector of Gaussian calculation signs [*frames**(*comps*+7)/8] (in bytes).
frames	Number of input vectors (rows in *pSign* vector).
comps	Number of Gaussian mixture components (bit columns in *pSign* vector).

Discussion

The functions ippsBuildSignTable are declared in the ippsr.h file. These functions build sign table for Gaussian mixture calculation using Gaussian selection technique. The sign table is used as the input argument for ippsLogGaussMixture_Select functions. The input vectors could activate one or several codebook clusters. Shortlist table rows correspond to clusters, and its columns – to Gaussians. Row substring of length *clust* indicates if the cluster is active for a mixture component.

Let $l(i) = i \cdot num$ for ippsBuildSignTable function, and

$l(i) = \sum_{j=0}^{i-1} pNum[j]$ for ippsBuildSignTable_Var function,

for $i = 0,\ldots frames - 1$.

Let also c_k be the bit substring of the k-th row of *mShortlist*, consisting of bits *shift*,...*shift*+*comps*-1 , for $k = 0,\ldots clust - 1$.

Then elements $l(i),\ldots l(i+1)-1$ of the *pIndx* vector are active cluster numbers for i-th input vector. The i-th row of *pSign* is set to the logical sum of substrings

$c_{l(i)},\ldots c_{l(i+1)-1}$, $i = 0,\ldots frames - 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pIndx*, *pSign*, *mShortlist* or *pNum* pointer is NULL.
ippStsSizeErr	Indicates an error when *clust*, *width*, *frames*, *comps*, *num*, or one of *pNum* vector elements is less than or equal to 0, or when *shift* is less than 0.
ippStsStrideErr	Indicates an error when *width* is less than (*shift*+*comps*+7)/8.
ippStsBadArgErr	Indicates an error when one of *pIndx* vector elements is less than 0, or greater than or equal to *clust*.

FillShortlist_Row

Fills row-wise shortlist table for Gaussian selection.

```
IppStatus ippsFillShortlist_Row_1u(const Ipp32s* pIndx, int height, int num,
   Ipp8u** mShortlist, int clust, int width, int shift);
IppStatus ippsFillShortlist_RowVar_1u(const Ipp32s* pIndx, const int* pNum,
   int height, Ipp8u** mShortlist, int clust, int width, int shift);
```

Arguments

pIndx	Pointer to the cluster indexes vector ([*num***clust*] or sum of *pNum* elements).
height	Number of Gaussian mixture components.
num	Number of Gaussian means for each cluster.
pNum	Number of Gaussians vector [*clust*].
mShortlist	Pointer to the shortlist matrix [*clust***width*] (in bytes).
clust	Number of rows in shortlist matrix (equal to the codebook size).

width	Row length in shortlist matrix (in bytes).
shift	First element displacement in shortlist matrix rows (in bits).

Discussion

The function ippsFillShortlist_Row is declared in the ippsr.h file. The function fills the shortlist table and provides that this table contains at least one Gaussian of the mixture for each cluster. This is done by quantizing of cluster centroids on codebook made of Gaussian mixture means. Quantization results are placed to *pIndx* vector.

Bits in the shortlist table *mShortlist* that correspond to one or more cluster centroids nearest to Gaussian mixture means are set to 1. The shortlist is the bit table with elements indicating whether the *i*-th Gaussian should be calculated if the input vector activates the *k*-th cluster of the codebook. The number of rows is equal to the codebook size, the number of columns is equal to the overall number of Gaussians in the model. Row step is chosen so as to start each row from the byte boundary.
The shortlist table should be zeroed after definition. Components of mixtures have sequential indexes. The *shift* argument is the index of the first mixture component in the model, and the *height* argument is the number of the components in the mixture.

Let $l(k) = k \cdot num$ for ippsFillShortlist_Row function,

$l(k) = \sum_{j=0}^{k-1} pNum[j]$ for ippsFillShortlist_RowVar function,

and $c(k,i)$ be equal to (*shift* + *i*)-th bit in the *k*-th row of *mShortlist*,
for $i = 0,\ldots$ *height* - 1, $k = 0,\ldots$ *clust* - 1.

Then *height* columns of the shortlist table are filled as:

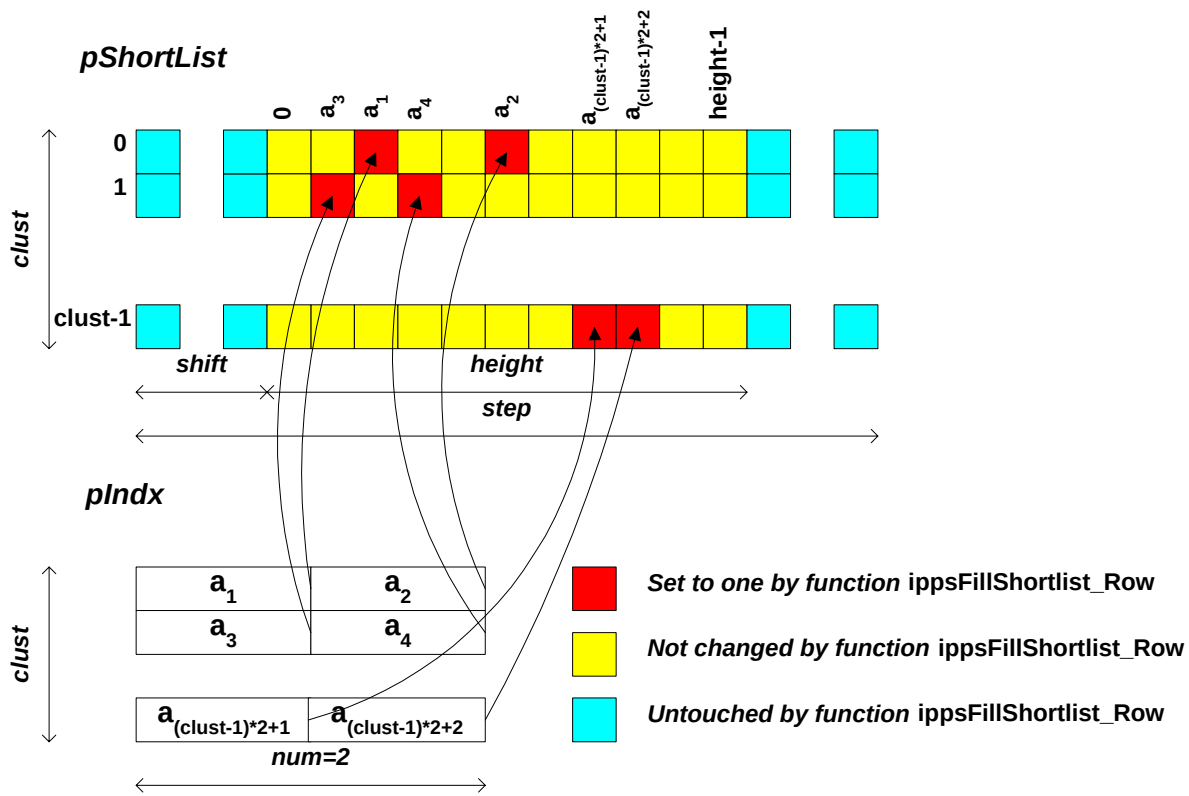
$c(k, pIndx[m]) = 1$, $m = l(k),\ldots l(k+1)-1$, $k = 0,\ldots$ *clust* - 1 .

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pIndx* or pShortlist pointer is NULL.

ippStsSizeErr	Indicates an error when *clust*, *height*, *width*, *num* or one of *pNum* vector elements is less than or equal to 0, or when *shift* is less than 0.
ippStsStrideErr	Indicates an error when *width* is less than (*shift*+*height*+7)/8.
ippStsBadArgErr	Indicates an error when one of *pIndx* vector elements is less than 0, or greater than or equal to *height*.

Figure 8-3 Execution of ippsFillShortlist_Row function for num=2



FillShortlist_Column

Fills column-wise shortlist table for Gaussian selection.

```
IppStatus ippsFillShortlist_Column_1u(const Ipp32s* pIndx, int num, Ipp8u**
   mShortlist, int clust, int width, int shift, int height);
IppStatus ippsFillShortlist_ColumnVar_1u(const Ipp32s* pIndx, const int* pNum,
   Ipp8u** mShortlist, int clust, int width, int shift, int height);
```

Arguments

pIndx	Pointer to the cluster indexes vector ([*num***height*] or sum of *pNum* elements).
height	Number of Gaussian mixture components.
num	Number of clusters for each Gaussian mean.
pNum	Number of clusters vector [*height*].
mShortlist	Pointer to the shortlist matrix [*clust***width*] (in bytes).
clust	Number of rows in shortlist matrix (equal to the codebook size).
width	Row length in shortlist matrix (in bytes).
shift	First element displacement in shortlist matrix rows (in bits).

Discussion

The function ippsFillShortlist_Column is declared in the ippsr.h file. This function fills the part of the shortlist table corresponding to the Gaussian mixture.

The shortlist is the bit table with elements indicating whether the *i*-th Gaussian should be calculated if the input vector activates the *k*-th cluster of the codebook. The number of rows is equal to the codebook size, the number of columns is equal to the overall number of Gaussians in the model. Row step is chosen so as to start each row from the byte boundary.

The shortlist table should be zeroed after definition. Components of mixtures have sequential indexes. The *shift* argument is the index of the first mixture component in the model, and the *height* argument is the number of the components in the mixture.

After means of Gaussian mixture are quantized by one of ippsVQSingle_Sort, VQSingle_Thresh functions, quantization results can be used to fill the shortlist table. Bits corresponding to active clusters for mixture components are set to 1.

Let $l(i) = i \cdot num$ for ippsFillShortlist_Column function,

$$l(i) = \sum_{j=0}^{i-1} pNum[j]$$ for ippsFillShortlist_ColumnVar function,

and $c(k,i)$ be equal to ($shift + i$)-th bit in the k-th row of *mShortlist*, for $i = 0,\ldots height - 1$, $k = 0,\ldots clust - 1$.

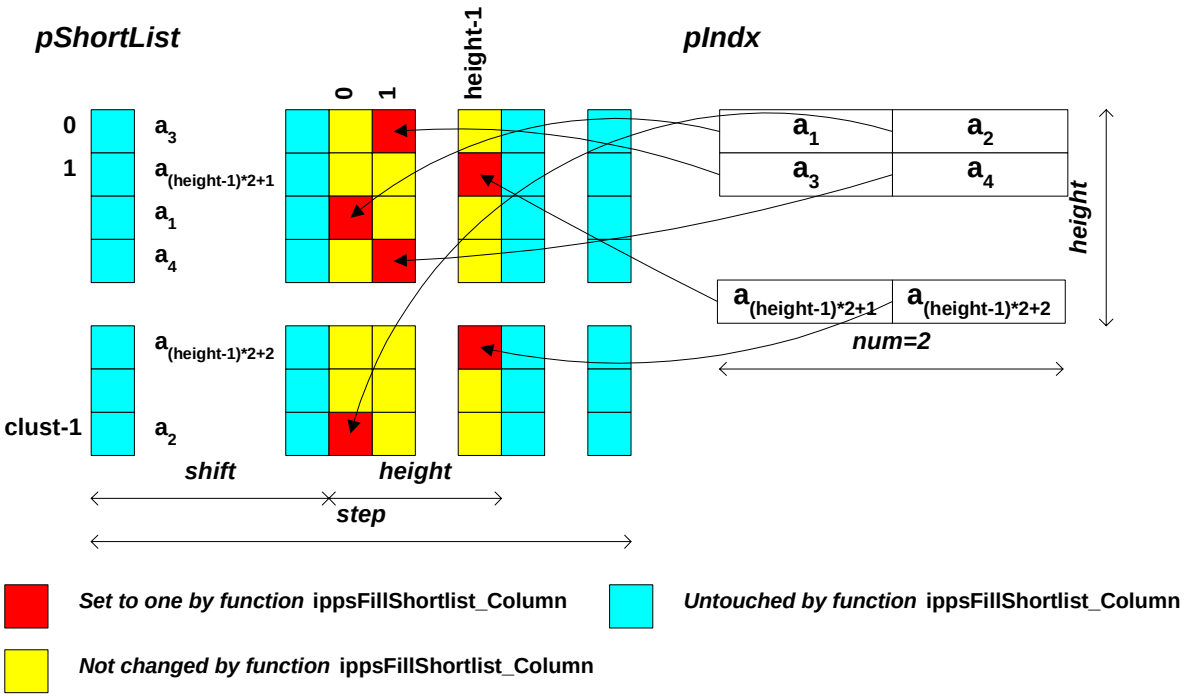
Then *height* columns of the shortlist table are filled as:

$c(pIndx[m], i) = 1$, $m = l(i),\ldots l(i+1)-1$, $i = 0,\ldots height - 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pIndx* or pShortlist pointer is NULL.
ippStsSizeErr	Indicates an error when *clust*, *height*, *width*, *num* or one of *pNum* vector elements is less than or equal to 0, or when *shift* is less than 0.
ippStsStrideErr	Indicates an error when width is less than (shift+*height*+7)/8.
ippStsBadArgErr	Indicates an error when one of *pIndx* vector elements is less than 0, or greater than or equal to *clust*.

Figure 8-4 Execution of ippsFillShortlist_Column function for num=2



DTW

Computes the distance between observation and reference vector sequences using Dynamic Time Warping algorithm.

```
IppStatus ippsDTW_L2_8u32s_D2Sfs(const Ipp8u* pSrc1, int height1, const Ipp8u*
   pSrc2, int height2, int width, int step, Ipp32s* pDist, int delta, Ipp32s
   beam, int scaleFactor);
IppStatus ippsDTW_L2_8u32s_D2LSfs(const Ipp8u** mSrc1, int height1, const
   Ipp8u** mSrc2, int height2, int width, Ipp32s* pDist, int delta, Ipp32s
   beam, int scaleFactor);
```

```
IppStatus ippsDTW_L2Low_16s32s_D2Sfs(const Ipp16s* pSrc1, int height1, const
   Ipp16s* pSrc2, int height2, int width, int step, Ipp32s* pDist, int delta,
   Ipp32s beam, int scaleFactor);
IppStatus ippsDTW_L2Low_16s32s_D2LSfs(const Ipp16s** mSrc1, int height1, const
   Ipp16s** mSrc2, int height2, int width, Ipp32s* pDist, int delta, Ipp32s
   beam, int scaleFactor);
IppStatus ippsDTW_L2_32f_D2(const Ipp32f* pSrc1, int height1, const Ipp32f*
   pSrc2, int height2, int width, int step, Ipp32f* pDist, int delta, Ipp32f
   beam);
IppStatus ippsDTW_L2_32f_D2L(const Ipp32f** mSrc1, int height1, const Ipp32f**
   mSrc2, int height2, int width, Ipp32f* pDist, int delta, Ipp32f beam);
```

Arguments

pSrc1	Pointer to the first input (observation) vector **x** [*height1*step*].
pSrc2	Pointer to the second input (reference) vector **y** [*height2*step*].
mSrc1	Pointer to the first input (observation) matrix **x** [*height1*][*width*].
mSrc2	Pointer to the second input (reference) matrix **y** [*height2*][*width*].
height1	Number of rows in the first input matrix (*N1)*.
height2	Number of rows in the second input matrix (*N2*).
width	Length of the input matrices row (*M*).
step	Row step in *pSrc1* and *pSrc2*.
pDist	Pointer to the distance value.
beam	Beam value, used if positive.
delta	Endpoint constraint value.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsDTW is declared in the ippsr.h file. This function calculates the distance between two vector sequences using the Dynamic Type Warping (DTW) principle. Computations are performed as follows:

$$pDist[0] = \min_{W} \sum_{i=0}^{N1-1} |x_i - y_{W(i)}| \ ,$$

when $0 \le W(0) \le \delta$, $0 \le W(i) - W(i-1) \le 2$, $i = 1, ..N1-1$, $N2 - \delta \le W(N1-1) \le N2-1$.

To compute distance between vectors *a* and *b*, the function uses L2-norm as given by:

$$|a-b| = \sqrt{\sum_{j=0}^{M-1} (a(j) - b(j))^2}$$

If the *beam* value is positive, paths *V* that are outside the beam, i.e., satisfy the condition

$$\sum_{i=0}^{k} |x_i - y_{V(i)}| > \min_{W} \sum_{i=0}^{k} |x_i - y_{W(i)}| - beam \ , \ k = 0, \ ..N1-1$$

are pruned away.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, *mSrc1*, *mSrc2*, or *pDist* pointer is NULL.
ippStsSizeErr	Indicates an error when *height1*, *height2*, or *width* is less than or equal to 0, or *delta* is less than 0 or greater than *height2*.
ippStsStrideErr	Indicates an error when *step* is less than *width*.
ippStsNoOperation	Indicates that there are now admissible paths for *height1*, *height2* and *delta* values.

Model Estimation

This section describes functions that are needed to estimate the parameters of the acoustic and language models.

MeanColumn

Computes the mean values for the column elements.

```
IppStatus ippsMeanColumn_16s_D2(const Ipp16s* pSrc, int height, int step,
   Ipp16s* pDstMean, int width);
IppStatus ippsMeanColumn_16s_D2L(const Ipp16s** mSrc, int height, Ipp16s*
   pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
   Ipp32f* pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
   pDstMean, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*.
step	Row step in the input vector (measured in *pSrc* elements).
pDstMean	Pointer to the output mean vector [*width*].
width	Number of columns in the input matrix *mSrc,* and also the length of the output mean vector *pDstMean*.

Discussion

The function `ippsMeanColumn` is declared in the `ippsr.h` file. This function calculates the mean values for the column elements of the input matrix as follows:

For functions with the D2 suffix,

$$pDstMean[j] = \frac{1}{height} \sum_{i=1}^{height-1} pSrc[i \cdot step + j], \quad 0 \le j < width$$

For functions with the D2L suffix,

$$pDstMean[j] = \frac{1}{height} \sum_{i=1}^{height-1} mSrc[i][j], \quad 0 \le j < width$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, or *pDstMean* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

VarColumn

Calculates the variances for the column elements.

```
IppStatus ippsVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int step,
   Ipp16s* pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height, Ipp16s*
   pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
   Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);
IppStatus ippsVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
   pSrcMean, Ipp32f* pDstVar, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].

mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc*.
step	Row step in the input vector *pSrc*.
pSrcMean	Pointer to the input mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the input mean vector *pSrcMean* and the output variance vector *pDstVar*.
scaleFactor	Scale factor, refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsVarColumn is declared in the ippsr.h file. This function calculates the variances for the column elements of the matrix as follows:
For functions with the D2 suffix,

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} pSrc[i \cdot step + j]^2}{height - 1},$$

$0 \le j < width$.

For functions with the D2L suffix

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} mSrc[i][j]^2}{height - 1},$$

$0 \le j < width$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pSrcMean*, or *pDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* is less than or equal to 0 or *height* is less than or equal to 1.

ippStsStrideErr	Indicates an error when *step* is less than *width*.

MeanVarColumn

Calculates the means and variances for the column elements of a matrix.

```
IppStatus ippsMeanVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int
   step, Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height,
   Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int height,
   int step, Ipp16s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
   Ipp16s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
   Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsMeanVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
   pDstMean, Ipp32f* pDstVar, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
height	Number of rows in the input matrix *mSrc* .
step	Row step in the input vector *pSrc* .
pDstMean	Pointer to the output mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix *mSrc*, and also the length of the output mean vector *pDstMean* and variance vector *pDstVar*.

scaleFactor	Scale factor, refer to "Integer Scaling" in Chapter 2. *scaleFactor* argument (and integer scaling) is used for *pDstVar* only, while *pDstMean* elements are not scaled.

Discussion

The function ippsMeanVarColumn is declared in the ippsr.h file. This function calculates both the means and the variances for the column elements of the input matrix. See functions ippsMeanColumn and ippsVarColumn for calculation details.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *mSrc*, *pDstMean*, or *pDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *width* is less than or equal to 0 or *height* is less than or equal to 1.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

WeightedMeanColumn

Computes the weighted mean values for the column elements.

```
IppStatus ippsWeightedMeanColumn_32f_D2(const Ipp32f* pSrc, int step, const
   Ipp32f* pWgt, int height, Ipp32f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
   pWgt, int height, Ipp32f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_64f_D2(const Ipp64f* pSrc, int step, const
   Ipp64f* pWgt, int height, Ipp64f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
   pWgt, int height, Ipp64f* pDstMean, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
pWgt	Pointer to the weight vector [*height*].
height	Number of rows in the input matrix.
step	Row step in the input vector (measured in *pSrc* elements).
pDstMean	Pointer to the output mean vector [*width*].
width	Number of columns in the input matrix, and also the length of the output mean vector *pDstMean*.

Discussion

The function ippsWeightedMeanColumn is declared in the ippsr.h file. This function calculates the weighted mean values for the column elements of the input matrix as follows:
For functions with the D2 suffix,

$$pDstMean[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot pSrc[i \cdot step + j], 0 \le j < width .$$

For functions with the D2L suffix,

$$pDstMean[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot mSrc[i][j], 0 \le j < width .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pWgt*, or *pDstMean* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

WeightedVarColumn

Computes the weighted variance values for the column elements.

```
IppStatus ippsWeightedVarColumn_32f_D2(const Ipp32f* pSrc, int step, const
   Ipp32f* pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar, int
   width);
IppStatus ippsWeightedVarColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
   pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);
IppStatus ippsWeightedVarColumn_64f_D2(const Ipp64f* pSrc, int step, const
   Ipp64f* pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar, int
   width);
IppStatus ippsWeightedVarColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
   pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
pWgt	Pointer to the weight vector [*height*].
height	Number of rows in the input matrix.
step	Row step in the input vector (measured in *pSrc* elements).
pSrcMean	Pointer to the input mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix, and also the length of the input mean vector *pSrcMean* and the output variance vector *pDstVar*.

Discussion

The function ippsWeightedVarColumn is declared in the ippsr.h file. This function calculates the weighted variance values for the column elements of the input matrix as follows:
For functions with the D2 suffix,

$$pDstVar[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot pSrc[i \cdot step + j]^2 - pSrcMean[j]^2 \ , 0 \le j < width \ .$$

For functions with the D2L suffix,

$$pDstVar[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot mSrc[i][j]^2 - pSrcMean[j]^2 \ , 0 \le j < width \ .$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pWgt*, *pSrcMean*, or *pDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

WeightedMeanVarColumn

Computes weighted mean and variance values for the column elements.

```
IppStatus ippsWeightedMeanVarColumn_32f_D2(const Ipp32f* pSrc, int step, const
   Ipp32f* pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
   pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_64f_D2(const Ipp64f* pSrc, int step, const
   Ipp64f* pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int width);
```

```
IppStatus ippsWeightedMeanVarColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
   pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int width);
```

Arguments

pSrc	Pointer to the input vector [*height*step*].
mSrc	Pointer to the input matrix [*height*][*width*].
pWgt	Pointer to the weight vector [*height*].
height	Number of rows in the input matrix.
step	Row step in the input vector (measured in *pSrc* elements).
pDstMean	Pointer to the output mean vector [*width*].
pDstVar	Pointer to the output variance vector [*width*].
width	Number of columns in the input matrix, and also the length of the output mean vector *pSrcMean* and the output variance vector *pDstVar*.

Discussion

The function `ippsWeightedMeanVarColumn` is declared in the `ippsr.h` file. This function calculates both weighted means and weighted variances for the column elements of the input matrix. See functions `ippsWeightedMeanColumn` and `ippsWeightedVarColumn` for calculation details.

The code example below shows how you can use the function `ippsWeightedMeanVarColumn`.

Example 8-2 Weights, Means and Variances EM Re-Estimation

```
/* Input:  int height;              // mixture components number
           int width;               // observation space dimension
          int step;               // row step for mean, var and obs (step>=width)
           int num;                 // observations number
           int step1;               // row step for gamma (step1>=num)
           float obs [num*step}     // observation vectors
   Update: float weight[height]     // Gaussian weights
           float mean[height*step]  // Gaussian mean vectors
           float var [height*step]  // Gaussian variance vectors
   Output: float result;            // probability logarithms sum */
{
   float gamma [height*step1] // gamma matrix
   float gammaT [height]      // gamma sums vector
   int k; float sum, sumGamma;
   /* adjust determinants for probability calculation */
   ippsLn_32f_I(weight,height);
   for (k=0; k<height; k++) {
      ippsSumLn_32f(var+k*step,width,&sum);
      weight[k]+=0.5f*(sum-width*log(2.0*3.1415926));
   }
   /* invert variances for probability calculation */
   ippsDivCRev_32f_I(var,height*step);
   /* logarithm of weighted Gaussian probabilities */
   ippsLogGauss_32f_D2(obs,step,mean,var,width,gamma,num,weight[0]);
   ippsCopy_32f(gamma,probs,num);
   for (k=1; k<height; k++) {
      ippsLogGauss_32f_D2(obs,step,mean+k*step,var+k*step,width,
                     gamma+k*step1,num,weight[k]);
      ippsLogAdd_32f(gamma+k*step1,probs,num,ippAlgHintNone);
   }
  ippsSum_32f(probs,num,&result,ippAlgHintNone);
   /* gamma matrix and sum calculation */
   ippsExp_32f_I(gamma,height*step1);
```

Example 8-2 Weights, Means and Variances EM Re-Estimation

```
        ippsSumRow_32f_D2(gamma,height,step1,gammaT,num);
       ippsSum_32f(gammaT,height,&sumGamma,ippAlgHintNone);
       /* weights update */
       ippsDivC_32f(gammaT,sumGamma,weight,height);
       /* means and variances update */
       for (k=0; k<height; k++) {
          ippsDivC_32f_I(gammaT[k],gamma+k*step1,num);
          ippsWeightedMeanVarColumn_32f_D2(obs,step,gamma+k*step1,num,
                            mean+k*step,var+k*step,width);
       }
}
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *mSrc*, *pWgt*, *pDstMean*, or *pDstVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

NormalizeColumn

Normalizes the matrix columns given the column means and variances.

```
IppStatus ippsNormalizeColumn_16s_D2Sfs(Ipp16s* pSrcDst, int step, int height,
   const Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);
IppStatus ippsNormalizeColumn_16s_D2LSfs(Ipp16s** mSrcDst, int height, const
   Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);
IppStatus ippsNormalizeColumn_32f_D2(Ipp32f* pSrcDst, int step, int height,
   const Ipp32f* pMean, const Ipp32f* pVar, int width);
IppStatus ippsNormalizeColumn_32f_D2L(Ipp32f** mSrcDst, int height, const
   Ipp32f* pMean, const Ipp32f* pVar, int width);
```

Arguments

pSrcDst	Pointer to the input and output vector [*height*step*].
mSrcDst	Pointer to the input and output matrix [*height*][*width*].
pMean	Pointer to the column mean vector [*width*].
pVar	Pointer to the column variance vector [*width*].
width	Length of the mean and variance vectors.
step	Row step in the input and output vector *pSrcDst*.
height	Number of rows in the input and output matrix *mSrcDst*.

Discussion

The function ippsNormalizeColumn is declared in the ippsr.h file. This function normalizes the columns of the matrix *mSrcDst* as follows:

For functions with the D2 suffix,

$$pSrcDst[i \cdot step + j] = (pSrcDst[i \cdot step + j] - pMean[j]) \cdot pVar[j],$$

$0 \le i < height$, $0 \le j < width$.

For functions with the D2L suffix,

$$mSrcDst[i][j] = (mSrcDst[i][j] - pMean[j]) \cdot pVar[j],$$

$0 \le i < height$, $0 \le j < width$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDst*, *mSrcDst*, *pMean*, or *pVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* or *width* is less than or equal to 0.
ippStsStrideErr	Indicates an error when *step* is less than *width*.

NormalizeInRange

Normalizes and scales input vector elements.

```
IppStatus ippsNormalizeInRange_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len,
   Ipp32f lowCut, Ipp32f highCut, Ipp8u range);
IppStatus ippsNormalizeInRange_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
   Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_16s_I(Ipp16s* pSrcDst, int len, Ipp32f lowCut,
   Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len,
   Ipp32f lowCut, Ipp32f highCut, Ipp8u range);
IppStatus ippsNormalizeInRange_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int
   len, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
   Ipp32f lowCut, Ipp32f highCut, Ipp32f range);
IppStatus ippsNormalizeInRange_32f_I(Ipp32f* pSrcDst, int len, Ipp32f lowCut,
   Ipp32f highCut, Ipp32f range);
IppStatus ippsNormalizeInRangeMinMax_16s8u(const Ipp16s* pSrc, Ipp8u* pDst,
   int len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp8u
   range);
IppStatus ippsNormalizeInRangeMinMax_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
   len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s
   range);
IppStatus ippsNormalizeInRangeMinMax_16s_I(Ipp16s* pSrcDst, int len, Ipp16s
   valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRangeMinMax_32f8u(const Ipp32f* pSrc, Ipp8u* pDst,
   int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp8u
   range);
IppStatus ippsNormalizeInRangeMinMax_32f16s(const Ipp32f* pSrc, Ipp16s* pDst,
   int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut,
   Ipp16s range);
IppStatus ippsNormalizeInRangeMinMax_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
   len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f
   range);
```

```
IppStatus ippsNormalizeInRangeMinMax_32f_I(Ipp32f* pSrcDst, int len, Ipp32f
   valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f range);
```

Arguments

pSrc	Pointer to the input array [*len*].
pSrcDst	Pointer to the input and output array (for the in-place operation) [*len*].
pDst	Pointer to the output array [*len*].
len	Number of elements in the input and output array.
lowCut	Lower cutoff value.
highCut	Higher cutoff value.
range	Upper bound of output data values (lower bound is 0).
valMin	Minimum value of input data.
valMax	Maximum value of input data.

Discussion

The function ippsNormalizeInRange is declared in the ippsr.h file. This function first normalizes the input vector elements to the range from 0 to 1 imposing lower and higher cutoffs ($0 \le lowCut < highCut \le 1$), and then scales them by the *range* value.

If we denote by x_k the input values (which are equal to *pSrc*[*k*] for not-in-place functions and to *pSrcDst*[*k*] for in-place functions), then the output values y_k (written to *pDst*[*k*] for not-in-place functions and to *pSrcDst*[*k*] for in-place functions) are calculated according to the following formula:

$$y_k = \max\left(0, \min\left(range, \frac{\frac{x_k - x_{min}}{x_{max} - x_{min}} - lowCut}{highCut - lowCut} \cdot range\right)\right), k = 0,\ldots len - 1,$$

where

$x_{min} = \min_{i=0,\ldots len-1} x_i$, $x_{max} = \max_{i=0,\ldots len-1} x_i$ for ippsNormalizeInRange function

and

$x_{min} = valMin$, $x_{max} = valMax$ for ippsNormalizeInRangeMinMax function.

This function can be used to form spectrogram data.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsBadArgErr	Indicates an error when the condition $0 \le lowCut < highCut \le 1$ is not met, or *range* is less than 0, or *valMin* is greater than *valMax*.
ippStsInvZero	Indicates a warning that $x_{min} = x_{max}$. All elements of the output vector are set to 0.

MeanVarAcc

Accumulates the estimates for the mean and variance re-estimation.

```
IppStatus ippsMeanVarAcc_32f(Ipp32f const* pSrc, Ipp32f const* pSrcMean,
   Ipp32f* pDstMeanAcc, Ipp32f* pDstVarAcc, int len, Ipp32f val);
IppStatus ippsMeanVarAcc_64f(Ipp64f const* pSrc, Ipp64f const* pSrcMean,
   Ipp64f* pDstMeanAcc, Ipp64f* pDstVarAcc, int len, Ipp64f val);
```

Arguments

pSrc	Pointer to the observation vector [*len*].
pSrcMean	Pointer to the old mean vector [*len*].
pDstMeanAcc	Pointer to the mean accumulator [*len*].
pDstVarAcc	Pointer to the variance accumulator [*len*].
len	Length of the observation vector.
val	Constant value in the re-estimation.

Discussion

The function ippsMeanVarAcc is declared in the ippsr.h file. This function accumulates the estimates for the mean and variance re-estimation in the forward-backward algorithm. The calculation is as follows:

$$pDstMeanAcc[i] = pDstMeanAcc[i] + val \cdot (pSrc[i] - pSrcMean[i]),$$

$$pDstVarAcc[i] = pDstVarAcc[i] + val \cdot (pSrc[i] - pSrcMean[i])^2,$$

$$0 \le i < len.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pSrcMean*, *pDstMeanAcc*, or *pDstVarAcc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

GaussianDist

Calculates the distance between two Gaussians.

```
IppStatus ippsGaussianDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
   const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f
   wgt1, Ipp32f det1, Ipp32f wgt2, Ipp32f det2);
IppStatus ippsGaussianDist_64f(const Ipp64f* pMean1, const Ipp64f* pVar1,
   const Ipp64f* pMean2, const Ipp64f* pVar2, int len, Ipp64f* pResult, Ipp64f
   wgt1, Ipp64f det1, Ipp64f wgt2, Ipp64f det2);
```

Arguments

pMean1	Pointer to the mean vector of the first Gaussian [*len*].
pVar1	Pointer to the variance vector of the first Gaussian [*len*].
pMean2	Pointer to the mean vector of the second Gaussian [*len*].
pVar2	Pointer to the variance vector of the second Gaussian [*len*].

len	Length of the mean and variance vectors.
pResult	Pointer to the distance value.
wgt1	Weight of the first Gaussian.
det1	Determinant of the first Gaussian (in the logarithmic representation).
wgt2	Weight of the second Gaussian.
det2	Determinant of the second Gaussian (in the logarithmic representation).

Discussion

The function ippsGaussianDist is declared in the ippsr.h file. This function calculates the distance between two Gaussians as follows:

$$pResult[0] = (wgt1 \cdot det1) + (wgt2 \cdot det2) - (wgt1 + wgt2) \cdot (len \cdot \ln(2\pi) - \ln(V)),$$

where

$$V = \prod_{i=0}^{len-1} \frac{W_1[i] + W_2[i] - (wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i])^2}{wgt1 + wgt2},$$

and

$$W_1[i] = wgt1 \cdot (pVar1[i] + pMean1[i]^2),$$

$$W_2[i] = wgt2 \cdot (pVar2[i] + pMean2[i]^2)$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, *pVar2*, or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

GaussianSplit

Splits a single Gaussian component into two with the same variance.

```
IppStatus ippsGaussianSplit_32f(Ipp32f* pMean1, Ipp32f* pVar1, Ipp32f* pMean2,
   Ipp32f* pVar2, int len, Ipp32f val);
IppStatus ippsGaussianSplit_64f(Ipp64f* pMean1, Ipp64f* pVar1, Ipp64f* pMean2,
   Ipp64f* pVar2, int len, Ipp64f val);
```

Arguments

pMean1	Pointer to the input Gaussian mean vector, also the mean vector of the first Gaussian after splitting [*len*].
pVar1	Pointer to the input Gaussian variance vector, also the variance vector of the first Gaussian after splitting [*len*].
pMean2	Pointer to the mean vector of the second Gaussian after splitting [*len*].
pVar2	Pointer to the variance vector of the second Gaussian after splitting [*len*].
len	Length of the mean and variance vectors.
val	Variance perturbation value.

Discussion

The function ippsGaussianSplit is declared in the ippsr.h file. This function splits the Gaussian component into two Gaussian mixture components as follows:

$$pMean1[i] = pMean1[i] + val \cdot \sqrt{pVar1[i]},$$

$$pMean2[i] = pMean1[i] - val \cdot \sqrt{pVar1[i]},$$

$$pVar2[i] = pVar1[i],$$

$$0 \le i < len.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, or *pVar2* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

GaussianMerge

Merges two Gaussian probability distribution functions.

```
IppStatus ippsGaussianMerge_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
   const Ipp32f* pMean2, const Ipp32f* pVar2, Ipp32f* pDstMean, Ipp32f*
   pDstVar, int len, Ipp32f* pDstDet, Ipp32f wgt1, Ipp32f wgt2);
IppStatus ippsGaussianMerge_64f(const Ipp64f* pMean1, const Ipp64f* pVar1,
   const Ipp64f* pMean2, const Ipp64f* pVar2, Ipp64f* pDstMean, Ipp64f*
   pDstVar, int len, Ipp64f* pDstDet, Ipp64f wgt1, Ipp64f wgt2);
```

Arguments

pMean1	Pointer to the mean vector of the first Gaussian [*len*].
pVar1	Pointer to the variance vector of the first Gaussian [*len*].
pMean2	Pointer to the mean vector of the second Gaussian [*len*].
pVar2	Pointer to the variance vector of the second Gaussian [*len*].
len	Length of the mean and variance vectors.
pDstMean	Pointer to the mean vector of the merged Gaussian [*len*].
pDstVar	Pointer to the variance vector of the merged Gaussian [*len*].
pDstDet	Pointer to the determinant of the merged Gaussian.
wgt1	Weight of the first Gaussian.
wgt2	Weight of the second Gaussian.

Discussion

The function ippsGaussianMerge is declared in the ippsr.h file. This function merges two Gaussian probability distribution functions as follows:

$$pDstMean[i] = \frac{wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i]}{wgt1 + wgt2},$$

$$pDstVar[i] = \frac{wgt1^2 \cdot (pVar1[i] + pMean1[i]^2) + wgt2^2 \cdot (pVar2[i] + pMean2[i]^2)}{wgt1 + wgt2} - pDstMean[i]^2,$$

$$pDstDet[0] = len \cdot \ln(2\pi) - \sum_{i=0}^{len-1} \ln pDstVar[i],$$

$0 \le i < len$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMean1*, *pMean2*, *pVar1*, *pVar2*, *pDstMean*, *pDstVar*, or *pDstDet* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Entropy

Calculates entropy of the input vector.

```
IppStatus ippsEntropy_32f(const Ipp32f* pSrc, int len, Ipp32f* pResult);
IppStatus ippsEntropy_16s32s_Sfs(const Ipp16s* pSrc, int srcShiftVal, int len,
   Ipp32s* pResult, int scaleFactor);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pResult	Pointer to the destination entropy value.

len	Length of the input vector.
srcShiftVal	Refer to "Integer Scaling" in Chapter 2.
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsEntropy is declared in the ippsr.h file. This function calculates entropy of the input vector as given by:

$$pResult[0] = \sum_{i=0}^{len-1} pSrc[i] \cdot \log_2 pSrc[i]$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnNegArg	Indicates a warning that some input vector elements are less than 0. Operation execution is not aborted. For floating-point operations the destination value is set to NaN, and for integer operations it is set to 0.

SinC

Calculates sine divided by its argument.

```
IppStatus ippsSinc_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinc_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinc_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinc_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSinc_64f_I(Ipp64f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the input vector [*len*].
pSrcDst	Pointer to the input and destination vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.

Discussion

The function ippsSinc is declared in the ippsr.h file. This function calculates destination vector elements according to the formula:

$pDst[k] = \frac{\sin(pSrc[k])}{pSrc[k]}$, $k = 0,...len - 1$.

Return Value

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

ExpNegSqr

Calculates exponential of the squared argument taken with the inverted sign.

```
IppStatus ippsExpNeqSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExpNeqSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNeqSqr_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNeqSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExpNeqSqr_64f_I(Ipp64f* pSrcDst, int len);
```

Arguments

pSrc	Pointer to the input vector [*len*].

pSrcDst	Pointer to the input and destination vector [*len*].
pDst	Pointer to the destination vector [*len*].
len	Length of the input and output vectors.

Discussion

The function ippsExpNeqSqr is declared in the ippsr.h file. This function calculates destination vector elements according to the following formula:

$pDst[k] = \exp(-pSrc[k]^2)$, $k = 0,\ldots len - 1$.

Return Value

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

BhatDist

Calculates the Bhattacharia distance between two Gaussians.

```
IppStatus ippsBhatDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
   Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult);
IppStatus ippsBhatDist_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
   Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult);
IppStatus ippsBhatDistSLog_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
   const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f
   sumLog1, Ipp32f sumLog2);
IppStatus ippsBhatDistSLog_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1,
   const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult, Ipp32f
   sumLog1, Ipp32f sumLog2);
```

Arguments

pMean1	Pointer to the first mean vector [*len*].
pVar1	Pointer to the first variance vector [*len*].
pMean2	Pointer to the second mean vector [*len*].
pVar2	Pointer to the second variance vector [*len*].
pResult	Pointer to the result.
len	Length of the input mean and variance vectors.
sumLog1	Sum of the first Gaussian variance in the logarithmic representation.
sumLog2	Sum of the second Gaussian variance in the logarithmic representation.

Discussion

The functions ippsBhatDist and ippsBhatDistSLog are declared in the ippsr.h file.

The function ippsBhatDist calculates the Bhattacharia distance between two Gaussians as follows:

$$pResult[0] = \frac{1}{4}\sum_{i=0}^{len-1}\frac{(pMean1[i]-pMean2[i])^2}{pVar1[i]+pVar2[i]} + \frac{1}{2}\sum_{i=0}^{len-1}\left(\ln\left(\frac{pVar1[i]+pVar2[i]}{2}\right)-\frac{\ln(pVar1[i])+\ln(pVar2[i])}{2}\right)$$

The function ippsBhatDistSLog calculates the Bhattacharia distance between two Gaussians as follows:

$$pResult[0] = \frac{1}{4}\sum_{i=0}^{len-1}\frac{(pMean1[i]-pMean2[i])^2}{pVar1[i]+pVar2[i]} + \frac{1}{2}\sum_{i=0}^{len-1}\ln\left(\frac{pVar1[i]+pVar2[i]}{2}\right)-\frac{sumLog1+sumLog2}{4}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMean1*, *pVar1*, *pMean2*, *pVar2*, or *pResult* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning that a zero value was detected in the input vector. The execution is not aborted. The result value is set to –Inf if there is no negative element in the vector.
ippStsLnNegArg	Indicates a warning that negative values were detected in the input vector. The execution is not aborted. The result value is set to NaN.

UpdateMean

Updates the mean vector in the EM training algorithm.

```
IppStatus ippsUpdateMean_32f(const Ipp32f* pMeanAcc, Ipp32f* pMean, int len,
   Ipp32f meanOcc);
IppStatus ippsUpdateMean_64f(const Ipp64f* pMeanAcc, Ipp64f* pMean, int len,
   Ipp64f meanOcc);
```

Arguments

pMeanAcc	Pointer to the mean accumulator [*len*].
pMean	Pointer to the mean vector [*len*].
len	Length of the mean vector.
meanOcc	Occupation sum of the Gaussian mixture.

Discussion

The function ippsUpdateMean is declared in the ippsr.h file. This function calculates the updated mean vector in the EM (Expectation-Maximization) training algorithm as follows:

$$pMean[i] = pMean[i] + \frac{pMeanAcc[i]}{meanOcc}, 0 \le i < len.$$

Note that if $meanOcc \le 0$, the mean vector *pMean* is not updated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMean* or *pMeanAcc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning that *meanOcc* is equal to 0.
ippStsNegOccErr	Indicates an error when *meanOcc* is less than 0.

UpdateVar

Updates the variance vector in the EM training algorithm.

```
IppStatus ippsUpdateVar_32f(const Ipp32f* pMeanAcc, const Ipp32f* pVarAcc,
   const Ipp32f* pVarFloor, Ipp32f* pVar, int len, Ipp32f meanOcc, Ipp32f
   varOcc);
IppStatus ippsUpdateVar_64f(const Ipp64f* pMeanAcc, const Ipp64f* pVarAcc,
   const Ipp64f* pVarFloor, Ipp64f* pVar, int len, Ipp64f meanOcc, Ipp64f
   varOcc);
```

Arguments

pMeanAcc	Pointer to the mean accumulator [*len*].
pVarAcc	Pointer to the variance accumulator [*len*].
pVarFloor	Pointer to the variance floor vector [*len*].
pVar	Pointer to the variance vector [*len*].

len	Length of the variance.
meanOcc	Occupation sum of the Gaussian mixture.
varOcc	Square occupation sum of the variance mixture.

Discussion

The function ippsUpdateVar is declared in the ippsr.h file. This function calculates the updated variance vector in the EM algorithm. The covariance matrix is assumed to be diagonal. The accumulators are calculated from the training data. The update equation is as follows:

$$pVar[i] = \max\left[varFloor[i],\ \frac{varAcc[i]}{varOcc} - \left(\frac{meanAcc[i]}{meanOcc}\right)^2\right],\ 0 \le i < len.$$

Note that if $meanOcc \le 0$ or $varOcc \le 0$, the variance vector *pVar* is not updated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMeanAcc*, *pVarAcc*, *pVarFloor*, or *pVar* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning that *meanOcc* or *varOcc* is equal to 0.
ippStsNegOccErr	Indicates an error when *meanOcc* and *varOcc* are less than 0.
ippStsResFloor	Indicates a warning that all variances are floored.

UpdateWeight

Updates the weight values of Gaussian mixtures in the EM training algorithm.

```
IppStatus ippsUpdateWeight_32f(const Ipp32f* pWgtAcc, Ipp32f* pWgt, int len,
   Ipp32f* pWgtSum, Ipp32f wgtOcc, Ipp32f wgtThresh);
```

```
IppStatus ippsUpdateWeight_64f(const Ipp64f* pWgtAcc, Ipp64f* pWgt, int len,
   Ipp64f* pWgtSum, Ipp64f wgtOcc, Ipp64f wgtThresh);
```

Arguments

pWgtAcc	Pointer to the weight accumulator [*len*].
pWgt	Pointer to the weight vector [*len*].
len	Number of Gaussian mixture components.
pWgtSum	Pointer to the output sum of weight values.
wgtOcc	Nominator of the weight update equation.
wgtThresh	Threshold for the weight values.

Discussion

The function `ippsUpdateWeight` is declared in the `ippsr.h` file. This function calculates the updated weight values for a Gaussian mixture. The accumulators are calculated from the training data. The update equation is as follows:

$$pWgt[i] = \max\left(\frac{pWgtAcc[i]}{wgtOcc}, wgtThresh\right), 0 \le i < len$$

$$pWgtSum[0] = \sum_{i=0}^{len-1} pWgt[i] .$$

Note that if $wgtOcc \le 0$, the weight vector *pWgt* is not updated.

This function can also be used to update the HMM transition matrix.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pWgt*, *pWgtAcc*, or *pWgtSum* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsZeroOcc`	Indicates a warning that *wgtOcc* is equal to 0.
`ippStsNegOccErr`	Indicates an error when *wgtOcc* is less than 0.
`ippStsResFloor`	Indicates a warning that all weights are floored.

UpdateGConst

Updates the fixed constant in the Gaussian output probability density function.

```
IppStatus ippsUpdateGConst_32f(const Ipp32f* pVar, int len, Ipp32f* pDet);
IppStatus ippsUpdateGConst_64f(const Ipp64f* pVar, int len, Ipp64f* pDet);
IppStatus ippsUpdateGConst_DirectVar_32f(const Ipp32f* pVar, int len,
   Ipp32f* pDet);
IppStatus ippsUpdateGConst_DirectVar_64f(const Ipp64f* pVar, int len,
   Ipp64f* pDet);
```

Arguments

pVar	Pointer to the variance vector [*len*].
len	Dimension of the variance vector.
pDet	Pointer to the result value.

Discussion

The function `ippsUpdateGConst` is declared in the `ippsr.h` file. This function calculates the fixed variance constant.
For functions without the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be inverse diagonal:

$$pDet[0] = len \cdot \ln 2\pi - \sum_{i=o}^{len-1} \ln(pVar[i])$$

For functions with the `DirectVar` suffix, the Gaussian covariance matrix is assumed to be diagonal:

$$pDet[0] = len \cdot \ln 2\pi + \sum_{i=o}^{len-1} \ln(pVar[i])$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pVar* or *pDet* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to –Inf if there are no negative elements in the vector.
ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to NaN.

OutProbPreCalc

Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.

```
IppStatus ippsOutProbPreCalc_32s(const Ipp32s* pWeight, const Ipp32s* pSrc,
   Ipp32s* pDst, int len);
IppStatus ippsOutProbPreCalc_32s_I(const Ipp32s* pWeight, Ipp32s* pSrcDst,
   int len);
IppStatus ippsOutProbPreCalc_32f(const Ipp32f* pWeight, const Ipp32f* pSrc,
   Ipp32f* pDst, int len);
IppStatus ippsOutProbPreCalc_64f(const Ipp64f* pWeight, const Ipp64f* pSrc,
   Ipp64f* pDst, int len);
IppStatus ippsOutProbPreCalc_32f_I(const Ipp32f* pWeight, Ipp32f* pSrcDst,
   int len);
IppStatus ippsOutProbPreCalc_64f_I(const Ipp64f* pWeight, Ipp64f* pSrcDst,
   int len);
```

Arguments

pWeight	Pointer to the Gaussian mixture weight vector [*len*].

pSrc	Pointer to the input vector calculated from ippsUpdateGConst function.
pSrcDst	Pointer to the input and output vector calculated from ippsUpdateGConst function.
pDst	Pointer to the result vector [*len*].
len	Number of mixtures in the HMM state.

Discussion

The function ippsOutProbPreCalc is declared in the ippsr.h file. This function pre-calculates the part of the Gaussian mixture output probability that is irrelevant to the observation vectors.

For the function ippsOutProbPreCalc,

$pDst[i] = pWeight[i] - 0.5 \cdot pSrc[i], 0 \le i < len.$

For the function ippsOutProbPreCalc_I,

$pSrcDst[i] = pWeight[i] - 0.5 \cdot pSrcDst[i], 0 \le i < len.$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pWeight*, *pDet*, or *pVal* pointer is NULL.
ippStsSizeErr	Indicates an error when the *len* is less than or equal to zero.

DcsClustLAccumulate

Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.

```
IppStatus ippsDcsClustLAccumulate_32f(const Ipp32f* pMean, const Ipp32f* pVar,
   Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);
```

```
IppStatus ippsDcsClustLAccumulate_64f(const Ipp64f* pMean, const Ipp64f* pVar,
   Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_32f(const Ipp32f* pMean, const
   Ipp32f* pVar, Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_64f(const Ipp64f* pMean, const
   Ipp64f* pVar, Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);
```

Arguments

pMean	Pointer to the mean vector of an HMM state in the cluster [*len*].
pVar	Pointer to the variance vector of an HMM state in the cluster[*len*].
pDstSum	Pointer to the summation part of the accumulator[*len*].
pDstSqr	Pointer to the square sum part of the accumulator[*len*].
len	Length of the mean and variance vectors.
occ	Occupation counts of the HMM state

Discussion

The function ippsDcsClustLAccumulate is declared in the ippsr.h file. This function updates the accumulators in the decision-tree clustering algorithm. The accumulators are used to calculate the likelihood of an HMM state cluster.

The accumulation equations are as follows:

For functions without the DirectVar suffix, the Gaussian covariance matrix is assumed to be inverse diagonal, and

$$pDstSum[i] = pMean[i] \cdot occ ,$$

$$pDstSqr[i] = \left[\frac{1}{pVar[i]} + (pMean[i])^2\right] \cdot occ \text{ , for } 0 \le i < len.$$

For functions with the DirectVar suffix, the Gaussian covariance matrix is assumed to be diagonal, and

$$pDstSum[i] = pMean[i] \cdot occ ,$$

$$pDstSqr[i] = [pVar[i] + (pMean[i])^2] \cdot occ \text{ , for } 0 \le i < len.$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pMean*, *pVar*, *pDstSum*, or *pDstSqr* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

DcsClustLCompute

Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.

```
IppStatus ippsDcsClustLCompute_64f(const Ipp64f* pSrcSum, const Ipp64f*
   pSrcSqr, int len, Ipp64f* pDst, Ipp64f occ);
IppStatus ippsDcsClustLCompute_32f64f(const Ipp32f* pSrcSum, const Ipp32f*
   pSrcSqr, int len, Ipp64f* pDst, Ipp32f occ);
```

Arguments

pSrcSum	Pointer to the summation part of the accumulator [*len*].
pSrcSqr	Pointer to the square sum part of the accumulator [*len*].
len	Length of the *pSrcSum* and *pSrcSqr* vectors.
pDst	Pointer to the result likelihood value.
occ	Occupation sum of the HMM state cluster.

Discussion

The function ippsDcsClustLCompute is declared in the ippsr.h file. This function calculates the likelihood of an HMM state cluster, according to the accumulators computed by ippsDcsClustLAccumulate function. The likelihood value is used to determine the splitting of a decision tree node in the decision-tree state-clustering algorithm. The calculation is as follows:

$$pDst[0] = -\frac{1}{2}occ \cdot \left\{ len \cdot [1 + \ln 2\pi - 2\ln(occ)] + \right.$$

$$+ \sum_{i=0}^{len-1} \ln[pSrcSqr[i] \cdot occ - (pSrcSum[i])^2] \Bigg\}$$

Note that if *occ* = 0 , *pDst*[0] is set to IPPLOGZERO.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcSum*, *pSrcSqr* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0 or *occ* is less than or equal to 0.
ippStsZeroOcc	Indicates a warning that *occ* is equal to 0.
ippStsNegOccErr	Indicates an error when *occ* is less than 0.
ippStsLnZeroArg	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to –Inf if there are no negative elements in the vector.
ippStsLnNegArg	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to NaN.

Model Adaptation

This section describes functions that can be used to adapt the acoustic and language models. The adaptation algorithms adjust the parameters of existing models to match the characteristics set by the users, given a few learning samples.

AddMulColumn

Adds a weighted matrix column to the other column.

```
IppStatus ippsAddMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height, int
   col1, int col2, int row1, const Ipp64f val);
```

Arguments

mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	Column number of the first operand.
col2	Column number of the second operand.
row1	Starting row number.
val	Weight factor.

Discussion

The function `AddMulColumn` is declared in the `ippsr.h` file. This function adds a matrix column weighted by *val* to the other matrix column.It is used for fast execution of the SVD algorithm.
The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] + mSrcDst[i][col1] \cdot val,$$

for $row1 \le i < height$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *mSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, *col2*, or *row1* is less than or equal to 0; or *col1* or *col2* is greater than or equal to *width*; or *row1* is greater than or equal to *height*.

AddMulRow

Adds a weighted vector to the other vector.

```
ippsAddMulRow_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
   const Ipp64f val);
```

pSrc	Pointer to the source vector [*len*].
pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the source and destination vectors.
val	Weight factor.

Discussion

The function AddMulRow is declared in the ippsr.h file. This function adds a vector weighted by *val* to the other vector. It is used for fast execution of the SVD algorithm.The calculation is as follows:

$pSrcDst[i] = pSrcDst[i] + pSrc[i] \cdot val, \quad 0 \le i < len$.

Return Value

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

QRTransColumn

Performs the QR transformation.

```
IppStatus ippsQRTransColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
    int col1, int col2, const Ipp64f val1, const Ipp64f val2);
```

Arguments

mSrcDst	Pointer to the source matrix and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	First column number.
col2	Second column number.
val1	First weight factor.
val2	Second weight factor.

Discussion

The ippsQRTransColumn is declared in the ippsr.h file. This function performs the *QR* transform. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] \cdot val1 + mSrcDst[i][col1] \cdot val2 ,$$

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val1 + mSrcDst[i][col2] \cdot val2 ,$$

for $0 \le i < height$.

Return Value

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

ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, or *col2* is less than or equal to 0; or *col1* or *col2* is greater than or equal to *width*; or *col2* is greater than or equal to *height*.

DotProdColumn

Calculates the dot product of two matrix columns.

```
IppStatus ippsDotProdColumn_64f_D2L(const Ipp64f** mSrc, int width,
   int height, Ipp64f* pSum, int col1, int col2, int row1);
```

Arguments

mSrc	Pointer to the source matrix [*height*][*width*].
width	Number of columns in the matrix *mSrc*.
height	Number of rows in the matrix *mSrc*.
pSum	Pointer to the computed sum.
col1	First column number.
col2	Second column number.
row1	First row number.

Discussion

The function ippsDotProdColumn is declared in the ippsr.h file. This function calculates the dot product of two matrix columns. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$pSum[0] = \sum_{i=row1}^{height-1} mSrc[i][col1] \cdot mSrc[i][col2]$$

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *mSrc* or *pSum* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

MulColumn

Multiplies a matrix column by a value.

```
IppStatus ippsMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
   int col1, int row1, const Ipp64f val);
```

Arguments

mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
col1	First column number.
row1	First row number.
val	Weight factor.

Discussion

The function ippsMulColumn is declared in the ippsr.h file. This function multiplies a column of the *mSrcDst* matrix by *val*. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val, \quad row1 \le i < height .$

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *mSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumColumnAbs

Calculates the absolute sum of matrix column elements.

```
IppStatus ippsSumColumnAbs_64f_D2L(const Ipp64f** mSrc, int width,
   int height, Ipp64f* pSum, int col1, int row1);
```

Arguments

mSrc	Pointer to the source matrix [*height*][*width*].
width	Number of columns in the matrix *mSrc*.
height	Number of rows in the matrix *mSrc*.
pSum	Pointer to the computed sum.
col1	First column number.
row1	First row number.

Discussion

The function ippsSumColumnAbs is declared in the ippsr.h file. This function calculates the absolute sum of the matrix column elements. It is used for fast execution of the SVD algorithm.
The calculation is as follows:

$$pSum[0] = \sum_{i=row1}^{height-1} |mSrc[i][col1]|$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *mSrc* or *pSum* pointer is NULL.
ippStsSizeErr	Indicates an error when *width*, *height*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumColumnSqr

Calculates the square sums of weighted matrix column elements.

```
IppStatus ippsSumColumnSqr_64f_D2L(Ipp64f** mSrcDst, int width, int height,
   Ipp64f* pSum, int col1, int row1, const Ipp64f val);
```

Arguments

mSrcDst	Pointer to the source and destination matrix [*height*][*width*].
width	Number of columns in the matrix *mSrcDst*.
height	Number of rows in the matrix *mSrcDst*.
pSum	Pointer to the value of the computed sum.
col1	First column number.
row1	Second row number.
val	Weight factor.

Discussion

The function ippsSumColumnSqr is declared in the ippsr.h file. This function multiplies the columns of the matrix *mSrcDst* by *val* and calculates the square sums of the column elements. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val, \quad 0 \le i < height$$

$$pSum[0] = \sum_{i=row1}^{height-1} (mSrcDst[i][col1])^2$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *mSrcDst* or *pSum* pointer is NULL.
ippStsSizeErr	Indicates an error when *width*, *height*, *col1*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* is greater than or equal to *width*.

SumRowAbs

Calculates the absolute sum of the vector elements.

```
IppStatus ippsSumRowAbs_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
```

Arguments

pSrc	Pointer to the source vector [*len*].
pSum	Pointer to the value of the computed sum.
len	Length of the source vector *pSrc*.

Discussion

The function ippsSumRowAbs is declared in the ippsr.h file. This function calculates the absolute sum of the vector elements as follows:

$$pSum[0] = \sum_{i=0}^{len-1} |pSrc[i]|$$

Return Value

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

SumRowSqr

Calculates the square sum of weighted vector elements.

```
IppStatus ippsSumRowSqr_64f(Ipp64f* pSrcDst, int len, Ipp64f* pSum,
   const Ipp64f val);
```

Arguments

pSrcDst	Pointer to the source and destination vector [*len*].
len	Length of the source vector *pSrcDst*.
pSum	Pointer to the value of the computed sum.
val	Weight factor.

Discussion

The function ippsSumRowSqr is declared in the ippsr.h file. This function multiplies the vector *pSrcDst* by *val* and calculates the square sum of the vector elements as follows:

$$pSrcDst[i] = pSrcDst[i] \cdot val, \quad 0 \le i < len$$

$$pSum[0] = \sum_{i=0}^{len-1} (pSrcDst[i])^2.$$

Return Value

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

SVD

Performs Single Value Decomposition on a matrix.

```
IppStatus ippsSVD_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int height,
   Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int height,
   Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVD_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW,
   Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f* pDstW,
   Ipp64f** mDstV, int width, int nIter);
```

Arguments

pSrcA	Pointer to the input vector *A* [*height*step*].
pDstU	Pointer to the output vector *U* [*height*step*].
pSrcDstA	Pointer to the input matrix *A* and output matrix *U* [*height*step*].
pDstV	Pointer to the output vector *V* [*width*step*].
mSrcA	Pointer to the input matrix *A* [*height*][*width*].
mDstU	Pointer to the output matrix *U* [*height*][*width*].

mSrcDstA	Pointer to the input matrix *A* and output matrix *U* [*height*][*width*].
pDstW	Pointer to the output vector *W* [*width*].
mDstV	Pointer to the output matrix *V* [*width*][*width*].
height	Number of rows in the input matrix.
width	Number of columns in the input matrix.
step	Row step in the vector *pSrcA*, *pSrcDstA*, or *pDstV*.
nIter	Number of iterations for diagonalization.

Discussion

The function ippsSVD is declared in the ippsr.h file. This function performs Single Value Decomposition (SVD) on the input matrix *A*.
The output matrices *U*, *W*, and *V* meet the following condition:

$$A = U \circ W \circ V^T,$$

where the matrix *U* is column-orthogonal, the matrix *W* is diagonal (stored as a vector), and the matrix *V* is orthogonal.
V^T is the transpose of the matrix *V*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *height*, *width*, *step*, or *nIter* is less than or equal to 0, or *width* is greater than *step*.
ippStsSVDCnvgErr	Indicates an error when the SVD algorithm has not converged after *nIter* iterations.

WeightedSum

Calculates the weighted sums of two input vector elements.

```
IppStatus ippsWeightedSum_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
   Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
   Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);
IppStatus ippsWeightedSumHalf_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
   Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
   Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);
```

Arguments

pSrc1	Pointer to the first input vector [*len*].
pSrc2	Pointer to the second input vector [*len*].
pDst	Pointer to the output vector [*len*].
len	Length of the input and output vectors.
weight1	First weight value.
weight2	Second weight value.

Discussion

The functions `ippsWeightedSum` and `ippsWeightedSumHalf` are declared in the `ippsr.h` file. The function `ippsWeightedSum` calculates the weighted sum as follows:

$$pDst[i] = \frac{weight1 \cdot pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, .. len - 1.$$

The function ippsWeightedSumHalf calculates the weighted sum as given by:

$$pDst[i] = \frac{pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, ..len - 1$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc1*, *pSrc2*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZero	Indicates a warning that a divisor vector element has zero value. The execution is not aborted. The value of the destination vector element for the floating-point operations is set as follows: NaN — For zero-valued dividend vector element; +Inf — For positive dividend vector element; -Inf — For negative dividend vector element.

Vector Quantization

This section describes some functions for vector quantization and codebook operations. These functions are commonly used in acoustic and language model compressions.

FormVector

Constructs an output vector of multiple streams from codebook entries.

```
IppStatus ippsFormVector_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
   Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
   Ipp16s* pDst);
IppStatus ippsFormVector_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc, const
   Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
   Ipp16s* pDst);
IppStatus ippsFormVector_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
   Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
   Ipp32f * pDst);
IppStatus ippsFormVector_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc, const
   Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
   Ipp32f * pDst);

IppStatus ippsFormVector_2i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc,
   const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_2i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
   const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_2i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc,
   const Ipp32s* pHeights, Ipp32f* pDst, int len);
IppStatus ippsFormVector_2i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
   const Ipp32s* pHeights, Ipp32f* pDst, int len);

IppStatus ippsFormVector_4i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc,
   const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_4i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
   const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_4i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc,
   const Ipp32s* pHeights, Ipp32f* pDst, int len);
IppStatus ippsFormVector_4i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
   const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

Arguments

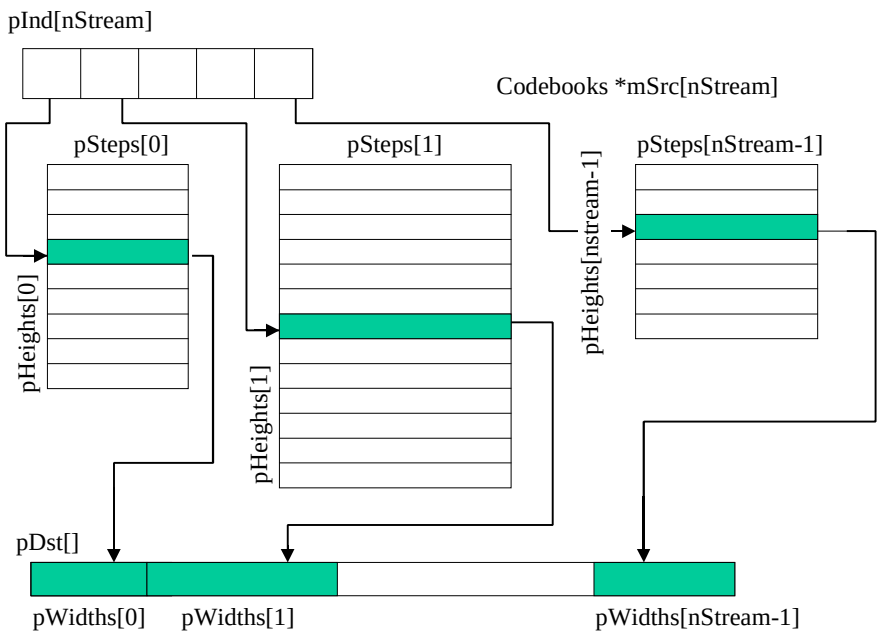
pInd	Pointer to the indexing vector [*nStream*].
mSrc	Pointer to the array of pointers to the codebooks [*nStream*].
pHeights	Pointer to the codebook lengths [*nStream*].
pWidths	Pointer to the stream lengths [*nStream*].
pSteps	Pointer to the codevector lengths [n*Stream*].
nStream	Number of codebooks.
pDst	Pointer to the output vector.
len	Length of the output vector.

Discussion

The function `ippsFormVector` is declared in the `ippsr.h` file. This function constructs an output vector of multiple streams. Each stream, of size *pWeights*[], is a codebook entry indexed by *pInd*[]. The codebooks are referenced by *mSrc*[], and have *pHeights*[] number of codevectors, each of which is of size *pSteps*[].

The following figure illustrates the layout:

Figure 8-5 Stream Layout for the ippsFormVector Function



The output vector is obtained as follows:

$$pDst\left[i + \sum_{j=0}^{k-1} pWidths[j]\right] = mSrc[k][pInd[k] \cdot pSteps[k] + i],$$

$k = 0,\ldots nStream - 1$; $i = 0,\ldots pWidths[k] - 1$.

The function ippsFormVector_2i simplifies the extraction process by posting the constraints of $pWidths[] = pSteps[] = 2$ and $nStream = len/2$.

Similarly, the function ippsFormVector_4i implies the constraints of *pWidths*[] = *pSteps*[] = 4 and *nStream* = *len*/4.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *pInd*[*k*] is less than 0, or when *len*, *nStream*, *pWidths*[*k*], or *pSteps*[*k*] is less than or equal to 0; or when *pHeights*[*k*] is less than or equal to *pInd*[*k*].

CdbkGetSize

Calculates the size in bytes of the codebook.

```
IppStatus ippsCdbkGetSize_16s(int width, int step, int height, int cdbkSize,
   IppCdbk_Hint hint, int *pSize);
```

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. See ippsCdbkInit for a complete description.
pSize	Pointer to the variable to contain the size in bytes of the codebook structure and associated storage.

Discussion

The function `ippsCdbkGetSize` is declared in the `ippsr.h` file. This function calculates the size in bytes of the codebook and additional information to be used for fast search.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSize`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`cdbkSize`* is less than or equal to 0; or *`cdbkSize`* is greater than 8192; or *`cdbkSize`* is not equal to *`height`*; or when *`width`*, *`step`*, or *`height`* is less than or equal to 0; or *`width`* is greater than *`step`*.
`ippStsCdbkFlagErr`	Indicates an error when the *`hint`* value is incorrect or not supported.

NOTE. *The only `hint` value that is currently supported for this function is `IPP_CDBK_FULL`. The `height` parameter is not needed in this mode and its value will be ignored.*

CdbkInit

Initializes the structure that contains the codebook.

```
IppStatus ippsCdbkInit_L2_16s(IppsCdbkState_16s* pCdbk, const Ipp16s* pSrc,
   int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
```

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 logarithmical search.
	IPP_CDBK_KMEANS_NUM LBG algorithm with splitting of the most numerous clusters was used for the codebook building. The nearest codebook entry is located through a logarithmical search.
pCdbk	Pointer to the codebook structure to be initialized.

Discussion

The function ippsCdbkInit is declared in the ippsr.h file. This function initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by the ippsSplitVQ function. The Euclidean distance is used to measure the similarity between two vectors.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pSrc* pointers are null.
ippStsSizeErr	Indicates an error when *width*, *step*, *height*, or *cdbkSize* is less than or equal to 0; or width is greater than *step*; or *cdbkSize* is greater than 8192.
ippStsCdbkFlagErr	Indicates an error when the *hint* value is incorrect or not supported.

NOTE. *The only hint value that is currently supported for this function is* IPP_CDBK_FULL. *State memory address stored in pCdbk must be aligned to 32-bit word boundary.*

CdbkInitAlloc

Initializes the codebook structure.

```
IppStatus ippsCdbkInitAlloc_L2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
   pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_L2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
   pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_WgtL2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
   pSrc, const Ipp16s* pWgt, int width, int step, int height, int cdbkSize,
   Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_WgtL2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
   pSrc, const Ipp32f* pWgt, int width, int step, int height, int cdbkSize,
   Ipp_Cdbk_Hint hint);
```

Arguments

pCdbk	Pointer to the codebook structure to be created.

pSrc	Pointer to the source vector [*height***step*].
pWgt	Pointer to the weight vector [width].
width	Length of the input vectors.
step	Row step in the source vector *pSrc*.
height	Number of rows in the source vector *pSrc*.
cdbkSize	Size of the codebook.
hint	One of the following values:

IPP_CDBK_FULL	The source data are entries of a codebook, *height* should be greater or equal to *cdbkSize*. The nearest codebook entry is located through a full search.
IPP_CDBK_KMEANS_LONG	LBG algorithm with splitting of the most extensional cluster is used for the codebook building. The nearest codebook entry is located through a logarithmical search.
IPP_CDBK_KMEANS_NUM	LBG algorithm with splitting of the most numerous clusters is used for the codebook building. The nearest codebook entry is located through a logarithmical search.

Discussion

The function ippsCdbkInitAlloc is declared in the ippsr.h file. This function initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by ippsVQ or ippsSplitVQ functions.
The ippsCdbkInitAlloc_L2 function uses the Euclidean distance to measure the similarity between two vectors, while the ippsCdbkInitAlloc_WgtL2 function uses the weighted Euclidean distance for these purposes.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pCdbk* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *width*, *step*, or *cdbkSize* is less than or equal to 0; or *cdbkSize* is greater than *height*; or *width* is greater than *step*; or *cdbkSize* is greater than 16383; or *hint* is equal to IPP_CDBK_FULL and *cdbkSize* is not equal to *height*.
ippStsCdbkFlagErr	Indicates an error when the *hint* value is incorrect.
ippStsMemAllocErr	Indicates an error when no memory was allocated.
ippStsBadArgErr	Indicates an error when one of *pWgt*[i] is less than or equal to 0.

CdbkFree

Destroys the codebook structure.

```
IppStatus ippsCdbkFree_16s(IppsCdbkState_16s* pCdbk);
IppStatus ippsCdbkFree_32f(IppsCdbkState_32f* pCdbk);
```

Arguments

pCdbk	Pointer to the codebook structure.

Discussion

The function ippsCdbkFree is declared in the ippsr.h file. This function destroys the codebook structure and frees all memory associated with it.

Return Value

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

GetCdbkSize

Retrieves the number of codevectors in the codebook.

```
IppStatus ippsGetCdbkSize_16s(const IppsCdbkState_16s* pCdbk, int* pNum);
IppStatus ippsGetCdbkSize_32f(const IppsCdbkState_32f* pCdbk, int* pNum);
```

Arguments

pCdbk	Pointer to the codebook structure.
pNum	Pointer to the result number of codevectors.

Discussion

The function `ippsGetCdbkSize` is declared in the `ippsr.h` file. This function retrieves the number of codevectors in the codebook *pCdbk*. This number could be less than *cdbkSize* if the number of different vectors in *pSrc* is less than *cdbkSize*. The codebook structure *pCdbk* is initialized by the `ippsCdbkAlloc` function.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCdbk* or *pNum* pointer is `NULL`.

GetCodebook

Retrieves the codevectors from the codebook.

```
IppStatus ippsGetCodebook_16s(const IppsCdbkState_16s* pCdbk, Ipp16s* pDst,
   int step);
IppStatus ippsGetCodebook_32f(const IppsCdbkState_32f* pCdbk, Ipp32f* pDst,
   int step);
```

Arguments

pCdbk	Pointer to the codebook structure.
pDst	Pointer to the destination vector for codevectors [*pNum*[0]**step*].
step	Row step in the destination vector *pDst*.

Discussion

The function ippsGetCodebook is declared in the ippsr.h file. This function retrieves the codevectors from the codebook structure *pCdbk* and stores them in the *pDst* vector with row step *step*.

The codebook structure *pCdbk* is initialized by the function ippsCdbkAlloc. The number of clusters can be obtained by the function ippsGetCdbkSize.

Return Value

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

VQ

Quantizes the input vectors given a codebook.

```
IppStatus ippsVQ_16s(const Ipp16s* pSrc, int step, Ipp32s* pIndx, int height,
   const IppsCdbkState_16s* pCdbk);
IppStatus ippsVQ_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx, int height,
   const IppsCdbkState_32f* pCdbk);
IppStatus ippsVQDist_16s32s_Sfs(const Ipp16s* pSrc, int step, Ipp32s* pIndx,
   Ipp32s* pDist, int height, const IppsCdbkState_16s* pCdbk,
   int scaleFactor);
IppStatus ippsVQDist_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx,
   Ipp32f* pDist, int height, const IppsCdbkState_32f* pCdbk);
```

Arguments

pCdbk	Pointer to the codebook structure.
pSrc	Pointer to the source vector [*height***step*].
step	Row step in the source vector *pSrc*.
height	Number of rows in the source vector *pSrc*.
pIndx	Pointer to the result index vector of the closest codevectors [*height*].
pDist	Pointer to the result quantization distances from the source vector [*height*].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions ippsVQ and ippsVQDist are declared in the ippsr.h file. The function ippsVQ performs Vector Quantization (VQ) on the input vectors. The resulting indexes of the closest codevectors are stored in the vector *pIndx*. The function ippsVQDist also stores the distances (scaled with *scaleFactor* for the integer versions) to the output distance vector *pDist*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk*, *pSrc*, *pIndx*, or *pDist* pointer is NULL.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.

VQSingle_Sort, VQSingle_Thresh

Quantizes the input vector given a codebook and gets several closest clusters.

```
IppStatus ippsVQSingle_Sort_32f(const Ipp32f *pSrc, Ipp32s *pIndx, const
   IppsCdbkState_32f* pCdbk, int num);
IppStatus ippsVQSingle_Sort_16s(const Ipp16s *pSrc, Ipp32s *pIndx, const
   IppsCdbkState_16s* pCdbk, int num);
IppStatus ippsVQDistSingle_Sort_32f(const Ipp32f *pSrc, Ipp32s *pIndx, Ipp32f
   *pDist, const IppsCdbkState_32f* pCdbk, int num);
IppStatus ippsVQDistSingle_Sort_16s32s_Sfs(const Ipp16s *pSrc, Ipp32s *pIndx,
   Ipp32s *pDist, const IppsCdbkState_16s* pCdbk, int num, int scaleFactor);
IppStatus ippsVQSingle_Thresh_32f(const Ipp32f *pSrc, Ipp32s *pIndx, const
   IppsCdbkState_32f* pCdbk, Ipp32f val, int *pnum);
IppStatus ippsVQSingle_Thresh_16s(const Ipp16s *pSrc, Ipp32s *pIndx, const
   IppsCdbkState_16s* pCdbk, Ipp32f val, int *pnum);
IppStatus ippsVQDistSingle_Thresh_32f(const Ipp32f *pSrc, Ipp32s *pIndx,
   Ipp32f *pDist, const IppsCdbkState_32f* pCdbk, Ipp32f val, int *pnum);
IppStatus ippsVQDistSingle_Thresh_16s32s_Sfs(const Ipp16s *pSrc, Ipp32s
   *pIndx, Ipp32s *pDist, const IppsCdbkState_16s* pCdbk, Ipp32f val, int
   *pnum, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
pIndx	Pointer to the destination indexes vector.
pDist	Pointer to the destination distances vector.
pCdbk	Pointer to the codebook structure.
val	Relative threshold value.
num	Number of closest clusters to search for each input vector.
pnum	Pointer to the number of clusters within threshold [1].

scaleFactor	Scaling factor for intermediate sums.

Discussion

The functions ippsVQSingle_Sort and ippsVQSingle_Thresh are declared in the ippsr.h file. These functions perform multiple vector quantization (VQ) for the input vector. Functions with Sort suffix provide *num* indexes of closest codebook centroids sorted in distance ascending order. Functions with Thresh suffix provide indexes of codebook centroids with distance less than minimum multiplied by *val* and the number of such clusters. Functions with Dist suffix provide distance values of closest clusters. The length of vectors *pSrc* is equal to codevector length, the length of output vectors *pIndx* and *pDist* is equal to codebook size. Full search is done for all codebook types.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pIndx*, *pDist*, *pCdbk* or *pnum* pointer is NULL.
ippStsSizeErr	Indicates an error when *num* is less than or equal to 0, or greater than the codebook size.
ippStsBadArgErr	Indicates an error when *val* is less than 1.

SplitVQ

Quantizes a multiple-stream vector given the codebooks.

```
IppStatus ippsSplitVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s* pDst,
   int dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_16s8u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst, int
   dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsSplitVQ_16s1u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst, int
   dstBitStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
```

```
IppStatus ippsSplitVQ_32f16s(const Ipp32f* pSrc, int srcStep, Ipp16s* pDst,
   int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsSplitVQ_32f8u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst, int
   dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsSplitVQ _32f1u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst, int
   dstBitStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
```

Arguments

pCdbks	Pointer to the codebook structures [*nStream*].
pSrc	Pointer to the source vector [*height*srcStep*].
srcStep	Row step in the source vector *pSrc*.
pDst	Pointer to the destination indexing vector [*height*dstStep*].
dstStep	Row step in the destination vector *pDst*.
height	Number of rows in the source and destination vectors.
dstBitStep	Row step in the destination vector (in bits).
nStream	Number of streams in the source vectors.

Discussion

The functions ippsSplitVQ is declared in the ippsr.h file. This function quantizes the multiple-stream vectors *pSrc* against given codebooks *pDdbks*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors. The outputs *pDst* are indexes to the codebook entries.

For functions with the 1u suffix, the output indexes are packed in bits. Each stream takes the least number of bits sufficient to represent its codebook indexes.

See also ippsFormVector, ippsFormVectorVQ functions.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pCdbk*, *pCdbk*[*k*], *pSrc*, or *pDst* pointer is NULL.

ippStsSizeErr	Indicates an error when *srcStep*, *dstStep*, *height*, or *nStream* is less than or equal to 0; or the sum of the stream length is greater than *srcStep;* or *nStream* is greater than *dstStep* for functions with the 16s or 8u suffix; or the number of bits sufficient to represent the indexes is greater than *dstStep* for functions with the 1u suffix; or the codebook size is greater than 256 for functions with the 8u suffix.

FormVectorVQ

Constructs multiple-stream vectors from codebooks, given indexes.

```
IppStatus ippsFormVectorVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s*
   pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks, int
   nStream);
IppStatus ippsFormVectorVQ_8u16s(const Ipp8u* pSrc, int srcStep, Ipp16s* pDst,
   int dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);
IppStatus ippsFormVectorVQ_1u16s(const Ipp8u* pSrc, int srcBitStep, Ipp16s*
   pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks, int
   nStream);
IppStatus ippsFormVectorVQ_16s32f(const Ipp16s* pSrc, int srcStep, Ipp32f*
   pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks, int
   nStream);
IppStatus ippsFormVectorVQ_8u32f(const Ipp8u* pSrc, int srcStep, Ipp32f* pDst,
   int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
IppStatus ippsFormVectorVQ_1u32f(const Ipp8u* pSrc, int srcBitStep, Ipp32f* pDst,
   int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);
```

Arguments

pCdbks	Pointer to the codebook structures [*nStream*].
pSrc	Pointer to the indexing vectors [*height*srcStep*].

pDst	Pointer to the constructed vectors [*height***dstStep*].
srcStep	Row step in the indexing vectors *pSrc*.
srcBitStep	Row step in the indexing vectors *pSrc* (in bits).
dstStep	Row step in the constructed vectors *pDst*.
height	Number of rows in the vectors *pSrc*.
nStream	Number of streams.

Discussion

The function ippsFormVectorVQ is declared in the ippsr.h file. This function constructs multiple-stream vectors *pDst* from the codebooks *pCdbks* given indexes *pSrc*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors.
For functions with the 1u suffix, each stream index is assumed to be in a packed format. Each stream takes the number of bits sufficient to represent its codebook indexes.
See also ippsFormVector, ippsSplitVQ functions.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pCdbk*, *pCdbk*[*k*], *pSrc*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *srcStep*, *dstStep*, *height*, or *nStream* is less than or equal to 0; or the codevector length sum is greater than *dstStep*; or *nStream* is greater than *srcStep* for functions with the 16s or 8u suffix; or the number of bits sufficient to represent the indexes is greater than *srcStep* for functions with the 1u suffix.

Polyphase Resampling

The Intel IPP functions described in this section build, apply and free Kaiser-windowed polyphase filters for data resampling. Functions with `Fixed` suffix are intended for fixed rational resampling factor and provide faster speed. Functions without this suffix build universal resampling filter with linear interpolation of filter coefficients and allow to use a variable factor.

ResamplePolyphaseInitAlloc

Initializes the structure for polyphase data resampling.

```
IppStatus ippsResamplePolyphaseInitAlloc_16s(IppsResamplePolyphaseSpec_16s**
   pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha,
   IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseInitAlloc_32f(IppsResamplePolyphaseSpec_32f**
   pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha,
   IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedInitAlloc_16s(
   IppsResamplePolyphaseSpecFixed_16s** pSpec, int inRate, int outRate, int
   len, Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedInitAlloc_32f(
   IppsResamplePolyphaseSpecFixed_32f** pSpec, int inRate, int outRate, int
   len, Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm hint);
```

Arguments

window	The size of the ideal lowpass filter window.
nStep	The discretization step for filter coefficients.
rollf	The roll-off frequency of the filter.
alpha	The parameter of the Kaiser window.
inRate	The input rate for resampling with fixed factor.
outRate	The output rate for resampling with fixed factor.

len	The filter length for resampling with fixed factor.
pSpec	The pointer to the resampling state structure to be created.
hint	Suggests using specific code. The values for the *hint* argument are described in Table 7-3, "Flag and Hint Arguments".

Discussion

These functions are declared in the ippsr.h file.
The function ippsResamplePolyphaseInitAlloc creates structures for data resampling using the ideal lowpass filter

$$h[i] = \begin{cases} \dfrac{rollf}{nStep}, & i = 0 \\ \dfrac{\sin(rollf \cdot i\pi / nStep)}{i\pi / nStep}, & i \neq 0 \end{cases}, \quad |i| \le window \cdot nStep \quad , \quad 0 < rollf \le 1$$

and applies the Kaiser window with *alpha* parameter and width *window* to it.

The structure created could be used to resample input samples by the function ippsResample with arbitrary resampling factor. Linear interpolation is used to calculate filter coefficients for each output sample. The filter size depends on the resampling factor.

The function ippsResamplePolyphaseFixedInitAlloc creates structures for data resampling with the factor equal to *inRate/outRate*. Values *window* and *nStep* in the formula above are chosen to provide the filter length

$$flen = \min_{l \ge len, l\%4=0} l$$

and *flen*+1 for zero phase.

$$fnum = outRate / \mathrm{HHO}(inRate, outRate)$$

of filters are created for different phases.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSpec* pointer is NULL.
ippStsSizeErr	Indicates an error when *inRate*, *outRate*, *nStep* or *len* is less than or equal to 0.
ippStsBadArgErr	Indicates an error when *rollf* is less than or equal to 0 or is greater than 1, or if *alpha* is less than 1, or if *window* is less than 2/*nStep*.
ippStsMemAllocErr	Indicates a memory allocation error.

ResamplePolyphaseFree

Frees structure for polyphase data resampling.

```
IppStatus ippsResamplePolyphaseFree_16s(IppsResamplePolyphaseSpec_16s* pSpec);
IppStatus ippsResamplePolyphaseFree_32f(IppsResamplePolyphaseSpec_32f* pSpec);
IppStatus ippsResamplePolyphaseFixedFree_16s(IppsResamplePolyphaseSpecFixed_16s*
   pSpec);
IppStatus ippsResamplePolyphaseFixedFree_32f(IppsResamplePolyphaseSpecFixed_32f*
   pSpec);
```

Arguments

pSpec	The pointer to the resampling state structure.

Discussion

The functions ippsResamplePolyphaseFree and ippsResamplePolyphaseFixedFree are declared in the ippsr.h file. These functions close the resampling structure by freeing all memory allocated for it by ippsResamplePolyphaseInitAlloc or ippsResamplePolyphaseFixedInitAlloc.

Return Value

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

ResamplePolyphase

Resamples input data using polyphase filters.

```
IppStatus ippsResamplePolyphase_16s(const Ipp16s *pSrc, int len, Ipp16s *pDst,
   int *pOutlen, Ipp64f factor, Ipp32f norm, Ipp64f *pTime, const
   IppsResamplePolyphaseSpec_16s *pSpec);
IppStatus ippsResamplePolyphase_32f(const Ipp32f *pSrc, int len, Ipp32f *pDst,
   int *pOutlen, Ipp64f factor, Ipp32f norm, Ipp64f *pTime, const
   IppsResamplePolyphaseSpec_32f *pSpec);
IppStatus ippsResamplePolyphaseFixed_16s(const Ipp16s *pSrc, int len, Ipp16s
   *pDst, int *pOutlen, Ipp32f norm, Ipp64f *pTime, const
   IppsResamplePolyphaseSpecFixed_16s *pSpec);
IppStatus ippsResamplePolyphaseFixed_32f(const Ipp32f *pSrc, int len, Ipp32f
   *pDst, int *pOutlen, Ipp32f norm, Ipp64f *pTime, const
   IppsResamplePolyphaseSpecFixed_32f *pSpec);
```

Arguments

pSpec	The pointer to the resampling state structure.
pSrc	The pointer to the input vector.
pDst	The pointer to the output vector.
len	The number of input vector elements to resample.
norm	The norm factor for output samples.
factor	The resampling factor.
pTime	The pointer to the start time of resampling (in input vector elements).
pOutlen	The number of calculated output vector elements.

Discussion

The functions `ippsResamplePolyphase` and `ippsResamplePolyphaseFixed` are declared in the `ippsr.h` file. These functions convert data from the input vector changing their frequency. The ratio of output and input frequencies is defined by the *`factor`* argument for the function `ippsResamplePolyphase`. For the function `ippsResamplePolyphaseFixed`, this ratio is defined during creation of the resampling structure. The value *`pTime`*`[0]` defines the time value for which the first output sample is calculated.

The history data of filters are in the input vector with indexes less than *`pTime`*`[0]`. The history length is equal to *flen*/2 for function `ippsResamplePolyphaseFixed` and

$$\left\lceil \frac{1}{2} window \cdot \max(1, 1/\ factor) \right\rceil + 1$$

for function `ippsResamplePolyphase`. The input vector should contain the same number of elements with indexes greater than *`pTime`*`[0]+`*`len`* for the right filter wing for the last element.

After function execution, the time value is updated and *`pOutlen`*`[0]` contains the number of calculated output samples.

The output samples are multiplied by *`norm`*$\cdot \min(1,$ *`factor`*$)$ before saturation.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSpec`*, *`pSrc`*, *`pDst`*, *`pTime`* or *`pOutlen`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsBadArgErr`	Indicates an error when *`factor`* is less than or equal to.

Example 8-3 Resampling of the input mono pcm file

```
void resampleIPP(
   int      inRate,    // input frequency
   int      outRate,   // output frequency
```

```
   FILE    *infd,      // input pcm file
   FILE    *outfd)     // output pcm file
{
   short *inBuf,*outBuf;
   int bufsize=4096;
   int history=128;
   double time=history;
   int lastread=history;
   int inCount=0,outCount=0,inLen,outLen;
   IppsResamplePolyphaseSpecFixed_16s *state;

   ippsResamplePolyphaseFixedInitAlloc_16s(&state,inRate,outRate,2*history,
                                  0.95f,9.0f,ippAlgHintAccurate);
   inBuf=ippsMalloc_16s(bufsize+history+2);
   outBuf=ippsMalloc_16s((int)((bufsize-history)*outRate/(float)inRate+2));

   ippsZero_16s(inBuf,history);
  while ((inLen=fread(inBuf+lastread,sizeof(short),bufsize-lastread,infd))>0)
{
      inCount+=inLen;
      lastread+=inLen;
      ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-history-(int)time,
                           outBuf,0.98f,&time,&outLen);
      fwrite(outBuf,outLen,sizeof(short),outfd);
      outCount+=outLen;
      ippsMove_16s(inBuf+(int)time-history,inBuf,lastread+history-(int)time);
      lastread-=(int)time-history;
      time-=(int)time-history;
   }
   ippsZero_16s(inBuf+lastread,history);
   ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-(int)time,
                         outBuf,0.98f,&time,&outLen);
   fwrite(outBuf,outLen,sizeof(short),outfd);
   outCount+=outLen;
```

```
   printf("%d inputs resampled to %d outputs\n",inCount,outCount);
   ippsFree(outBuf);
   ippsFree(inBuf);
   ippsResamplePolyphaseFixedFree_16s(state);
}
```

Advanced Aurora Functions

The Intel IPP functions discussed later in this section are implemented to support the speech processing and compression algorithm described in ETSI ES 202 050 V1.1.1 standard (see [ES202]).

SmoothedPowerSpectrum_Aurora

Calculates smoothed magnitude of the FFT output.

```
IppStatus ippsSmoothedPowerSpectrum_Aurora_16s(const Ipp16s* pSrc, Ipp16s*
   pDst, int len);
IppStatus ippsSmoothedPowerSpectrum_Aurora_32f(const Ipp32f* pSrc, Ipp32f*
   pDst, int len);
```

Arguments

pSrc	Pointer to the input vector in PERM format [*len*].
pDst	Pointer to the output vector [*len*/4+1].
len	The input vector length (multiple of 4).

Discussion

The function `ippsSmoothedPowerSpectrum_Aurora` is declared in the `ippsr.h` file. This function calculates the smoothed square magnitude for the Fast Fourier Transform (FFT) output vector in PERM format ([ES202] , 5.1.3-5.1.4):

$$pDst[0] = \left(pSrc[0]^2 + pSrc[2]^2 + pSrc[3]^2\right)/2$$

$$pDst[i] = \left(pSrc[4\cdot i]^2 + pSrc[4\cdot i+1]^2 + pSrc[4\cdot i+2]^2 + pSrc[4\cdot i+3]^2\right)/2$$

$$pDst[len/4] = pSrc[1]^2$$

where $0 < i < len/4$.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0, or *`len`* is not a multiple of 4.

NoiseSpectrumUpdate_Aurora

Updates the noise spectrum.

```
IppStatus ippsNoiseSpectrumUpdate_Aurora_32f(const Ipp32f* pSrc, const Ipp32f*
   pSrcNoise,  Ipp32f* pDst, int len);
```

Arguments

`pSrc`	Pointer to the input vector of the power spectral density mean [*`len`*].
`pSrcNoise`	Pointer to the input vector of the previous noiseless signal spectrum [*`len`*].

pDst	Pointer to the output vector of the improved transfer function [*len*].
len	The length of input and output vectors.

Discussion

The function ippsNoiseSpectrumUpdate is declared in the ippsr.h file. This function updates the noise spectrum estimate on the second stage of noise reduction according to the formula ([ES202], 5.1.5):

$$pDst[i] = pSrcNoise[i] \cdot \left(0.9 + \frac{0.1 \cdot pSrc[i]}{pSrc[i] + pSrcNoise[i]} \cdot \left(1 + \frac{pSrcNoise[i]}{pSrcNoise[i] + 0.1 \cdot pSrc[i]} \right) \right)$$

for $0 \le i < len$.

Return Value

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

WienerFilterDesign_Aurora

Calculates an improved transfer function of the adaptive Wiener filter.

```
IppStatus ippsWienerFilterDesign_Aurora_32f(const Ipp32f* pSrc, const Ipp32f*
   pNoise, const Ipp32f* pDen, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the input vector of square roots of the power spectral density mean [*len*].

pNoise	Pointer to the input vector of square roots of the noise spectrum estimate [*len*].
pDen	Pointer to the input vector of square roots of the previous noiseless signal spectrum [*len*].
pDst	Pointer to the output vector of the improved transfer function [*len*].
len	The length of input and output vectors.

Discussion

The function ippsWienerFilterDesign_Aurora is declared in the ippsr.h file. This function processes the input vector according to the following formulas ([ES202], 5.1.5):

1. The noiseless signal is estimated:

$$p[i] = 0.98 \cdot pDen[i] + 0.02 \cdot \max(pSrc[i] - pNoise[i], 0)$$

2. The a priori Signal to Noise Ratio (SNR) is computed:

$$\eta[i] = p[i] / pNoise[i]$$

3. Then the filter transfer function

$$H[i] = \frac{\eta[i]}{1 + \eta[i]}$$

is used to improve the estimation of the noiseless signal spectrum:

$$p2[i] = H[i] \cdot pSrc[i]$$

4. The improved a priori SNR is calculated as:

$$\eta 2[i] = \max\left(p2[i] / pNoise[i], 0.079432823\right)$$

5. The improved filter transfer function is obtained as:

$$pDst[i] = \frac{\eta 2[i]}{1 + \eta 2[i]}$$

for $0 \le i < len$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pNoise*, *pDen*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsBadArg	Indicates an error when a negative or NaN argument of a square root operation is detected.

MelFBankInitAlloc_Aurora

Initializes the structure for performing the Mel-frequency filter bank analysis.

```
IppStatus ippsMelFBankInitAllocLow_Aurora_16s(IppsFBankState_16s** pFBank);
IppStatus ippsMelFBankInitAllocLow_Aurora_32f(IppsFBankState_32f** pFBank);
IppStatus ippsMelFBankInitAllocHigh_Aurora_16s(IppsFBankState_16s** pFBank);
IppStatus ippsMelFBankInitAllocHigh_Aurora_32f(IppsFBankState_32f** pFBank);
```

Arguments

pFBank	Pointer to the Mel-scale filter bank structure to be created.

Discussion

The function ippsMelFBankInitAlloc_Aurora is declared in the ippsr.h file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis. The function with Low suffix builds a filter bank of 23 filters for 8 KHz data processing. The function with High suffix builds a filter bank of 3 filters for high frequency band of 16 KHz data processing ([ES202], 5.1.7).

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pFBank* pointer is NULL.

TabsCalculation_Aurora

Calculates filter coefficients for residual filter.

```
IppStatus ippsTabsCalculation_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s*
   pDst);
IppStatus ippsTabsCalculation_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst);
```

Arguments

pSrc	Pointer to the input vector of Mel-scaled filter bank output [10].
pDst	Pointer to the output filter coefficients vector [17].

Discussion

The function ippsTabsCalculation_Aurora is declared in the ippsr.h file. This function calculates residual filter coefficients for the Mel-scaled filter bank output ([ES202], 5.1.9-5.1.10).

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is not equal to 23 or 26.

ResidualFilter_Aurora

Calculates a denoised waveform signal.

```
IppStatus ippsResidualFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
```

```
const Ipp16s* pTabs, int scaleFactor);
IppStatus ippsResidualFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
Ipp32f* pTabs);
```

Arguments

pTabs	Pointer to the filter coefficients vector [17].
pSrc	Pointer to the input vector [96].
pDst	Pointer to the output vector [80].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsResidualFilter_Aurora is declared in the ippsr.h file.
This function filters the input vector to get the low band and high band parts ([ES202], 5.1.10):

$$pDst[j] = \sum_{i=0}^{16} pTabs[i] \cdot pSrc[i+j] \quad , \qquad 0 \le j < 80$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pTabs* pointer is NULL.

WaveProcessing_Aurora

Processes waveform data after noise reduction.

```
IppStatus ippsWaveProcessing_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst);
```

Arguments

pSrc	Pointer to the input vector [200].
pDst	Pointer to the output vector [200].

Discussion

The function ippsWaveProcessing_Aurora is declared in the ippsr.h file. This function processes the input vector according to the following formulas ([ES202], 5.2):

1. Smoothed Teager operator is calculated:

$$s[i] = \frac{1}{9}\sum_{k=-4}^{4} s_T[i] \quad , \qquad 0 \le i < 200$$

where

$$s_T[i] = \left| pSrc[i]^2 - pSrc[\max(0, i-1)] \cdot pSrc[\min(199, i+1)] \right|$$

for $0 \le i < 200$,

and $s[i] = 0$ for $i < 0$ and $200 \le i$.

2. Peaks

$$p[l_{\min}], \ldots p[l_{\max}]$$

are found:

$$p[0] = \arg\max_{0 \le i < 200} s[i]$$

$$p[l+1] = \arg\max_{\substack{p[l]+25 \le k < p[l]+80 \\ k < 200}} s[k] \quad , \qquad l = 0, \ldots l_{\max} - 1$$

$$p[l-1] = \arg\max_{\substack{p[l]-25 \ge k > p[l]-80 \\ k \ge 0}} s[k] \quad , \qquad l = l_{\min}+1, \ldots 0$$

3. Weighting function is calculated:

$$w[i] = \begin{cases} 1.2, & p[k]-4 < i < p[k]+0.8\cdot\rceil p[k+1]-p[k]\lceil \\ 1.0, & i = p[k]-4, i = p[k]+0.8\cdot\rceil p[k+1]-p[k]\lceil \\ 0.8 & \text{otherwise} \end{cases}$$

$l_{\min} \le k < l_{\max}$

4. Output vector is computed as:

$pDst[i] = w[i]\cdot pSrc[i]\ ,\qquad 0 \le i < 200$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is `NULL`.

LowHighFilter_Aurora

Calculates low band and high band filters.

```
IppStatus ippsLowHighFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s*
   pDstLow, Ipp16s* pDstHigh, int len, const Ipp16s* pTabs, int tapsLen, int
   scaleFactor);
IppStatus ippsLowHighFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDstLow,
   Ipp32f* pDstHigh, int len, const Ipp32f* pTabs, int tapsLen);
```

Arguments

`pSrc`	Pointer to the input vector [*`len`*+*`tapsLen`*-1].
`pDstLow`	Pointer to the low frequency band output vector [*`len`*/2].
`pDstHigh`	Pointer to the high frequency band output vector [*`len`*/2].
`len`	The input samples number (even).
`pTabs`	Pointer to the filter coefficients vector [*`tapsLen`*].

tapsLen	The filter taps number (even).
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsLowHighFilter_Aurora is declared in the ippsr.h file. This function filters the input vector to get the low band and high band parts ([ES202], 5.5.1):

$$pDstLow[j] = \sum_{i=0}^{tapslen-1} pTabs[i] \cdot pSrc[i + 2 \cdot j]$$

$$pDstHigh[j] = \sum_{i=0}^{tapslen-1} pTabs[i] \cdot pSrc[i + 2 \cdot j] \cdot (-1)^{i+j} \quad , \qquad 0 \le j < len/2$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDstLow*, *pDstHigh*, or *pTabs* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0, or *tapsLen* is less than or equal to 0, or *len* or *tapsLen* is not even.

HighBandCoding_Aurora

Codes and decodes the high frequency band energy values.

```
IppStatus ippsHighBandCoding_Aurora_32f(const Ipp32f* pSrcHFB, const Ipp32f*
   pInSWP, const Ipp32f* pDSWP, Ipp32f* pDstHFB);
```

Arguments

pSrcHFB	Pointer to the input high frequency band energy vector [3].
pInSWP	Pointer to the input signal smoothed power spectrum vector [65].
pDSWP	Pointer to the denoised signal power spectrum vector [129].
pDstHFB	Pointer to the output coded high frequency band log energy vector [3].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function ippsHighBandCoding_Aurora is declared in the ippsr.h file. This function performs coding and decoding of high frequency band energies according to the following formulas ([ES202], 5.5.3):

1. Logarithm operation is applied to high frequency band energies:

$$S_{HFB}[k] = \ln pSrcHFB[k] \quad , \qquad 0 \le k < 3$$

2. Auxiliary log energies of the input signal power spectrum are calculated:

$$S_{LFB\_aux}[l] = \max\left(-50, \ln\left(\sum_{i=n1[l]}^{n1[l+1]-1} pInSWP[i]\right)\right) \quad , \qquad 0 \le l < 3$$

$n1[0] = 33,\ n1[1] = 39,\ n1[2] = 49,\ n1[3] = 65.$

3. Energies are coded as:

$$code[k,l] = S_{LFB\_aux}[k] - S_{HFB}[l] \quad , \qquad 0 \le k,l < 3$$

4. Auxiliary bands for decoding are calculated:

$$S_{Dec\_aux}[l] = \max\left(-50, \ln\left(\frac{1}{2}\sum_{i=n2[l]}^{n2[l+1]-1} pDSWP[i]\right)\right) \quad , \qquad 0 \le l < 3$$

$n2[0] = 66,\ n2[1] = 77,\ n2[2] = 97,\ n2[3] = 129.$

5. Decoded high frequency band energies are computed as:

$$pDstHFB\ [k] = \sum_{l=0}^{2} w[l] \cdot \left(S_{Dec\_aux}[l] - code[l,k]\right), \quad 0 \le k < 3$$

$w[0] = 0.1,\ w[1] = 0.2,\ w[2] = 0.7.$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrcHFB`*, *`pInSWP`*, *`pDSWP`*, or *`pDstHFB`* pointer is `NULL`.
`ippStsBadArg`	Indicates an error when a non-positive or `NaN` argument of the logarithm operation is detected.

BlindEqualization_Aurora

Equalizes the cepstral coefficients.

```
IppStatus ippsBlindEqualization_Aurora_32f(const Ipp32f* pRefs, Ipp32f* pCeps,
   Ipp32f* pBias, int len, Ipp32f val);
```

Arguments

`pRefs`	Pointer to the input vector of reference cepstrum [*`len`*].
`pCeps`	Pointer to the input and output vector of cepstrum [*`len`*].
`pBias`	Pointer to the input and output vector of bias [*`len`*].
`len`	The number of cepstral coefficients.
`val`	The log energy value.
`scaleFactor`	Refer to "Integer Scaling" in Chapter 2.

Discussion

The function `ippsBlindEqualization_Aurora` is declared in the `ippsr.h` file. This function equalizes cepstral coefficients using the LMS algorithm ([ES202], 5.4):

$$step = 0.0087890625 \cdot \min(1, \max(0, val - 211/64))$$

$$pCeps[i] = pCeps[i] - pBias[i]$$

$$pBias[i] += step \cdot (pCeps[i] - pRefs[i]) \quad , \qquad 0 \le i < len$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pRefs`*, *`pCeps`*, or *`pBias`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

DeltaDelta_Aurora

Calculates the first and second derivatives according to ETSI ES 202 050 standard.

```
IppStatus ippsDeltaDelta_Aurora_16s_D2Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
   int dstStep, int height, int deltaMode, int scaleFactor);
IppStatus ippsDeltaDeltaMul_Aurora_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
   pVal, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
   scaleFactor);
IppStatus ippsDeltaDelta_Aurora_32f_D2(const Ipp32f* pSrc, Ipp32f* pDst, int
   dstStep, int height, int deltaMode);
IppStatus ippsDeltaDeltaMul_Aurora_32f_D2(const Ipp32f* pSrc, const Ipp32f*
   pVal, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

Arguments

`pSrc`	Pointer to the input feature sequence [*`height`**14].

pDst	Pointer to the output feature sequence [*height***dstStep*].
dstStep	Length of the output feature in the output sequence *pDst*.
height	Number of feature vectors.
deltaMode	Execution mode.
pVal	Pointer to the delta coefficients vector [39].
scaleFactor	Refer to "Integer Scaling" in Chapter 2.

Discussion

The functions ippsDeltaDelta_Aurora and ippsDeltaDeltaMul_Aurora are declared in the ippsr.h file. These functions calculate full feature vectors according to ETSI ES 202 050 standard.
The input vectors of length 14 contain values $c_1,...c_{12}$, c_0, ln$E$. First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated ([ES202], 9.1-9.2).

The function implies the following constraints on the delta coefficients:

val[*j*] = *pVal*[*j*] for ippsDeltaDelta_Aurora function
val[*j*] = 1 for ippsDeltaDeltaMul_Aurora function.

vel[-4] = -1.0, *vel*[-3] = -0.75, *vel*[-2] = -0.5, *vel*[-1] = -0.25, *vel*[0] = 0.0, *vel*[1] = 0.25, *vel*[2] = 0.5, *vel*[3] = 0.75, *vel*[4] = 1.0.

acc[-4] = 1.0, *acc*[-3] = 0.25, *acc*[-2] = -0.285714, *acc*[-1] = -0.607143, *acc*[0] = -0.714286, *acc*[1] = -0.607143, *acc*[2] = -0.285714, *acc*[3] = 0.25, *acc*[4] = 1.0.

The execution mode *deltaMode* provides additional controls for the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to IPP_DELTA_BEGIN|IPP_DELTA_END

Functions perform the offline delta feature calculation. All base features are assumed available at the time of the calculation. The base features are copied from the input stream *pSrc* to the output stream *pDst* as follows:

$pDst[i \cdot dstStep + j] = val[j] \cdot pSrc[i \cdot 14 + j]$, $0 \le j < 12$.

$$pDst[i \cdot dstStep + 12] = val[12] \cdot (pSrc[i \cdot 14 + 12] \cdot 0.6/23 + pSrc[i \cdot 14 + 13] \cdot 0.4)$$

$$0 \le i < height\ . \tag{8.11}$$

Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[max(0, min(i + k, height - 1)) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[max(0, min(i + k, height - 1)) \cdot dstStep + j]$$

for $0 \le i < height,\ 0 \le j < 13$.

2. `deltaMode` is equal to `0`

Online delta feature calculation is performed. The input features are the current segment of a continuous stream. The function `ippsDeltaDelta_Aurora` calculates the partial delta features accordingly. First, the base features are copied as follows:

$$pDst[(i + 8) \cdot dstStep + j] = val[j] \cdot pSrc[i \cdot 14 + j]\ ,\ 0 \le j < 12.$$

$$pDst[(i + 8) \cdot dstStep + 12] = val[12] \cdot (pSrc[i \cdot 14 + 12] \cdot 0.6/23 + pSrc[i \cdot 14 + 13] \cdot 0.4)$$

$$0 \le i < height\ . \tag{8.12}$$

Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[(i + k) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[(i + k) \cdot dstStep + j]$$

for $0 \le i - 4 < height,\ 0 \le j < 13$.

In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the first and

second derivatives in $pDst[k \cdot dstStep + 13 + l]$ are assumed available through a previous derivative calculation for $0 \le j < 13$, $0 \le i < 8$, $0 \le k < 26$, $0 \le l < 4$.

3. *deltaMode* is equal to IPP_DELTA_BEGIN

Functions perform the partial online delta feature calculation, where the beginning of the input stream is known. First, the base features are copied as in equation (8.11). Then, the first and second derivatives are calculated as follows:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[\max(0, i + k) \cdot dstStep + j]$$

and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[\max(0, i + k) \cdot dstStep + j]$$

for $0 \le i < height - 4$, $0 \le j < 13$.

4. *deltaMode* is equal to IPP_DELTA_END

Functions perform the partial online delta feature calculation, where the ending of the input stream is known. First, the base features are copied as in equation (8.12). Then the first and second derivatives are calculated as given by:

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[min(i + k, height - 1) \cdot dstStep + j]$$

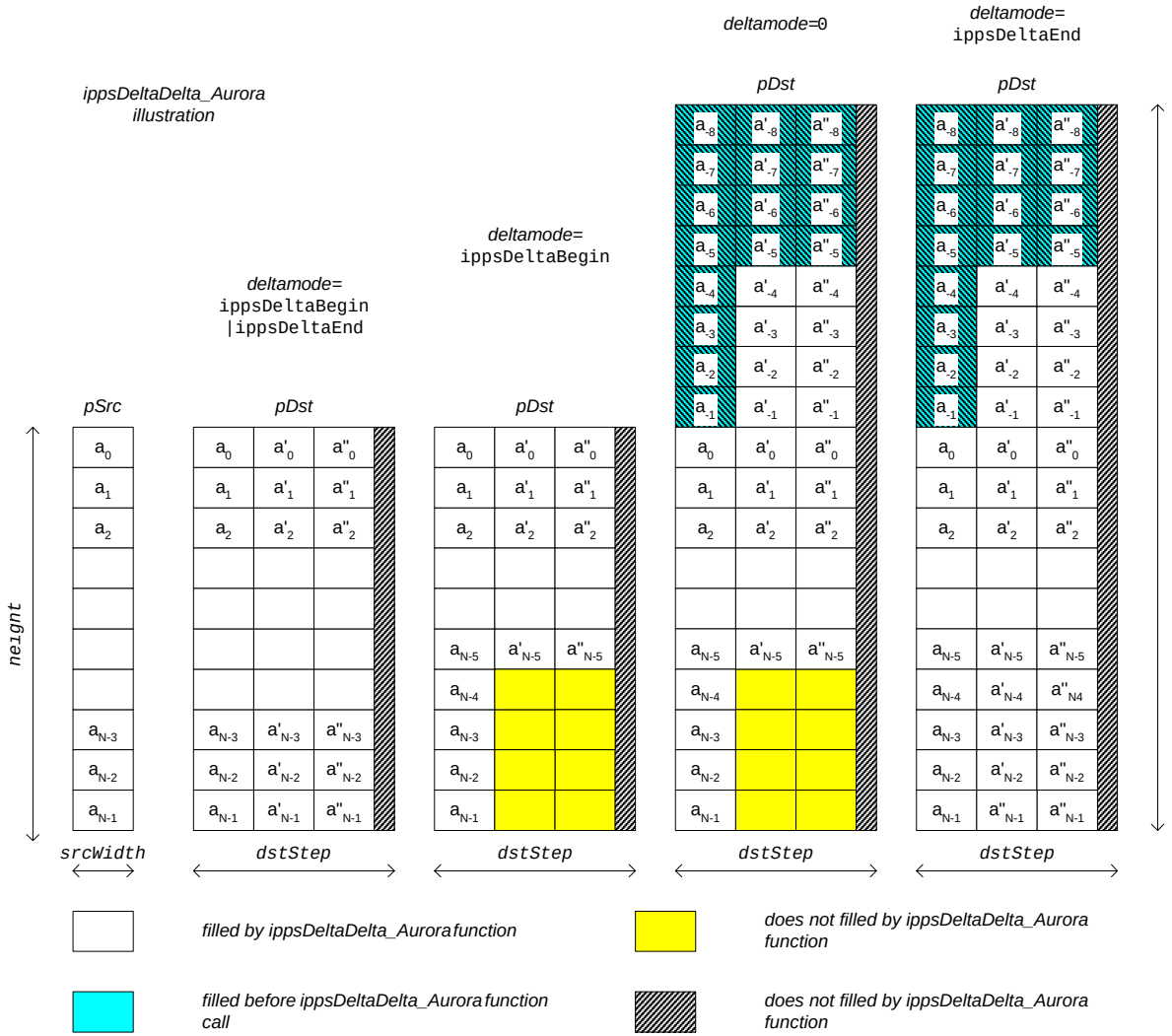
and

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[min(i + k, height - 1) \cdot dstStep + j]$$

for $0 \le i - 4 < height + 4$, $0 \le j < 13$.
In this execution mode, the base features in $pDst[i \cdot dstStep + j]$ and the first and second derivatives in $pDst[k \cdot dstStep + 13 + l]$ are assumed available through a previous derivative calculation for $0 \le j < 13$, $0 \le i < 8$, $0 \le k < 26$, $0 \le l < 4$.

The following figure illustrates the above four delta calculation modes:

Figure 8-6 Execution Modes of ippsDeltaDelta_Aurora function



Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc*, *pDst*, or *pVal* pointer is NULL.
ippStsSizeErr	Indicates an error when *height* is less than or equal to 8 when IPP_DELTA_BEGIN is set; or *height* is less than 0 when IPP_DELTA_BEGIN is not set.
ippStsStrideErr	Indicates an error when *dstStep* is less than 39.

VADGetBufSize_Aurora

Queries the memory size for VAD decision.

```
IppStatus ippsVADGetBufSize_Aurora_32f(int* pSize);
```

Arguments

pSize	Pointer to the output value of the memory size needed for VAD decision.

Discussion

The function ippsVADGetBufSize_Aurora is declared in the ippsr.h file. This function returns the size of memory that should be allocated by user. The memory block of *pSize*[0] bytes is used for VAD algorithm initialization by ippsVadInit_Aurora function.

Return Value

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

VADInit_Aurora

Gets the VAD structure size.

```
IppStatus ippsVADInit_Aurora_32f(char* pVADmem);
```

Arguments

pVADmem	Pointer to the VAD decision memory.

Discussion

The function `ippsVADInit_Aurora` is declared in the `ippsr.h` file. This function initializes the VAD decision process.

Return Value

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

VADDecision_Aurora

Takes the VAD decision.

```
IppStatus ippsVADDecision_Aurora_32f(const Ipp32f* pCoeff, const Ipp32f*
   pTrans, IppVADDecision_Aurora * pRes, int nbSpeechFrame, char* pVADmem);
```

Arguments

pCoeff	Pointer to the input vector of Mel-warped Wiener filter coefficients `[25]`.
pTrans	Pointer to the input vector of Wiener filter transfer function `[64]`.
pRes	Pointer to VAD decision (“1” if voice is detected, “0” otherwise).

nbSpeechFrame	Speech frame hangover counter.
pVADmem	Pointer to the VAD decision memory.

Discussion

The function ippsVADDecision_Aurora is declared in the ippsr.h file. This function takes VAD decision for the input frame according to ([ES202], Annex A).

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCoeff*, *pTrans*, *pRes* or *pVADmem* pointer is NULL.

VADFlush_Aurora

Takes VAD decision for zero input frame.

```
IppStatus ippsVADFlush_Aurora_32f(IppVADDecision_Aurora* pRes, char* pVADmem);
```

Arguments

pCoeff	Pointer to the input vector of Mel-warped Wiener filter coefficients [25].
pTrans	Pointer to the input vector of Wiener filter transfer function [64].
pRes	Pointer to VAD decision (“1” if voice is detected, “0” otherwise).
pVADmem	Pointer to the VAD decision memory.

Discussion

The function ippsVADDecision_Aurora is declared in the ippsr.h file. This function takes VAD decision for the zero frame. It is used when speech is finished.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pCoeff`*, *`pTrans`*, *`pRes`* or *`pVADmem`* pointer is `NULL`.

Ephraim-Malah Noise Suppressor

This section describes the Intel IPP functions that can be combined to construct an implementation of the Ephraim-Malah Noise Suppressor (EMNS) originally described in [Eph84]. The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The EMNS is a frequency domain noise suppression algorithm that seeks to minimize the minimum mean squared error of the speech short-time spectral amplitude estimate.

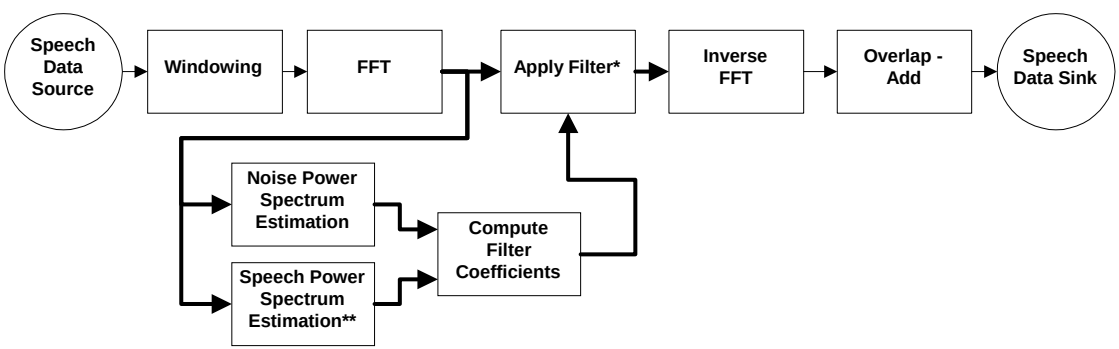
Intel IPP EMNS primitives support the following features:

- Filter update
- Noise floor estimation

Noise Suppressor Architecture

The major building blocks of the Ephraim-Malah Noise Suppressor are shown in Figure 8-7, where the steps are given to apply noise reduction to a speech signal stream.

Figure 8-7 Major blocks in an Ephraim-Malah Noise Suppression System



* applying the filter consists of multiplying each FFT bin by a real-valued filter coefficient

** speech power spectral estimation is typically accomplished via spectral subtraction

Ephraim-Malah Noise Suppressor Details

Algorithm Steps

1. **New input block.** The most recently acquired N/2 input samples ζ_n and the previous N/2 input samples ζ_{n-1} make up the new input block $\mathbf{z}_n$.

$$\mathbf{z}_n = \begin{bmatrix} \zeta_{n-1} \\ \zeta_n \end{bmatrix}$$

2. **Windowed DFT.** The input block is multiplied by the square root of a window function. The window function is constrained such that when its first half is added to its second half, all values add to one. A convenient window function is the triangular window defined as

$$w(m) = \begin{cases} \dfrac{m+0.5}{N/2} & \text{for } m = 0,...,N/2-1 \\ 1-w(m-N/2) & \text{for } m = N/2,...,N-1 \end{cases}$$

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, • denotes point-wise multiplication and $\sqrt{\mathbf{w}}$ denotes a vector containing the square root of the entries of **w**. F is the Fourier transform matrix with entries,

$f(m,n)=\exp(-j2\pi mn/N)$, where N is the size of the transform.

3. **Update noisy speech PSD.** The noisy speech magnitude-squared spectral components are averaged to provide an estimate of the noisy speech power spectrum.

$$\mathbf{P}_n^z(k) = \beta_n \cdot \left|\mathbf{Z}_n(k)\right|^2 + (1-\beta_n)\cdot \mathbf{P}_{n-1}^z(k)$$

The adaptive step size is defined as

$$\beta_n = \beta_{\min} + \rho_{n-1}^y(\beta_{\max} - \beta_{\min})$$

where $\beta_{\min} = 0.9$, $\beta_{\max} = 1.0$, and ρ_{n-1}^{y} is the likelihood of speech presence in bin k.

4. **Update clean speech PSD.** A course estimate of the clean speech power spectral components is obtained by spectral subtraction and averaging.

$$\mathbf{P}_n^y(k) = \alpha_n \cdot \left|\hat{\mathbf{Y}}_{n-1}(k)\right|^2 + (1-\alpha_n)\cdot \psi_0\left(\mathbf{P}_n^z(k) - \mathbf{P}_{n-1}^v(k)\right)$$

The thresholding operator ψ is defined as

$$\psi_c(x) = \begin{cases} c, & x \le c \\ x, & x > c \end{cases}$$

The adaptive step size is defined as

$$\alpha_n = \alpha_{\min} + \left(1 - \rho_{n-1}^{y}\right)\left(\alpha_{\max} - \alpha_{\min}\right)$$

where $\alpha_{\min} = 0.91$, $\alpha_{\max} = 0.95$, and ρ_{n-1}^{y} is the likelihood of speech presence in bin k.

NOTE. *The previous frame's noise power spectral component is used in this calculation. If the noise floor estimator is independent of the rest of the algorithm, it may be possible to use the current frame's noise estimate instead. In this document, a dependency is assumed.*

5. **Update Wiener filter weights.** It turns out that one of the quantities used to compute the Ephraim-Malah suppression rule is actually the Wiener filter (a different noise suppression rule). The Wiener filter weights are given by:

$$\mathbf{W}_n^y(k) = \psi_{W_{\min}}\left(\frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^y(k) + \mathbf{P}_{n-1}^v(k)}\right)$$

where $W_{\min}$ corresponds to the threshold defined in [Cap94]. There it was recommended that a lower limit for a priori SNR

$$\Re_n^{prio}(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_{n-1}^v(k)}$$

of $\Re_{\min dB}^{prio} = -15.0\,\text{dB}$ be imposed to avoid musical noise. This corresponds to

$$W_{\min} = \frac{1}{1 + 10^{-\frac{\Re_{\min dB}^{prio}}{10}}}$$

Note that if the Wiener filter is written in terms of the a priori SNR, then the Wiener filter calculation can be replaced by a table lookup. This may be advantageous on processors where division is expensive.

6. **Update a posteriori SNR.** The a posteriori signal to noise ratio for each frequency bin is defined as follows.

$$\Re_n^{post}(k) = \frac{\mathbf{P}_n^z(k)}{\mathbf{P}_{n-1}^v(k)}$$

7. **Update Ephraim-Malah filter weights.** The Ephraim-Malah filter weights are given by

$$\mathbf{H}_n^y(k) = \frac{1}{\Re_n^{post}(k)} \cdot M\left(\mathbf{W}_n^y(k)\Re_n^{post}(k)\right)$$

where $M(\cdot)$ is the function

$$M(\theta) = \frac{1}{2} \cdot \sqrt{\pi\theta} \cdot e^{-\frac{\theta}{2}} \left[(1+\theta) \cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right) \right]$$

Note that this function can be replaced by a table lookup on processors where Bessel function evaluation is expensive

8. **Update noise PSD estimate.** A noise power spectral estimator must be employed to calculate $\mathbf{P}_n^v(k)$. One such estimator is the method of Martin based on minimum statistics and optimal smoothing [Mar01].

9. **Update speech presence likelihood.** The probability of speech presence is not calculated directly. Rather, it is roughly approximated by the MMSE (Wiener) estimator of the overall speech energy.

$$\rho_n^y = \frac{\sum_{k=0}^{N/2} \mathbf{P}_n^y(k)}{\sum_{k=0}^{N/2} \mathbf{P}_n^y(k) + \sum_{k=0}^{N/2} \mathbf{P}_n^v(k)}$$

10. **Apply additional heuristic modifications to filter coefficients.**

The filter coefficients $\mathbf{H}_n^v(k)$ may be modified to improve perceptual speech quality or reduce perceptible musical tones.For example, to efficiently handle loud, low-pass noise such as that encountered in automotive environments, low-frequency filter coefficients (e.g., below 60 Hz) may be set to zero.

11. **Calculate filter output.** The filter output is defined as follows.

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n^y(k) \cdot \mathbf{Z}_n(k)$$

12. **Inverse DFT and overlap-add.** The time-domain filter output is defined as follows.

$$\hat{\mathbf{y}}_{n-1} = \left[\mathbf{0}_{\frac{N}{2}\times\frac{N}{2}} \quad \mathbf{I}_{\frac{N}{2}\times\frac{N}{2}}\right] \cdot \sqrt{\mathbf{w}} \bullet \mathbf{F}^{-1}\hat{\mathbf{Y}}_{n-1} + \left[\mathbf{I}_{\frac{N}{2}\times\frac{N}{2}} \quad \mathbf{0}_{\frac{N}{2}\times\frac{N}{2}}\right] \cdot \sqrt{\mathbf{w}} \bullet \mathbf{F}^{-1}\hat{\mathbf{Y}}_n$$

Note that he algorithm introduces a delay of N/2 samples in the output.

Data Structures

There is one structure associated with the MCRA noise floor estimator: `IppsMCRAParam`. This structure is used internally and should not be modified by the programmer.

Filter Update Primitives

FilterUpdateEMNS

Calculates the noise suppression filter coefficients.

```
IppStatus ippsFilterUpdateEMNS_32s(const Ipp32s *pSrcWienerCoefsQ31, const Ipp32s
   *pSrcPostSNRQ15, Ipp32s *pDstFilterCoefsQ31, int len);
```

Arguments

pSrcWienerCoefsQ31	Pointer to a real-valued vector containing the Q31 format Wiener filter coefficients ($0.0_{Q31} \leq$ *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).
pDstFilterCoefsQ31	Pointer to a real-valued vector containing the Q31 format filter coefficients ($0.0_{Q31} \leq$ *pDstFilterCoefsQ31*[k] $< 1.0_{Q31}$).

Discussion

The function `ippsFilterUpdateEMNS` is declared in the `ippsr.h` file. This function calculates the noise suppression filter coefficients. Three filter sizes are typically used: 65, 129, and 257 (corresponding to FFT sizes of 128, 256, and 512). These are recommended for sample rates Fs ≤11025 Hz, 11025 Hz < Fs ≤22050 Hz, and

22050 Hz < Fs ≤44100 Hz, respectively. The noise suppression filter coefficients are the gains (scalar values between zero and one) that are applied to each FFT bin. These gains appear in the solution of the equation

$$\min E\left\{\left(\left|\mathbf{Y}_n(k)\right| - \left|\hat{\mathbf{Y}}_n(k)\right|\right)^2\right\}$$

where $\mathbf{Y}_n(k)$ is the *k*-th DFT component corresponding to the *n*-th block of speech samples and $\hat{\mathbf{Y}}_n(k)$ is an estimate of $\mathbf{Y}_n(k)$. Speech spectral components are assumed to have a Gaussian probability density. Since speech spectral components are not directly observable, the solution is written in terms of noisy observations, $\mathbf{Z}_n(k)$. The noise is assumed to be additive and Gaussian. The minimum mean squared error amplitude estimate is

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n(k) \cdot \mathbf{Z}_n(k)$$

where

$$\mathbf{H}_n(k) = \frac{\sqrt{\pi}}{2} \cdot \frac{\sqrt{\mathbf{W}_n^y(k) \cdot \Re_n^{post}(k)}}{\Re_n^{post}(k)} \cdot M\left(\mathbf{W}_n^y(k) \cdot \Re_n^{post}(k)\right)$$

$$M(\theta) = e^{-\frac{\theta}{2}}\left[(1+\theta) \cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right)\right]$$

$$\mathbf{W}_n^y(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^y(k) + \mathbf{P}_{n-1}^v(k)}$$

$$\Re_n^{post}(k) = \frac{\left|\mathbf{Z}_n(k)\right|^2}{\mathbf{P}_{n-1}^v(k)}$$

I_0 and I_1 are Bessel functions, $\mathbf{P}_n^v(k)$ is an estimate of the *k*-th component of the power spectrum of the noise,

$\mathbf{P}_n^y(k)$ is an estimate of the *k*-th speech power spectral component.

The function `ippsFilterUpdate_EMNS` computes the filter coefficients $\mathbf{H}_n(k)$ according to the above equation. Power spectral estimates are typically computed using the `ippsAddWeighted` primitive.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrcWienerCoefsQ31`*, *`pSrcPostSNRQ15`*, or *`pDstFilterCoefsQ31`* pointer is `NULL`.
`ippStsLengthErr`	Indicates an error when *`len`* has an illegal value.

FilterUpdateWiener

Calculates the Wiener filter coefficients.

```
IppStatus ippsFilterUpdateWiener_32s(const Ipp32s *pSrcPriorSNRQ15,Ipp32s
   *pDstFilterCoefsQ31, int len);
```

Arguments

`pSrcPriorSNRQ15`	Pointer to a real-valued Q15 format vector containing an estimate of the a priori signal to noise ratio ($0.0_{Q15} < pSrcPriorSNRQ15[k] < 32768.0_{Q15}$).
`pDstFilterCoefsQ31`	Pointer to a real-valued vector containing the Q31 format filter coefficients ($0.0_{Q31} \le pDstFilterCoefsQ31[k] < 1.0_{Q31}$).
`len`	Number of elements contained in input and output vectors ($0 < len < 65536$).

Discussion

The function `ippsFilterUpdateWiener` is declared in the `ippsr.h` file. This function calculates the Wiener filter coefficients. Three filter sizes are typically used: 65, 129, and 257 (corresponding to FFT sizes of 128, 256, and 512). These are recommended for sample rates Fs ≤11025 Hz, 11025 Hz < Fs ≤2050 Hz, and 22050 Hz < Fs ≤44100 Hz, respectively. The Wiener filter coefficients are the gains (scalar values between zero and one) that are applied to each FFT bin. These gains appear in the solution of the equation

$$\min E\left\{\left|\mathbf{Y}_n(k)-\hat{\mathbf{Y}}_n(k)\right|^2\right\}$$

where $\mathbf{Y}_n(k)$ is the *k*-th DFT component corresponding to the *n*-th block of speech samples and $\hat{\mathbf{Y}}_n(k)$ is an estimate of $\mathbf{Y}_n(k)$. Speech spectral components are assumed to have a Gaussian probability density. Since speech spectral components are not directly observable, the solution is written in terms of noisy observations, $\mathbf{Z}_n(k)$. The noise is assumed to be additive and Gaussian. The minimum mean squared error amplitude estimate is

$$\hat{\mathbf{Y}}_n(k)=\mathbf{W}_n^y(k)\cdot\mathbf{Z}_n(k)$$

where

$$\mathbf{W}_n^y(k)=\frac{1}{1+1/\Re_n^{prio}(k)}$$

$$\Re_n^{prio}(k)=\frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^v(k)}$$

This primitive implements a coarse approximation to $\mathbf{W}_n^y(k)$ in order to achieve very low execution time. This coarse approximation was found to be sufficient for most noise reduction applications. However, if high accuracy is desired one may choose to call `ippsDiv_32s_Sfs` with the proper parameters instead of `ippsFilterUpdateWiener_32s`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrcPriorSNRQ15`* or *`pDstFilterCoefsQ31`* pointer is `NULL`.
`ippStsLengthErr`	Indicates an error when *`len`* has an illegal value.

Noise Floor Estimation Primitives

GetSizeMCRA

Calculates the size in bytes required for the `IppMCRAState` *state structure.*

```
IppStatus ippsGetSizeMCRA_32s(int nFFTSize, int *pDstSize);
```

Arguments

`nFFTSize`	Size of the FFT used for noise PSD estimations (`8` $\leq$ *`nFFTSize`* $\leq$ `8192`).
`pDstSize`	Pointer to the variable to contain the size in bytes.

Discussion

The function ippsGetSizeMCRA is declared in the ippsr.h file. This function calculates the size in bytes that should be allocated for the IppMCRAState state structure required by the ippsUpdateNoisePSDMCRA function. The function ippsGetSizeMCRA should be called before allocating memory and before calling ippsInitMCRA.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDstSize* pointer is NULL.
ippStsSizeErr	Indicates an error when *nFFTSize* has an Illegal value.

InitMCRA

Initializes the IppMCRAState *state structure.*

```
IppStatus ippsInitMCRA_32s_I(int nSamplesPerSec, int nFFTSize, IppMCRAState
   *pDst);
```

Arguments

nSamplesPerSec	Input sample rate ($8000 \le nSamplesPerSec \le 48000$).
nFFTSize	Size of the FFT used to for noise PSD estimation ($8 \le nFFTSize \le 8192$).
pDst	Pointer to the state structure.

Discussion

The function ippsInitMCRA is declared in the ippsr.h file. This function initializes the state structure for the ippsUpdateNoisePSDMCRA function.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates no error when *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *nFFTSize* has an illegal value.
ippStsRangeErr	Indicates an error when *nSamplesPerSec* is out of range.

NOTE. *State memory address stored in pDst must be aligned to 32-bit word boundary.*

InitAllocMCRA

Allocates memory and initializes the IppMCRAState *state structure.*

```
IppStatus ippsInitAllocMCRA_32s_I(int nSamplesPerSec, int nFFTSize, IppMCRAState
**ppDst);
```

Arguments

nSamplesPerSec	Input sample rate ($0 < nSamplesPerSec \leq 48000$).
nFFTSize	Size of the FFT used to for noise PSD estimation ($8 \leq nFFTSize \leq 8192$).
ppDst	Pointer to the pointer to the state structure.

Discussion

The function ippsInitAllocMCRA is declared in the ippsr.h file. This function allocates memory and initializes the state structure for the ippsUpdateNoisePSDMCRA function.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *nFFTSize* has an illegal value.
ippStsRangeErr	Indicates an error when *nSamplesPerSec* is out of range.

UpdateNoisePSDMCRA

Re-estimates the noise power spectrum.

```
IppStatus ippsUpdateNoisePSDMCRA_32s_I(const Ipp32s *pSrcNoisySpeech,
   IppMCRAState *pSrcDstState, Ipp32s *pSrcDstNoisePSD);
```

Arguments

pSrcNoisySpeech	Pointer to a real-valued vector containing the magnitude squared of the FFT of the noisy speech ($0 \leq$ *pSrcNoisySpeech*[k] $< 2^{31}$).
pSrcDstState	Pointer to the state structure.
pSrcDstNoisePSD	Pointer to the noise power spectrum vector.

Discussion

The function ippsUpdateNoisePSDMCRA is declared in the ippsr.h file. This function re-estimates the noise power spectrum given a new measurement of the magnitude squared noisy speech. The algorithm is based on the Minima Controlled Recursive Averaging approach described in [Coh02].

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcNoisySpeech*, *pSrcDstState*, or *pSrcDstNoisePSD* pointer is NULL.

Acoustic Echo Canceller

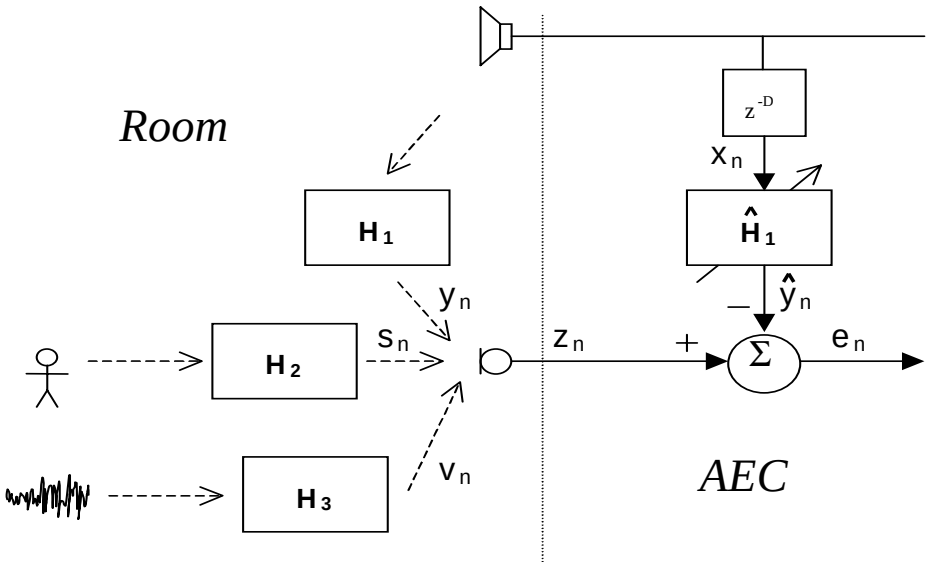
This section describes the Intel IPP functions that can be combined to construct an Acoustic Echo Canceller (AEC). The primitives are primarily concerned with the well-defined, computationally expensive core operations.
The AEC consists of a frequency-domain adaptive filter algorithm and controller. IPP AEC primitives support the following features:

- Filtering
- Filter coefficient
- Step size update
- AEC controller

Acoustic Echo Canceller Architecture

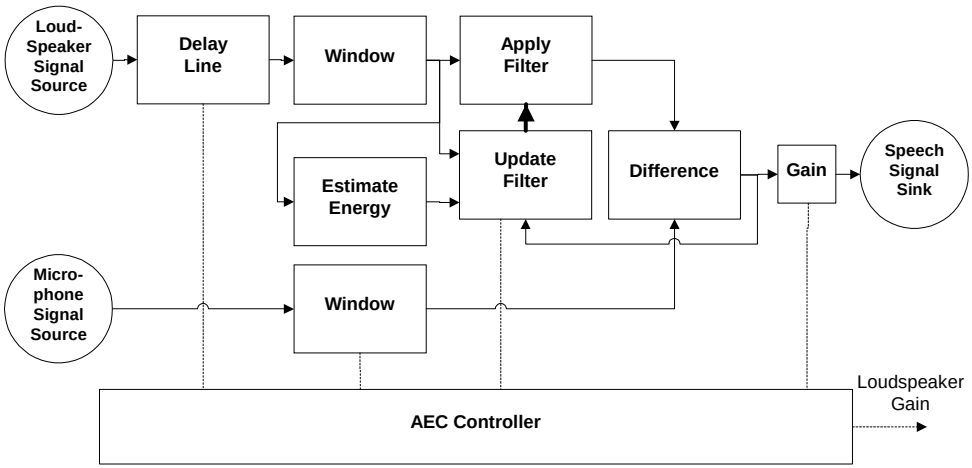
Figure 8-8 shows a typical application of acoustic echo cancellation, the monophonic speakerphone. Here, audio is played into a room through a loudspeaker and captured at a microphone. The acoustic transfer function between the loudspeaker and the microphone is denoted $\mathbf{H}_1$ and the resulting signal at the microphone is denoted 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 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 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 may be subtracted from the microphone signal, cancelling the echo.

Figure 8-8 A typical Acoustic Echo Cancellation Scenario



The major building blocks of the Acoustic Echo Canceller are shown in Figure 8-9, where the steps are given to apply AEC to a speech signal stream.

Figure 8-9 Major blocks in an Acoustic Echo Cancellation System



Frequency Domain Block 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 [Ash94]. The algorithm consists of the following steps:

1. **New input block**. The most recently acquired N/2 input samples χ_n and the previous N/2 input samples χ_{n-1} make up the new input block. $\mathbf{x}_n$

$$\mathbf{x}_n = \begin{bmatrix} \chi_{n-1} \\ \chi_n \end{bmatrix}$$

2. **Input DFT**. The discrete Fourier transform of the input is calculated.

$$\mathbf{X}_n = \mathbf{F}\mathbf{x}_n$$

3. **Update input history.** The most recent L frequency-domain input blocks are retained.

$$\mathrm{X}_n = [\mathbf{X}_n \mathbf{X}_{n-1} \cdots \mathbf{X}_{n-L+1}]$$

4. **Update input energy estimate.** The energy at each bin $P_{xx}(k)$ may be estimated using a leaky average with $0 < \beta < 1$.

$$\hat{P}_{xx}(k) = (1-\beta) \cdot \hat{P}_{xx}(k) + \beta \cdot |\mathbf{X}_n(k)|^2 \quad \text{for } k = 0,..., N/2$$

5. **Compute adaptive step sizes.** The adaptive step sizes are computed using the classical normalized LMS approach with $0 < \mu << 1$ and $P_{x\min} > 0$.

$$\mathrm{M}(k) = \begin{cases} \dfrac{\mu}{\hat{P}_{xx}(k)}, & \hat{P}_{xx}(k) > P_{x\min} \\ \dfrac{\mu}{P_{x\min}}, & \hat{P}_{xx}(k) \le P_{x\min} \end{cases} \quad \text{for } k = 0,..., N/2$$

6. **Compute the filter output.** The adaptive filter uses the low-latency structure described in [Ash94] where 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

$$y(m) = \sum_{j=0}^{J-1} \sum_{k=0}^{I/J-1} h(k + j(I/J)) \cdot x(m - j(I/J) - k) = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} h_j(k) \cdot x_j(m-k)$$

The filter output may be computed in the frequency domain as

$$\hat{\mathbf{Y}}_n(k) = \sum_{i=0}^{L-1} \mathbf{X}_{n-i}(k) \cdot \mathbf{H}_i(k) \text{ for } k = 0,..., N/2$$

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.

$$\mathbf{e}_n = \mathbf{y}_n - \hat{\mathbf{y}}_n$$

The time-domain error must be constrained as follows

$$\mathbf{E}_n = \mathbf{F}\begin{bmatrix} \mathbf{0} \\ \boldsymbol{\varepsilon}_n \end{bmatrix}$$

to properly implement the adaptation step (see step 8) in the frequency-domain.

The n^{th} block of the echo canceller output is defined here as

$$\boldsymbol{\varepsilon}_n = [\mathbf{e}_n(N/2) \cdots \mathbf{e}_n(N-1)]^T$$

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

$$\mathbf{H}_i = \mathbf{H}_i + \mathbf{M}_n \bullet \mathbf{X}_{n-i}^* \bullet \mathbf{E}_n \text{ for } i = 0,..., L-1$$

where $\bullet$ denotes pointwise multiplication. Here $\mathbf{X}_{n-i}^{*} \bullet \mathbf{E}_{n}$ may be 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

$$\mathbf{h}_i = \mathbf{F}^{-1}\mathbf{H}_i$$

$$\mathbf{h}_i = \mathbf{F}^{-1}\mathbf{H}_i$$

for $i = 0,\ldots, L - 1$ .Here, the first half of each segment of the impulse response

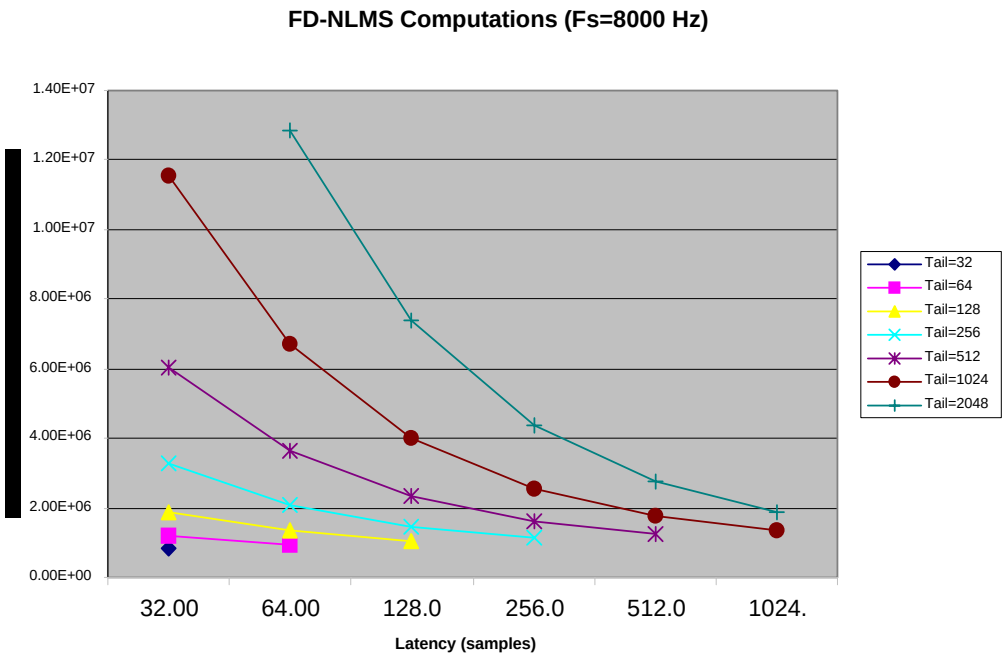
$$\boldsymbol{\eta}_i = \left[\mathbf{h}_i(0)\cdots\mathbf{h}_i(N/2-1)\right]^T$$

is retained and the rest set to zero.

Computational Complexity

The computational complexity of the adaptive filter is illustrated in Figure 8-10 Here the trade-off between low latency and filter tail length are illustrated. 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 around 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 8-10 Computational complexity of FD-NLMS adaptive filter



AEC Controller

To understand the need for an AEC controller, consider Figure 8-8.

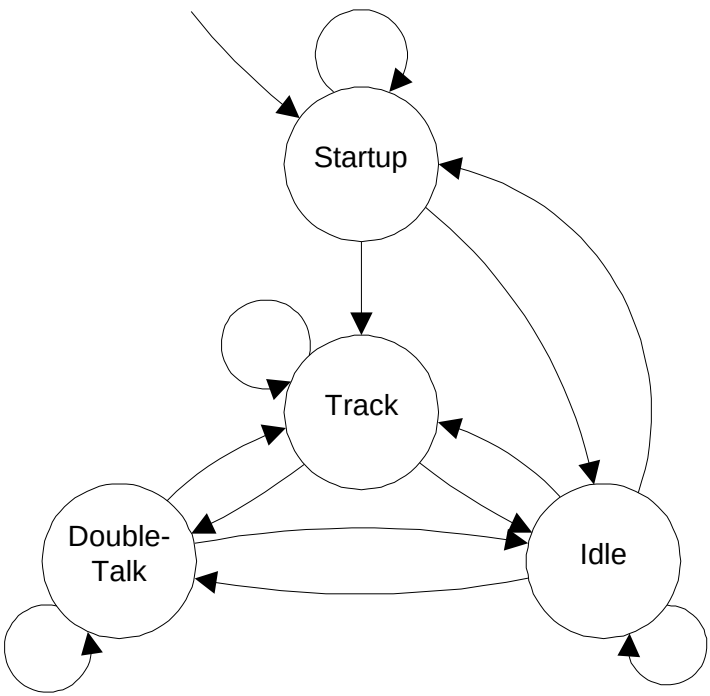
The adaptive filter $\hat{H}_1$ modifies its coefficients such that the filtered loudspeaker signal $\hat{y}_n$ best matches the microphone signal z_n in the least squares sense. When the loudspeaker signal is active, s_n and v_n are weak, and the loudspeaker-to-microphone transfer function H_1 is nearly 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 above) 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.

Algorithm Description

The AEC controller update primitive `ippsControllerUpdateAEC` is based on the simple state machine shown in Figure 8-11

Figure 8-11 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 (e.g., 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^x is $6.0 \cdot \min\{E_n^z, E_{n-1}^z, ..., E_{n-windowsize}^z\}$ (8 dB above the local minimum), and the external gain between the loudspeaker and microphone G_{xz} is 0.25 (-6 dB). The minimum-tracking window size corresponds to 2 seconds.

State transitions are triggered by the four conditions according to the table below. The step size, loudspeaker gain, and output gain are adjusted as shown.

C	R	M	N	Current State	Next State	μ	α	β
F	T	T	T	Startup	Startup	4“_c	0.0	1.0
F	T	F	X	Startup	Startup	“_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“_c	1.0	1.0
T	T	T	F	Startup	Track	“_c	1.0	1.0
T	T	F	T	Startup	Track	“_c	0.0	1.0
T	T	F	F	Startup	Idle	0.0	0.5	0.5
T	T	T	X	Track	Track	2“_c	1.0	1.0
X	T	F	X	Track	Track	“_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

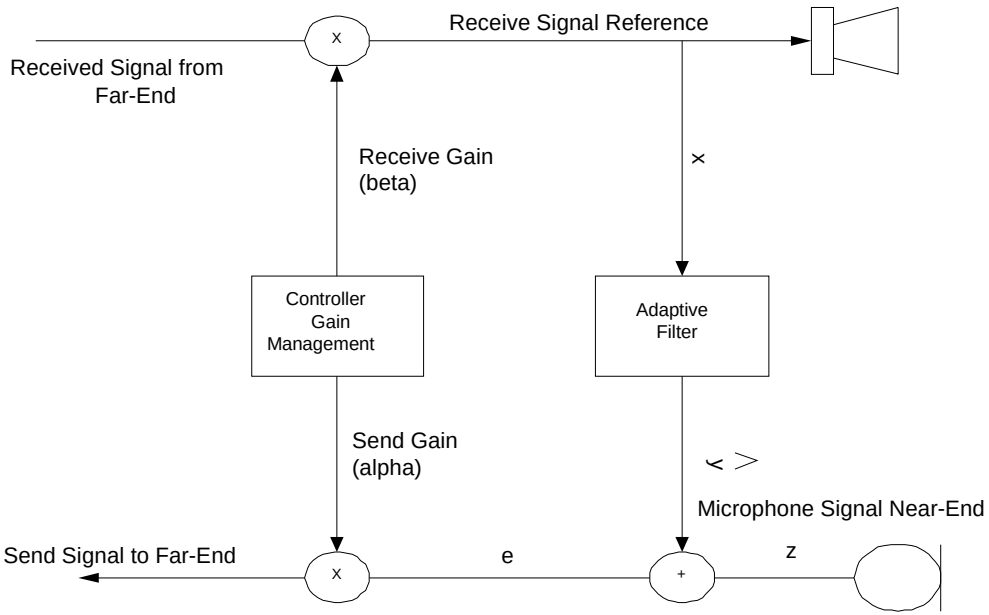
X	F	T	X	Idle	Idle	0.0	1.0	1.0
X	F	F	X	Idle	Idle	0.0	0.5	0.5
T	T	T	T	Idle	Track	2“_c	1.0	1.0
T	T	T	F	Idle	Track	2“_c	1.0	1.0
X	T	F	X	Idle	Track	2“_c	1.0	1.0
X	T	T	F	Idle	Double-talk*	0.0	1.0	1.0
F	T	T	X	Double-talk	Double-talk	0.0	1.0	1.0
T	T	T	T	Double-talk	Track	2“_c	1.0	1.0
T	T	T	F	Double-talk	Track	“_c	1.0	1.0
X	T	F	X	Double-talk	Track	“_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

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 and receive 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 8-12 illustrates that application of the send and receive gain in an AEC system.

Figure 8-12 Application of AEC send and receive gains by the AEC controller.



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;
```

Note that `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 the table below.

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 [32 , 4096]
sampleRate	In range [8000 , 48000]

Filter tail lengths of up to 2 seconds are supported. 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`.

The `IppAECCtrlState` structure is used internally and should not be modified by the programmer.

The structure IppAECScaled32s provides a representation for scaled 32-bit signed integers.

```
typedef struct {
    Ipp32s val;
    Ipp32s sf;
} 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® PCA processor family 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 $2^{30} \le x.val < 2^{31}$ or $-2^{30} < x.val \le -2^{31}$.

Filter Primitives

FilterAECNLMS

Computes the frequency-domain adaptive filter output.

```
IppStatus ippsFilterAECNLMS_32sc_Sfs(const Ipp32sc **ppSrcSignalIn,
   const Ipp32sc **ppSrcCoefs, Ipp32sc *pDstSignalOut, int numSegments,
   int len, int scaleFactor);
```

Arguments

ppSrcSignalIn	Pointer to an array of pointers to the most recent input blocks(e.g., X_n, X_{n-1}, …, X_{n-L+1}). These are the complex-valued vectors that contain the FFT of the input signal. The argument *ppSrcSignalIn* is a two-dimensional complex vector of size [*numSegments*][*len*].
ppSrcCoefs	Pointer to an array of pointers to the filter coefficients vectors.These are the complex-valued vectors containing the filter coefficients. The argument *ppSrcCoefs* is a two-dimensional complex vector of size [*numSegments*][*len*].

pDstSignalOut	Pointer to the complex-valued filter output vector.
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)

Discussion

The function ippsFilterAECNLMS is declared in the ippsr.h file. This function implements step 6 of the above algorithm to compute the frequency-domain adaptive filter output.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *ppSrcSignalIn*, *ppSrcCoefs*, or *pDstSignalOut* pointer is NULL.
ippStsLengthErr	Indicates an error when *len* has an illegal value.
ippStsRangeErr	Indicates an error when *numSegments* or *scaleFactor* is out of range.

Filter Update Primitives

CoefUpdateAECNLMS

Updates the adaptive filter coefficients.

```
IppStatus ippsCoefUpdateAECNLMS_32sc_I(const IppAECScaled32s*pSrcStepSize,
   const Ipp32sc **ppSrcFilterInput, const Ipp32sc *pSrcError,
   Ipp32sc **ppSrcDstCoefs, int numSegments, int len, int scaleFactorCoef);
```

Arguments

pSrcStepSize	Pointer to the scaled integer step size vector. It is recommended that the `val` field is left justified, that is, 230 ≤ *pSrcStepSize*[k].`val` < 231 and the `sf` field is restricted to the range (-64 ≤ *pSrcStepSize*[k].`sf` < 64). The dimension of *pSrcStepSize* is [*len*].
ppSrcFilterInput	Pointer to an array of pointers to the most recent input blocks (e.g., X_n, X_{n-1}, …, X_{n-L+1}). These are the complex-valued vectors that contain the FFT of the input signal. The dimension of *ppSrcFilterInput* is [*numSegments*][*len*].
pSrcError	Pointer to the complex-valued vector containing the filter error. The dimension of *pSrcError* is [*len*].
ppSrcDstCoefs	Pointer to an array of pointers to the filter coefficient vectors. These are the complex-valued vectors containing the filter coefficients. The dimension of *ppSrcDstCoefs* is [*numSegments*][*len*].
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` and `ippsFilterAECNLMS` functions. The filter coefficients are assumed to have been multiplied by $2^{scaleFactorCoef}$ prior to the call to this function. The result of the update is saturated if overflow occurs.

Discussion

The function `ippsCoefUpdateAECNLMS` is declared in the `ippsr.h` file. This function implements step 8 of the above algorithm to update the adaptive filter coefficients.The scale factor is applied only to the additive update term

" $+\mathbf{M}_n \bullet \mathbf{X}^*_{n-i} \bullet \mathbf{E}_n$ ".

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`ppSrcStepSize`*, *`ppSrcFilterInput`*, *`pSrcError`*, or *`ppSrcDstCoefs`* pointer is `NULL`.
`ippStsLengthErr`	Indicates an error when *`len`* has an illegal value.
`ippStsRangeErr`	Indicates an error when *`pSrcStepSize`*[i]`.val`<0, or when *`numSegments`* or *`scaleFactorCoef`* is out of range.

Step Size Update Primitives

StepSizeUpdateAECNLMS

Computes the adaptive step size.

```
IppStatus ippsStepSizeUpdateAECNLMS_32s(const Ipp32s *pSrcInputPSD,
   Ipp32s muQ31, IppAECScaled32s maxStepSize, Ipp32s minInputPSD,
   IppAECScaled32s *pDstStepSize, int len);
```

Arguments

`pSrcInputPSD`	Pointer to real-valued vector containing the input power spectrum estimate.

muQ31	Real-valued Q31 scalar ($0.0_{Q31} \leq$ *muQ31* $< 1.0_{Q31}$).
maxStepSize	Left justified scaled integer representing the maximum step size allowed (usually this is equal to *mu* / *minInputPSD* > 0).
minInputPSD	Minimum value of input PSD for which the step size update should occur (0 < *minInputPSD*).
len	Number of elements contained in input and output vectors (0 < *len* ≤ 4097).
pDstStepSize	Pointer to the left justified scaled integer output vector.

Discussion

The function ippsStepSizeUpdateAECNLMS is declared in the ippsr.h file. This function implements step 5 of the above algorithm to compute the adaptive step size.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcInputPSD* or *pDstStepSize* pointer is NULL.
ippStsLengthErr	Indicates an error when *len* has an illegal value.
ippStsRangeErr	Indicates an error when *pSrcInputPSD*[i]< 0, *smuQ31*< 0, *minInputPSD*< 0, or *maxStepSize*.val< 0 .

AEC Controller Primitives

ControllerGetSizeAEC

Returns the size of the AEC controller state structure.

```
IppStatus ippsControllerGetSizeAEC_32s(int *pDstSize);
```

Arguments

pDstSize	Pointer to variable that will hold the size in bytes of the state.

Discussion

The function ippsControllerGetSizeAEC is declared in the ippsr.h file. This function returns the size in bytes of the AEC controller state structure, so that the correct amount of memory can be allocated.

Return Values

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

ControllerInitAEC

Initializes the AEC controller state structure.

```
IppStatus ippsControllerInitAEC_32s(const IppAECNLMSParam *pSrcParams,
   IppAECCtrlState *pDstState);
```

Arguments

pSrcParams	Pointer to AEC parameters structure.
pDstState	Pointer to the AEC controller state structure.

Discussion

The function ippsControllerInitAEC is declared in the ippsr.h file. This function initializes the AEC controller state structure.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcParams* or *pSrcDstState* pointer is NULL.

ippStsRangeErr	Indicates an error when *pSrcParams->numSamples* or *pSrcParams->sampleRate* is out of range (see Data Structures for description of range).

NOTE. *State memory address stored in pDstState must be aligned to 32-bit word boundary.*

ControllerUpdateAEC

Implements the energy-based AEC controller.

```
IppStatus ippsControllerUpdateAEC_32s(const IppAECNLMSParam *pSrcParams,
   IppAECCtrlState *pSrcDstState, Ipp32s *pDstMuQ31,
   Ipp32s *pDstAECOutGainQ30, Ipp32s *pDstSpeakerGainQ30);
```

Arguments

pSrcParams	Pointer to AEC parameters structure.
pSrcDstState	Pointer to AEC controller state structure.
pDstMuQ31	Pointer to real-valued Q31 scalar ($0.0_{Q31} \le$ *pDstMuQ31* $< 1.0_{Q31}$).
pDstAECOutGainQ30	Pointer to real-valued Q30 scalar ($0.0_{Q30} \le$ *pDstAECOutGainQ31* $\le 1.0_{Q30}$).
pDstSpeakerGainQ30	Pointer to real-valued Q30 scalar ($0.0_{Q30} \le$ *pDstSpeakerGainQ31* $\le 1.0_{Q30}$).

Discussion

The function ippsControllerUpdateAEC is declared in the ippsr.h file. This function implements the energy-based AEC controller described in Algorithm Description section. 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" (that is, the condition where a person is speaking into the microphone while audio is playing through the loudspeaker), the adaptation step size is reduced.

NOTE. *The* `ippsControllerUpdateAEC_32s` *primitive is intended to provide basic AEC control functionality. 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 this controller is to provide baseline AEC performance during the prototyping stage. It is expected that this controller will be replaced with a custom controller in a final product. Limitations of this controller include the following:*

- *does not maintain adaptive filter convergence for double-talk durations of more than about 2 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.*

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrcParams`*, *`pSrcDstState`*, *`pDstMuQ31`*, *`pDstAECOutGainQ30`*, or *`pDstSpeakerGainQ30`* pointer is `NULL`.

Voice Activity Detector

This section describes the Intel IPP functions that can be combined to construct a Voice Activity Detector (VAD). The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The VAD consists of parameter estimation and speech modeling heuristics.

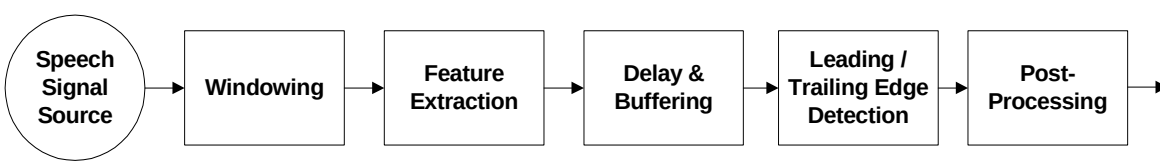
IPP VAD primitives support the following features:

- Peak picking
- Periodicity
- Zero crossing rate

Voice Activity Detector Architecture

A typical voice activity detector is illustrated below. The speech signal is windowed and separated into (possibly overlapping) frames. Feature extraction is then performed. Feature extraction may include one or more of the following: zero crossing rate calculation, energy calculation, segmental SNR estimation, periodicity detection, and sample or sub-band histogram measurement. Measurements are buffered so that speech onset and ending can be distinguished from other spurious events. A delay of 400 milliseconds is typical. Speech leading and trailing edge detection is carried out based on heuristic rules. Post-processing (e.g., median filtering) may be applied in low-delay implementations to eliminate false detections.

Figure 8-13 Major blocks in a Voice Activity Detector



Voice Activity Detection Primitives

FindPeaks

Identifies peaks in the input vector.

```
IppStatus ippsFindPeaks_32s8u(const Ipp32s *pSrc, Ipp8u *pDstPeaks, int len,
   int searchSize, int movingAvgSize);
```

Arguments

pSrc	Pointer to an input vector.
pDstPeaks	Pointer to the output vector containing a one in positions corresponding to a peak in the input vector and zeros elsewhere.
len	Number of elements contained in the input and output vectors ($0 <$ *len* < 65536).
searchSize	Number of elements on either side to consider when picking peak ($0 <$ *searchSize* < 128).
movingAvgSize	Number of elements on either side to include in moving average window that is applied before peak picking ($0 \leq$ *movingAvgSize* < 128).

Discussion

The function ippsFindPeaks is declared in the ippsr.h file. This function identifies peaks in the input vector, places a one in the output vector at the locations of the peaks, and places a zero elsewhere.
A peak is defined as a point *pSrc*[i] such that *pSrc*[i-L] < *pSrc*[i-L+1] < … < *pSrc*[i-1] < *pSrc*[i] > … > *pSrc*[i+L-1] > *pSrc*[i+L] , where *searchSize* is L.
If *movingAvgSize* is greater than 0, then the source vector is smoothed via moving average before peaks are selected.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDstPeaks* pointer is NULL.
ippStsLengthErr	Indicates an error when *len* is out of range.
ippStsSizeErr	Indicates an error when *SearchSize* or *movingAvgSize* is out of range.

PeriodicityLSPE

Computes the periodicity of the input speech frame.

```
IppStatus ippsPeriodicityLSPE_16s(const Ipp16s *pSrc, int len, Ipp16s
   *pPeriodicityQ15, int *period, int maxPeriod, int minPeriod);
```

Arguments

pSrc	Pointer to an input speech vector.
len	Number of elements contained in the input vector subject to $6 < len \leq \min(16*minPeriod, 1024)$
pPeriodicityQ15	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ($0.0_{Q15} \leq pPeriodicityQ15 \leq 1.0_{Q15}$).
period	The period (in samples) that minimizes the LSPE cost function.
maxPeriod	Maximum period to search ($minPeriod < maxPeriod < len$).
minPeriod	Minimum period to search ($6 \leq minPeriod < maxPeriod$).

Discussion

The function ippsPeriodicityLSPE is declared in the ippsr.h file. This function computes the periodicity of the input speech frame. The periodicity is calculated according the least squares periodicity estimate (LSPE) algorithm defined in [Tuc92]. The periodicity search is designed for speech signals with sample rate between 2000-24000 Hz and may not perform well on music or other sources. If only periodicity (voicing) is required, it is more efficient to downsample input speech before calling this primitive. For example, at 2000 Hz sample rate, the settings *minPeriod*=10, *maxPeriod*=20, and *len*=64, provide adequate periodicity for some applications.

Return Values

ippStsNoErr	Indicates no error.
ippStsLengthErr	Indicates an error when *len* is out of range.
ippStsRangeErr	Indicates an error when *maxPeriod* or *minPeriod* is out of range.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pPeriodicityQ15* pointer is NULL.

Periodicity

Computes the periodicity of the input block.

```
IppStatus ippsPeriodicity_32s16s(const Ipp32s *pSrc, int len, Ipp16s
   *pPeriodicityQ15, int *period, int maxPeriod, int minPeriod);
```

Arguments

pSrc	Pointer to an input vector containing non-negative entries.
len	Number of elements contained in the input vector ($0 < len \leq 4096$).

pPeriodicityQ15	Pointer to the Q15 format value corresponding to the normalized sum of the largest harmonic sampling ($0.0_{Q15} \le$ *pPeriodicityQ15* $\le 1.0_{Q15}$).
period	Pointer to the period (in samples) that provided the maximum-energy harmonic sampling.
maxPeriod	Maximum period to search (*minPeriod* < *maxPeriod* < *len*).
minPeriod	Minimum period to search (0 < *minPeriod* < *maxPeriod*).

Discussion

The function `ippsPeriodicity` is declared in the `ippsr.h` file. This function computes the periodicity of the input block. In typical applications, the input block is the magnitude-squared of the discrete Fourier transform of windowed speech. The periodicity is defined as the periodic sampling of the input block that preserves the most energy.

$$\max_{k,T_0} \sum_{n} x(k + nT_0) \quad \text{where } 0 < k \le T_0$$

Bias removal is performed prior to the search to ensure an accurate measurement.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsLengthErr`	Indicates an error when *len* is out of range.
`ippStsRangeErr`	Indicates an error when *pSrc*[k], *maxPeriod*, or *minPeriod* is out of range.
`ippStsNullPtrErr`	Indicates an error when *pSrc*, *period*, or *pPeriodicityQ15* pointer is `NULL`.

Compatibility with version 1.1

Some function names of the speech recognition primitives in Intel IPP version 1.1 are deprecated for a uniform naming convention of Intel IPP version 2. The following table lists the deprecated functions in the left column and the renamed functions in the right column. For compatibility, the use of those deprecated functions is supported (but not recommended) in Intel IPP version 2.

Table 8-2 Intel IPP 2.0 Renamed Functions List

Deprecated Functions	New Function Names
ippsAddNRow	ippsAddNRows
ippsSumCol	ippsSumColumn
ippsSumAllRow	ippsCopyColumn
ippsCopyCol	ippsCopyColumn
ippsMeanCol	ippsMeanColumn
ippsVarCol	ippsVarColumn
ippsMeanVarCol	ippsMeanVarColumn
ippsNormalizeCol	ippsNormalizeColumn
ippsAddMulCol	ippsAddMulColumn
ippsQRTransCol	ippsQRTransColumn
ippsDotProdCol	ippsDotProdColumn
ippsMulCol	ippsMulColumn
ippsSumColAbs	ippsSumColumnAbs
ippsSumColSqr	ippsSumColumnSqr
ippsDeltaW1	ippsDelta_Win1
ippsDeltaW2	ippsDelta_Win2
ippsDeltaDeltaW1	ippsDeltaDelta_Win1
ippsDeltaDeltaW2	ippsDeltaDelta_Win2
ippsRecSqrt_Th	ippsRecSqrt
ippsLMThreshold	ippsScaleLM
ippsMahDist1	ippsMahDist
ippsMahDist2	ippsMahDist_MultiMix
ippsLogGauss1	ippsLogGauss

Table 8-2 Intel IPP 2.0 Renamed Functions List

Deprecated Functions	New Function Names
ippsLogGauss2	ippsLogGauss_MultiMix
ippsLogGaussMax1	ippsLogGaussMax
ippsLogGaussAdd1	ippsLogGaussAdd
ippsLogGaussAdd2	ippsLogGaussAdd_MultiMix

Compatibility with version 2.0

A number of functions in Intel IPP version 2.0 were replaced by functions with extended functionality and are deprecated for Intel IPP version 3. The following table lists the deprecated functions in the left column and the new functions in the right column. For compatibility, the use of those deprecated functions is supported (but not recommended) in Intel IPP version 3. The deprecated functions are equivalent to Intel IPP 3.0 functions with *scaleFactor* argument set to 0.

Table 8-3 Intel IPP 3.0 Replaced Functions List

Deprecated Functions	New Function Names
ippsLogGaussSingle_16s32f	ippsLogGaussSingle_Scaled_16s32f
ippsLogGaussSingle_DirectVar_16s32f	ippsLogGaussSingle_DirectVarScaled_16s32f
ippsLogGaussSingle_IdVar_16s32f	ippsLogGaussSingle_IdVarScaled_16s32f
ippsLogGaussSingle_BlockDVar_16s32f	ippsLogGaussSingle_BlockDVarScaled_16s32f
ippsLogGauss_16s32f_D2	ippsLogGauss_Scaled_16s32f_D2
ippsLogGauss_16s32f_D2L	ippsLogGauss_Scaled_16s32f_D2L
ippsLogGauss_IdVar_16s32f_D2	ippsLogGauss_IdVarScaled_16s32f_D2
ippsLogGauss_IdVar_16s32f_D2L	ippsLogGauss_IdVarScaled_16s32f_D2L
ippsLogGaussMultiMix_16s32f_D2	ippsLogGaussMultiMix_Scaled_16s32f_D2
ippsLogGaussMultiMix_16s32f_D2L	ippsLogGaussMultiMix_Scaled_16s32f_D2L
ippsLogGaussMax_16s32f_D2	ippsLogGaussMax_Scaled_16s32f_D2
ippsLogGaussMax_16s32f_D2L	ippsLogGaussMax_Scaled_16s32f_D2L
ippsLogGaussMax_IdVar_16s32f_D2	ippsLogGaussMax_IdVarScaled_16s32f_D2
ippsLogGaussMax_IdVar_16s32f_D2L	ippsLogGaussMax_IdVarScaled_16s32f_D2L
ippsLogGaussMaxMultiMix_16s32f_D2	ippsLogGaussMaxMultiMix_Scaled_16s32f_D2

Table 8-3 Intel IPP 3.0 Replaced Functions List

Deprecated Functions	New Function Names
ippsLogGaussMaxMultiMix_16s32f_D2L	ippsLogGaussMaxMultiMix_Scaled_16s32f_D2L

The following Intel IPP version 2 functions are renamed in Intel IPP version 3 to follow the uniform naming style. For compatibility, the use of those deprecated functions is supported (but not recommended) in Intel IPP version 3.

Table 8-4 Intel IPP 3.0 Renamed Functions List

Deprecated Functions	New Function Names
ippsLogGaussAdd_16s32f_D2	ippsLogGaussAdd_Scaled_16s32f_D2
ippsLogGaussAdd_16s32f_D2L	ippsLogGaussAdd_Scaled_16s32f_D2L
ippsLogGaussAdd_IdVar_16s32f_D2	ippsLogGaussAdd_IdVarScaled_16s32f_D2
ippsLogGaussAdd_IdVar_16s32f_D2L	ippsLogGaussAdd_IdVarScaled_16s32f_D2L
ippsLogGaussAddMultiMix_16s32f_D2	ippsLogGaussAddMultiMix_Scaled_16s32f_D2
ippsLogGaussAddMultiMix_16s32f_D2L	ippsLogGaussAddMultiMix_Scaled_16s32f_D2L
ippsLPToLSP_32f_D2	ippsLPToLSP_32f
ippsLPToLSP_16s_D2Sfs	ippsLPToLSP_16s_Sfs
ippsLSPToLP_32f_D2	ippsLSPToLP_32f
ippsLSPToLP_16s_D2Sfs	ippsLSPToLP_16s_Sfs

Speech Coding Functions 9

This chapter describes Intel® IPP functions that can be used for implementing speech codecs which follow ITU-T recommendations G.729, G.723.1, G.722.1, G.726 and G.728, as well as GSM-AMR and GSM-FR codecs.
When properly built, such speech codecs can be compliant with the bit-exact specifications for published test vectors.
The organization of this chapter is the following. The introductory part includes description of rounding modes, defines notational conventions, header files and data structures used by Intel IPP speech coding functions. This is succeeded by the major functionality sections, which include:

- Common Functions
- G.729 Related Functions
- G.723.1 Related Functions
- GSM-AMR Related Functions
- GSM Full Rate Related Functions
- G.722.1 Related Functions
- G.726 Related Functions
- G.728 Related Functions

Each section starts with the table that gives the full list of Intel IPP functions specific for that functional group, followed by detailed description of the respective API.

Rounding mode

As many speech codecs have to meet the bit-exact requirement, Intel IPP functions described in this chapter use rounding modes that are different from the default rounding mode used in general signal processing functions.
For general signal processing functions, the default rounding mode can be described as

"nearest even", so that the fixed point number $x = N + \alpha,\ 0 \le \alpha < 1$, where N is an integer number, is rounded as given by:

$$\lceil x \rceil = \begin{cases} N\ ,\ 0 \le \alpha < 0.5 \\ N+1\ ,\ 0.5 < \alpha < 1 \\ N\ ,\ \alpha = 0.5\ ,\ N-even \\ N+1\ ,\ \alpha = 0.5\ ,\ N-odd \end{cases}$$

For example, 1.5 will be rounded to 2 and 2.5 to 2.

For functions in this chapter, there are two rounding modes.

The default rounding mode is "clipping", so that the fractional part of the fixed point number is cut off and the result of rounding is always less than the initial value. Specifically, the fixed point number $x = N + \alpha,\ 0 \le \alpha < 1$, where N is an integer number, is always rounded to N.
For example, -1.3 will be rounded to -2 and 1.7 to 1. No special suffix is added to the names of functions that use this default mode.

Another rounding mode is "nearest right", in which the fixed point number $x = N + \alpha,\ 0 \le \alpha < 1$, where N is an integer number, is rounded as follows:

$$\lceil x \rceil = \begin{cases} N\ ,\ 0 \le \alpha < 0.5 \\ N+1\ ,\ 0.5 \le \alpha < 1 \end{cases}$$

For example, 1.5 will be rounded to 2 (same as for "nearest even" mode)
but –1.5 to –1 (different from "nearest even" mode where –1.5 is rounded to –2).

The suffix "NR" is added to names of functions that use the "nearest right" rounding mode.

Notational Conventions

This chapter, in addition to conventions used throughout the manual, uses the following notational conventions:

- In the description of function arguments, when an argument refers to a vector, the expression [n] in square brackets that may be given after the explanation of the argument specifies the number of elements (length) of that vector;

- Most of the speech coding functions for their proper execution interpret their integer and integer array arguments as fixed point numbers, which represent real numbers that vary in their specific ranges. Notation Q*n* used in the argument description means that this argument values are used in integer calculations inside the function as real numbers equal to the integer value multiplied by 2^{-n} (where "*n*" is called a scale factor).
  For example, if an argument value is described as "4096 in Q12", then it is interpreted as the real number 1.0; the value described as "15565 with scale factor 14" represents the real number 0.95; an argument described as "Q14 in [0, 1]" must be passed as an integer value in the range [0,16386].

Definitions

This section identifies the header files required for using the Intel IPP speech coding API described later in this chapter. For the definition of bit rate specifiers, refer to Data Structures section.

The header files `ippdefs.h` and `ippsc.h` must be included in order to link against any of the speech coding primitives, as shown in the following example code:

```
#include "ippdefs.h"
#include "ippsc.h"
int main()
{
    ...
    /* call GSM-AMR IPP functions */
        ippsLevinsonDurbin_GSMAMR(pSrcAutoCorr, pSrcDstLpc);
    ...
}
```

Data Structures

Some of the speech coding functions use a bit rate parameter of the enumerated type `IppSpchBitRate`. The set contains one specifier for each of the supported bit rates, as given in the structure below:

```
typedef enum {
  IPP_SPCHBR_4750
  IPP_SPCHBR_5150
  IPP_SPCHBR_5300
  IPP_SPCHBR_5900
  IPP_SPCHBR_6300
  IPP_SPCHBR_6700
  IPP_SPCHBR_7400
  IPP_SPCHBR_7950
  IPP_SPCHBR_9600
  IPP_SPCHBR_10200
  IPP_SPCHBR_12200
  IPP_SPCHBR_12800
  IPP_SPCHBR_16000
  IPP_SPCHBR_24000
  IPP_SPCHBR_32000
  IPP_SPCHBR_40000
  IPP_SPCHBR_DTX
} IppSpchBitRate;
```

The specifiers `IPP_SPCHBR_4750, IPP_SPCHBR_5150, IPP_SPCHBR_5900, IPP_SPCHBR_6700, IPP_SPCHBR_7400, IPP_SPCHBR_7950, IPP_SPCHBR_10200` and `IPP_SPCHBR_12200` are used in GSM-AMR functions and correspond to 4.75, 5.15, 5.9, 6.7, 7.4, 7.95, 10.2 and 12.2 Kbits/s transmitting rates, respectively.
The specifiers `IPP_SPCHBR_5300` and `IPP_SPCHBR_6300` are used in G.723.1 codec and correspond to 5.3 (low) and 6.3 (high) Kbits/s transmitting rates.
The specifiers `IPP_SPCHBR_9600, IPP_SPCHBR_12800` and `IPP_SPCHBR_16000` are used in G.728 codec and correspond to 9.6, 12.8 and 16 Kbits/s transmitting rates.

The specifiers `IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000` and `IPP_SPCHBR_40000` are used in G.726 codec and correspond to 16, 24, 32 and 40 Kbits/s transmitting rates, respectively.

Common Functions

This section describes common functions used in different speech codecs. The list of these functions is given in the following table:

Table 9-1 Intel IPP Common Speech Coding Functions

Function Base Name	Operation
ConvPartial	Performs linear convolution of 1D signals.
Mul_NR	Multiplies the elements of two vectors.
MulC_NR	Multiplies each element of a vector by a constant value.
MulPowerC_NR	Performs power weighting for each element of a vector.
AutoScale	Scales by the maximal elements.
DotProdAutoScale	Computes the dot product of two vectors using the automatic scaling.
InvSqrt	Computes inverse square root of vector elements.
AutoCorr	Calculates autocorrelation of a vector.
AutoCorrLagMax	Estimates the maximum auto-correlation of a vector.
AutoCorr_NormE	Estimates normal auto-correlation of a vector.
CrossCorr	Estimates the cross-correlation of two vectors.
CrossCorrLagMax	Estimates the maximum cross-correlation between two vectors.
SynthesisFilter	Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).

ConvPartial

Performs linear convolution of 1D signals.

```
IppStatus ippsConvPartial_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsConvPartial_16s32s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   ipp32s* pDst, int len);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the source and destination vectors.
scaleFactor	Scale factor used for output data scaling.

Discussion

The function ippsConvPartial_16s32s is declared in the ippsc.h file. This function computes the convolution of the vectors *pSrc1* and *pSrc2* as:

$$pDst[i] = \sum_{j=0}^{i} pSrc1[i] \cdot pSrc2[i-j], \quad i = 0, \ldots len-1.$$

The function ippsConvPartial_16s_Sfs computes the convolution of the vectors *pSrc1* and *pSrc2* and scales output data as given by:

$$pDst[i] = 2^{-scaleFactor} \cdot \sum_{j=0}^{i} pSrc1[i] \cdot pSrc2[i-j], \quad i = 0, \ldots len-1$$

Computed results are clipped.

Note that the output results produced by the partial convolution function are the same as the first *len* results of the general signal processing convolution function ippsConv_16s_Sfs , if the difference due to rounding is not taken into account.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.
ippStsScaleRangeErr	Indicates an error when *scaleFactor* is negative.

Mul_NR

Multiplies the elements of two vectors.

```
IppStatus ippsMul_NR_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsMul_NR_16s_ISfs (const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
   int scaleFactor);
```

Arguments

pSrc1, *pSrc2*	Pointers to the source vectors to be multiplied.
pDst	Pointer to the destination vector.
pSrc	Pointer to the vector to be multiplied by the elements of *pSrcDst* (for in-place operation).
pSrcDst	Pointer to the source and destination vector (for in-place operation).
len	Number of elements in each vector.
scaleFactor	Scale factor for output data scaling.

Discussion

The function ippsMul_NR is declared in the ippsc.h file. This function multiplies the first source vector *pSrc1* by the second source vector *pSrc2* element-wise, and stores results in *pDst*.

The in-place function flavor multiplies the vector *pSrc* by the vector *pSrcDst* element-wise, and stores results in *pSrcDst*.

Both function flavors scale the multiplication results in accordance with the *scaleFactor* value (see "Integer Scaling" in Chapter 2). If output values exceed the data range, the results are saturated.

Functions ippsMul_NR perform "nearest right" rounding needed to meet the bit-to-bit exactness requirement.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, *pSrc*, *pSrcDst*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsScaleRangeErr	Indicates an error when *scaleFactor* is less than 0.

MulC_NR

Multiplies each element of a vector by a constant value.

```
IppStatus ippsMulC_NR_16s_Sfs (Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int
   len, int scaleFactor);
IppStatus ippsMulC_NR_16s_ISfs (Ipp16s val, Ipp16s* pSrcDst, int len, int
   scaleFactor);
```

Arguments

val	The scalar value used to multiply to each element of the source vector.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Scale factor for output data scaling.

Discussion

The function ippsMulC_NR is declared in the ippsc.h file. This function multiplies the source vector *pSrc* by the scalar value *val*, and stores the result in *pDst*.

The in-place function flavor multiplies the vector *pSrcDst* by *val*, and stores the result in *pSrcDst*.

Both function flavors scale the multiplication results in accordance with the *scaleFactor* value (see "Integer Scaling" in Chapter 2). If output values exceed the data range, the results are saturated.

Functions ippsMulC_NR perform "nearest right" rounding needed to meet the bit-to-bit exactness requirement.

Return Value

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

MulPowerC_NR

Performs power weighting for each element of a vector.

```
IppStatus ippsMulPowerC_NR_16s_Sfs (const Ipp16s* pSrc, Ipp16s val, Ipp16s*
   pDst, int len, int scaleFactor);
```

Arguments

pSrc	Pointer to the source vector.
val	The scalar value to be multiplied to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in each vector.
scaleFactor	Scale factor for output data scaling.

Discussion

The function ippsMulPowerC_NR is declared in the ippsc.h file. This function multiplies each element of the source vector *pSrc* by a power of a scalar value, *val*, and stores the results in *pDst*. The calculation is performed as:

$pDst[i] = val^{i} \cdot pSrc[i], \quad 0 \le i < len$

The function scales the multiplication results in accordance with the *scaleFactor* value (see "Integer Scaling" in Chapter 2). If output values exceed the data range, the results are saturated.

The function ippsMulPowerC_NR performs "nearest right" rounding needed to meet the bit-to-bit exactness requirement.

Return Value

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

AutoScale

Scales by the maximal elements.

```
IppStatus ippsAutoScale_16s (const Ipp16s *pSrc, Ipp16s *pDst, int len,
   int *pScale);
IppStatus ippsAutoScale_16s_I(const Ipp16s *pSrcDst, int len, int *pScale);
```

Arguments

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

len	Number of elements in each vector.
pScale	Pointer to the input/output scaling factor.

Discussion

The function ippsAutoScale is declared in the ippsc.h file. This function scales the input vector as follows:

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

where *scaleFactor* = *normMax*-*pScale*[0],
and *normMax* is calculated so that the maximal absolute value of the vector can be normalized. The value *pScale*[0] is the input scaling factor. This scaling factor is replaced by *scaleFactor* after processing.

Return Value

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

DotProdAutoScale

Computes the dot product of two vectors using the automatic scaling.

```
IppStatus ippsDotProdAutoScale_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
   pSrc2, int len, Ipp32s* pDp, int* pSfs);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.

len	Number of elements in each vector.
pDp	Pointer to the output result.
pSfs	Pointer to return the scaling factor.

Discussion

The function `ippsDotProdAutoScale` is declared in the `ippsc.h` file. This function computes the dot product of two vectors and automatically scales it during calculation, adjusting the scale factor so as to eliminate possible overflow in the process:

$$pDp = 2^{scaleFactor} \cdot \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i]$$

The final scaling factor is returned via *pSfs*.
Vectors *pSrc1* and *pSrc2* must have the same length, *len*.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, *pDp*, or *pSfs* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsOverflow`	Indicates a warning that at least one result value was saturated.

InvSqrt

Computes inverse square root of vector elements.

```
IppStatus ippsInvSqrt_32s_I (Ipp32s *pSrcDst, int len );
```

Arguments

pSrcDst	Pointer to the source and destination vector.

len	Number of elements in the vector.

Discussion

The function ippsInvSqrt is declared in the ippsc.h file. This function computes the inverse square roots as:

$$pDst[i] = \frac{2^{31}}{\sqrt{pSrc[i]}}, \quad i = 0, \ldots len - 1,$$

using the approximation table. The input and output vectors are scaled by 30 bits.

Return Value

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

AutoCorr

Calculates autocorrelation of a vector.

```
IppStatus ippsAutoCorr_16s32s (const Ipp16s *pSrc, int srcLen, Ipp32s
*pDst, int dstLen );
```

Arguments

pSrc	Pointer to the first source vector [*srcLen*].
srcLen	Length of the source vector.
pDst	Pointer to the destination vector [*dstLen*].
dstLen	Length of the destination vector (the number of autocorrelation values to calculate).

Discussion

The function ippsAutoCorr is declared in the ippsc.h file. This function calculates autocorrelation of the input vector as follows:

$$pDst[n] = \sum_{i=n}^{srcLen-1} pSrc[i-n] \cdot pSrc[i], \quad n = 0,...,dstLen - 1.$$

and stores the result in *pDst*.

Return Value

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

AutoCorrLagMax

Estimates the maximum auto-correlation of a vector.

```
IppStatus ippsAutoCorrLagMax_Inv_16s (const Ipp16s* pSrc, int len, int
   lowerLag, int upperLag, Ipp32s* pMax, int* maxLag);
IppStatus ippsAutoCorrLagMax_Fwd_16s (const Ipp16s* pSrc, int len, int
   lowerLag, int upperLag, Ipp32s* pMax, int* maxLag);
```

Arguments

pSrc	Pointer to the source vector.
len	Length of autocorrelation.
lowerLag	Lower input lag value.
upperLag	Upper input lag value.
pMax	Pointer to the output maximum of the correlation.

maxLag	Pointer to the output lag which holds the maximum of the correlation.

Discussion

These functions are declared in the `ippsc.h` file. The function `ippsAutoCorrLagMax_Inv` finds the maximum autocorrelation within the given lag range as:

$$pMax = \max_{n} \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i-n] \ , \ lowerLag \le n \le upperLag \ .$$

The function `ippsAutoCorrLagMax_Fwd` finds the maximum autocorrelation within the given lag range as given by:

$$pMax = \max_{n} \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n] \ , \ lowerLag \le n \le upperLag \ .$$

If several maximums exist, functions return the first of them.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pMax*, or *maxLag* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

AutoCorr_NormE

Estimates normal auto-correlation of a vector.

```
IppStatus ippsAutoCorr_NormE_16s32s (const Ipp16s* pSrc, int len, Ipp32s*
   pDst, int lenDst, int *pNorm);
```

```
IppStatus ippsAutoCorr_NormE_NR_16s (const Ipp16s* pSrc, int len, Ipp16s*
   pDst, int lenDst, int* pNorm);
```

Arguments

pSrc	Pointer to the source vector in Q12.
len	Number of elements in the source vector.
pDst	Pointer to the destination vector that stores the estimated auto-correlation results of the source vector.
lenDst	The number of elements in the destination vector.
pNorm	Pointer to the output scale factor.

Discussion

The function ippsAutoCorr_NormE is declared in the ippsc.h file. This function calculates the autocorrelation of the source vector *pSrc*. The results are scaled according to the first autocorrelation coefficient (energy) value. Specifically, autocorrelation coefficients are multiplied by the factor 2^{norm}, where $norm \geq 0$ is calculated so as to make the first coefficient normalized:

$$pDst[n] = 2^{norm} \cdot \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n], \quad 0 \leq n < lenDst, \quad 0 \leq norm$$

where

$$pSrc[i] = \begin{cases} pSrc[i], & 0 \leq i < len \\ 0, & otherwise \end{cases}$$

The corresponding scaling factor, *norm*, is returned via *pNorm*.

The function that has the NR suffix performs the “nearest right” rounding (see Rounding mode).

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* or *pNorm* pointer is NULL.

ippStsSizeErr	Indicates an error when *len* or *lenDst* is less than or equal to zero.
ippStsOverflow	Indicates a warning that at least one result value was saturated.

CrossCorr

Estimates the cross-correlation of two vectors.

```
IppStatus ippsCrossCorr_16s32s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   int len, Ipp32s* pDst, int scaleFactor);
IppStatus ippsCrossCorr_NormM_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   int len, Ipp16s* pDst);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
len	Number of elements in the source and destination vectors.
pDst	Pointer to the destination vector.
scaleFactor	Scale factor of the destination vector.

Discussion

The function ippsCrossCorr is declared in the ippsc.h file. This function estimates the cross-correlation between the vector *pSrc1* and the vector *pSrc2* as given by:

$$corr[n] = 2^{scaleFactor} \cdot \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n] \quad , \; 0 \le n \le len \; ,$$

where

$$pSrc2[i] = \begin{cases} pSrc2[i] , & 0 \le i < len \\ 0 , & otherwise \end{cases}$$

Results are stored in the vector *pDst*. Correlation sums are saturated. Scaling is performed according to the *scaleFactor* value.

The function that has the NormM suffix uses the scale factor that normalizes the absolute maximum of the correlation sums.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is NULL.
ippStsScaleRangeErr	Indicates an error when *scaleFactor* is negative.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.

CrossCorrLagMax

Estimates the maximum cross-correlation between two vectors.

```
IppStatus ippsCrossCorrLagMax_16s (const Ipp16s *pSrc1, const Ipp16s
*pSrc2, int len, int lag, Ipp32s *pMax, int *maxLag);
```

Arguments

pSrc1	Pointer to the first source vector [*len*].
pSrc2	Pointer to the second source vector [*len*+*lag*+1].
len	Length of the cross-correlation.
lag	The maximal lag value.
pMax	Pointer to the output maximum cross-correlation.

maxLag	Pointer to the output lag which holds the cross-correlation maximum.

Discussion

The function ippsCrossCorrLagMax is declared in the ippsc.h file. This function finds the maximum of the cross-correlation between two input vectors by the formulae:

$$pMax = \max_{0 \le n \le lag} \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n]$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, *pMax*, or *maxLag* pointer is NULL.
ippStsRangeErr	Indicates an error when *lag* is less than 0.
ippStsSizeErr	Indicates an error when *len* is less or equal to 0.

SynthesisFilter

Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).

```
IppStatus ippsSynthesisFilter_NR_16s_Sfs(const Ipp16s * pLPC, const Ipp16s *
   pSrc, Ipp16s * pDst, int len, int scaleFactor, const Ipp16s *pMem);
IppStatus ippsSynthesisFilterLow_NR_16s_ISfs(const Ipp16s * pLPC, Ipp16s *
   pSrcDst, int len, int scaleFactor, const Ipp16s *pMem);
IppStatus ippsSynthesisFilter_NR_16s_ISfs(const Ipp16s * pLPC, Ipp16s *
   pSrcDst, int len, int scaleFactor, const Ipp16s *pMem);
```

Arguments

pLPC	Pointer to the input filter coefficients a_0, a_1,…,a_{10}.

pSrc	Pointer to the input vector.
pSrcDst	Pointer to the history input/filtered output vector.
pDst	Pointer to the filtered output.
len	Number of elements in the vector.
scaleFactor	Scale factor for the destination vector.
pMem	Pointer to the memory supplied for filtering. Should be updated outside the function with last 10 values of the destination vector.

Discussion

The function ippsSynthesisFilter is declared in the ippsc.h file. This function filters the input signal through the synthesis filter as follows:

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) \quad , n = 0,1,...,len-1 .$$

The function flavor ippsSynthesisFilterLow_NR_16s_ISfs performs no saturation and assumes that input data are small enough to ensure correct calculation without checking the overflow condition. This mode of operation is indicated by the suffix Low in the function name.

All flavors perform “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pLPC*, *pSrcDst*, *pDst*, or *pMem* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsScaleRangeErr	Indicates an error when *scaleFactor* is not equal to 12 or 13.
ippStsOverflow	Indicates a warning that at least one result value was saturated.

G.729 Related Functions

Intel IPP functions described in this section implement building blocks that can be used to create speech codecs compliant with the ITU-T Recommendation G.729 (see [ITU729], [ITU729A], [ITU729B]).
The list of these functions is given in the following table:

Table 9-2 Intel IPP G.729 Related Functions

Function Base Name	Operation
Basic Functions	
DotProd_G729	Computes the dot product of two vectors.
Interpolate_G729	Computes the weighted sum of two vectors.
Linear Prediction Analysis Functions	
AutoCorr_G729	Estimates the auto-correlation of a vector.
LevinsonDurbin_G729	Calculates LP coefficients from the autocorrelation coefficients.
LPCToLSP_G729	Converts LP coefficients to LSP coefficients.
LSFToLSP_G729	Converts Line Spectral Frequencies to LSP coefficients.
LSFQuant_G729	Performs quantization of LSF coefficients.
LSFDecode_G729	Decodes quantized LSFs.
LSFDecodeErased_G729	Reconstructs quantized LSFs in case when a frame is erased.
LSPToLPC_G729	Converts LSP coefficients to LP coefficients.
LSPQuant_G729	Quantizes the LSP coefficients.
LSPToLSF_G729	Converts LSP coefficients to LSF coefficients.
LagWindow_G729	Applies 60Hz bandwidth expansion.
Codebook Search Functions	
OpenLoopPitchSearch_G729	Searches for an optimal pitch value.
AdaptiveCodebookSearch_G729	Searches for the integer delay and the fraction delay, and computes the adaptive vector.
DecodeAdaptiveVector_G729	Restores the adaptive codebook vector by interpolating the past excitation.
FixedCodebookSearch_G729	Searches for the fixed codebook vector.
ToeplizMatrix_G729	Calculates 616 elements of the Toepliz matrix for the fixed codebook search.

Table 9-2 Intel IPP G.729 Related Functions (continued)

Function Base Name	Operation
Codebook Gain Functions	
DecodeGain_G729	Decodes the adaptive and fixed-codebook gains.
GainControl_G729	Calculates adaptive gain control.
GainQuant_G729	Quantizes the codebook gain using a two-stage conjugate-structured codebook.
AdaptiveCodebookContribution_G729	Updates target vector for codebook search by subtracting adaptive codebook contribution.
AdaptiveCodebookGain_G729	Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.
Filter Functions	
ResidualFilter_G729	Implements an inverse LP filter and obtains the residual signal.
SynthesisFilter_G729	Reconstructs the speech signal from LP coefficients and residuals.
LongTermPostFilter_G729	Restores the long-term information from the old speech signal.
ShortTermPostFilter_G729	Restores speech signal from the residuals.
TiltCompensation_G729	Compensates for the tilt in the short-term filter.
HarmonicFilter	Calculates the harmonic filter.
HighPassFilterSize_G729	Returns the G729 high-pass filter size.
HighPassFilterInit_G729	Initializes the high-pass filter.
HighPassFilter_G729	Performs G729 high-pass filtering.
IIR16s_G729	Performs IIR filtering.
PhaseDispersionGetStateSize_G729D	Queries the memory length of the phase dispersion filter.
PhaseDispersionInit_G729D	Initializes the phase dispersion filter memory.
PhaseDispersionUpdate_G729D	Updates the phase dispersion filter state.
PhaseDispersion_G729D	Performs the phase dispersion filtering.
Preemphasize_G729A	Computes pre-emphasis of a post filter.
WinHybridGetStateSize_G729E	Queries the length of the hybrid windowing module memory.
WinHybridInit_G729E	Initializes the hybrid windowing module memory.
WinHybrid_G729E	Applies the hybrid window and computes autocorrelation coefficients.
RandomNoiseExcitation_G729B	Initializes a random vector with a Gaussian distribution.

Basic Functions

These functions may be applied in both the encoding and decoding process.

DotProd_G729

Computes the dot product of two vectors.

```
IppStatus ippsDotProd_G729A_16s32s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   int len, Ipp32s* pDp);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
len	Number of elements in each vector.
pDp	Pointer to the output result.

Discussion

The function `ippsDotProd_G729A` is declared in the `ippsc.h` file. This function computes the dot product of two source vectors *pSrc1* and *pSrc2* as:

$$pDp = 2 \cdot \sum_{i=0}^{len/2} pSrc1[2 \cdot i] \cdot pSrc2[2 \cdot i]$$

Vectors *pSrc1* and *pSrc2* must have the same length, *len*.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, or *pDp* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

ippStsOverflow	Indicates a warning that at least one result value was saturated.

Interpolate_G729

Computes the weighted sum of two vectors.

```
IppStatus ippsInterpolate_G729_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   Ipp16s* pDst, int len);
IppStatus ippsInterpolateC_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s val1,
   const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsInterpolateC_NR_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s val1,
   const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int scaleFactor);
```

Arguments

pSrc1	Pointer to the first source vector.
val1	Factor to multiply to the first source vector.
pSrc2	Pointer to the second source vector.
val2	Factor to multiply to the second source vector.
pDst	Pointer to the destination (interpolated) vector.
len	Number of elements in the vectors.
scaleFactor	Scale factor for the destination vector.

Discussion

These functions are declared in the `ippsc.h` file. The function `ippsInterpolate_G729` computes the weighted sum as:

$$pDst[i] = (pSrc1[i] + \text{sign}(pSrc1[i])) >> 1 + (pSrc2[i] + \text{sign}(pSrc2[i])) >> 1 \ ,$$

for $i = 0, \ldots len - 1$.

Functions `ippsInterpolateC_G729` and `ippsInterpolateC_NR_G729` both use the same formula to compute the weighted sum as:

$pDst[i] = (val1 \cdot pSrc1[i] + val2 \cdot pSrc2[i]) >> scaleFactor, i = 0, ..len - 1.$

However, these functions differ in the rounding mode they use for output results (see Rounding mode).

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsScaleRangeErr`	Indicates an error when *scaleFactor* is less than 0.

Linear Prediction Analysis Functions

Functions described in this section implement LSP coding (quantization) and decoding, as well as transformation between LPC, LSP and LSF coefficients.

AutoCorr_G729

Estimates the auto-correlation of a vector.

```
IppStatus ippsAutoCorr_G729B(const Ipp16s * pSrcSpch,
   Ipp16s * pResultAutoCorrExp, Ipp32s * pDstAutoCorr);
```

Arguments

pSrcSpch	Pointer to the input speech signal vector [240].
pResultAutoCorrExp	Pointer to the exponent of autocorrelation coefficients.
pDstAutoCorr	Pointer to the autocorrelation coefficients vector [13].

Discussion

The function `ippsAutoCorr_G729` is declared in the `ippsc.h` file. This function calculates a set of 11 autocorrelation coefficients and their exponent for the input speech signal. The function is applied to the vector of 240 speech samples, which include 120 samples from past speech frames, 80 samples from the present speech frame, and 40 samples from the future frame. The functionality is as follows.

1. First, apply to speech samples the asymmetric windows given by:

$$w_{lp}(n) = \begin{cases} 0.54 - 0.46\cos\left(\frac{2n\pi}{399}\right), & n = 0, 1, \ldots, 199 \\ \cos\left(\frac{2(n-200)\pi}{159}\right), & n = 200, 201, \ldots, 239 \end{cases}$$

2. Next, calculate autocorrelations of the windowed speech samples $s(i)$, $i = 0, 1, \ldots, 239$, using the formula:

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k) \ , \ k = 0, 1, \ldots, 11$$

3. Finally, the autocorrelation coefficients are scaled according to the first autocorrelation coefficient (energy) value. Specifically, autocorrelation coefficients are multiplied by the factor 2^{norm}, where $norm \leq 0$ is calculated so as to make the first coefficient normalized. The corresponding scaling factor, *norm*, is returned via *pResultAutoCorrExp*.

NOTE. *The function* ippsAutoCorr_G729B *is actually a combination of* ippsMul_NR *and* ippsAutoCorr_NormE *functions. The following code details the correspondence.*

```
{
      short sig_win[240];
      ippsMul_NR_16s_Sfs(pSrcSpch,window,sig_win,240,15);
      ippsAutoCorr_NormE_16s32s(sig_win,240,pDstAutoCorr,11,
      &pResultAutoCorrExp);
}
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcSpch*, *pResultAutoCorrExp* or *pDstAutoCorr* pointer is NULL.

LevinsonDurbin_G729

Calculates LP coefficients from the autocorrelation coefficients.

```
IppStatus ippsLevinsonDurbin_G729_32s16s(const Ipp32s* pSrcAutoCorr, int
   order, Ipp16s* pDstLPC, Ipp16s* pDstRc, Ipp16s* pResultResidualEnergy);
```

```
IppStatus ippsLevinsonDurbin_G729B(const Ipp32s * pSrcAutoCorr, Ipp16s *
   pDstLPC, Ipp16s * pDstRC, Ipp16s * pResultResidualEnergy);
```

Arguments

order	The LP order.
pSrcAutoCorr	Pointer to the autocorrelation coefficients vector [*order*+1].
pDstLPC	Pointer to the output LP coefficients [*order*+1].
pDstRC	Pointer to the output reflection coefficients vector [*order*].
pResultResidualEnergy	Pointer to the residual energy.

Discussion

The functions ippsLevinsonDurbin_G729 and ippsLevinsonDurbin_G729B are declared in the ippsc.h file. These functions calculate Linear Prediction (LP) coefficients of the LP filter with the given order from the autocorrelation coefficients, using Levinson-Durbin algorithm. The function ippsLevinsonDurbin_G729 uses the parameter *order*, while the function ippsLevinsonDurbin_G729B operates for the default *order* = 10.

To obtain LP coefficients a_i, $i = 1, 2, \ldots,$ *order*, the following set of equations is to be solved:

$$\sum_{i=1}^{order} a_i \times r(|i-k|) = -r(k) \text{ , } k = 1,2,...,order \text{ .}$$

The functions perform the following steps:

1. Levinson-Durbin algorithm is applied to solve the above set of equations. This algorithm uses the following recursion.

$E^{[0]} = r(o)$

for $i = 1$ to *order*

$a_0^{[i-1]} = 1$

$$k_i = -\left[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j)\right] / E^{[i-1]}$$

$$a_i^{[i]} = k_i$$

for $j = 1$ to i-1

$$a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$$

end

$$E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$$

end

2. Set E as the output residual energy and k_i as reflection coefficients.

3. If the LPC filter used in this algorithm is unstable, that is some $|k_i|$ is very close to 1.0 during recursion, the elements of input vectors *pSrcDstLPC* and *pSrcDstRC* RC coefficients are not changed and the residual energy is set to 0.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcAutoCorr*, *pDstRC*, *pResultResidualEnergy*, or *pDstLPC* pointer is NULL.

LPCToLSP_G729

Converts LP coefficients to LSP coefficients.

```
IppStatus ippsLPCToLSP_G729_16s(const Ipp16s * pSrcLPC, const Ipp16s *
   pSrcPrevLSP, Ipp16s * pDstLSP);
IppStatus ippsLPCToLSP_G729A_16s(const Ipp16s * pSrcLPC, const Ipp16s *
   pSrcPrevLSP, Ipp16s * pDstLSP);
```

Arguments

`pSrcLPC`	Pointer to the LP coefficients vector [11], in Q12.
`pSrcPrevLSP`	Pointer to the previous LSP coefficients vector [10], in Q15.
`pDstLSP`	Pointer to the computed LSP coefficients vector [10].

Discussion

These functions are declared in the `ippsc.h` file. They convert 10th order LP coefficients to LSP coefficients.
The first function `ippsLPCToLSP_G729` is designed for G.729/B codec while the second function `ippsLPCToLSP_G729A` is designed for G.729A codec.
Functions perform the following steps:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the following recursive relations:

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i) \text{ , } i = 0,1,...,4 \text{ ,}$$

where $f_1(0) = f_2(0) = 1.0$.

2. Use Chebyshev polynomials to evaluate $F_1(z)$ and $F_2(z)$. The Chebyshev polynomials are given by.

$$C_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2$$

$$C_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2$$

For G.729/B, evaluate $F_1(z)$ and $F_2(z)$ at 60 points equally spaced between 0 and π and check for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided four times to track the root.
For G.729A, evaluate $F_1(z)$ and $F_2(z)$ at 50 points equally spaced between 0 and π and check for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided two times to track the root.

3. If all 10 roots needed to determine LSP coefficients are not found, just use the previous set of LSP coefficients.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcLPC*, *pSrcPrevLSP* or *pDstLSP* pointer is NULL.

LSFToLSP_G729

Converts Line Spectral Frequencies to LSP coefficients.

```
IppStatus ippsLSFToLSP_G729_16s (const Ipp16s *pLSF, Ipp16s *pLSP);
```

Arguments

pLSF	Pointer to the LSF vector [10], in Q13 in the range [0,π].
pLSP	Pointer to the LSP vector [10], in Q15 in the range[-1;1].

Discussion

The function ippsLSPToLSF_G729 is declared in the ippsc.h file. This function converts the Line Spectral Frequencies (LSF) to the LSP coefficients as given by:

$pLSP[i] = \cos(pLSF[i])$, $i = 1,..., 10$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pLSP* or *pLSF* pointer is NULL.
ippStsOverflow	Indicates a warning that at least one result value was saturated.

LSFQuant_G729

Performs quantization of LSF coefficients.

```
IppStatus ippsLSFQuant_G729_16s (const Ipp16s* pLSF, Ipp16s* pQuantLSFTable,
   Ipp16s* pQuantLSF, Ipp16s* quantIndex);
IppStatus ippsLSFQuant_G729B_16s (const Ipp16s* pLSF, Ipp16s* pQuantLSFTable,
   Ipp16s* pQuantLSF, Ipp16s* quantIndex);
```

Arguments

pLSF	Pointer to the LSF vector [10], in Q13 in the range [0,π].
pQuantLSFTable	Pointer to a matrix of 4 rows and 10 columns, containing previously quantized LSFs.
pQuantLSF	Pointer to the quantized LSF vector, in Q13 in the range [0,π].
quantIndex	Pointer to the output combined codebook indices *L*0, *L*1, *L*2, and *L*3.

Discussion

These functions are declared in the `ippsc.h` file.
The function `ippsLSFQuant_G729_16s` quantizes the difference between the input LSF coefficients and previously quantized LSF coefficients, using the switched Moving Average (MA) predictor.
Quantization is performed using a two-stage Vector Quantizer (VQ): 10-dimensional VQ using codebook **L**1 (128 entries), and 10-bit split VQ (5 dimensions each) on two codebooks, **L**2 and **L**3 (32 entries each).

The function `ippsLSFQuant_G729B_16s` quantizes the LSF coefficients using a two-stage split VQ, (in 5 and 4 bits). The second order MA predictor used in the SID-LPC quantization procedure is calculated as a linear interpolation of the first and second MA predictors, with weight values 0.6 and 0.4, respectively. The first stage of quantization is the same as that in G.729, but only part of the quantization table (32 entries) is used. The second stage is different. No splitting is done and also only portion of the second table (16 entries) is used.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pLSF*, *pQuantLSFTable*, *pQuantLSF*, or *quantIndex* pointer is NULL.
ippStsLSFLow	Indicates a warning that the corresponding filter has low stability.
ippStsLSFHigh	Indicates a warning that the corresponding filter has high stability.
ippStsLSFLowAndHigh	Indicates a warning that the corresponding filter has both low and high stability.

LSFDecode_G729

Decodes quantized LSFs.

```
IppStatus ippsLSFDecode_G729_16s (const Ipp16s *quantIndex, Ipp16s*
   pQuantLSFTable, Ipp16s* pQuantLSF);
IppStatus ippsLSFDecode_G729B_16s(const Ipp16s *quantIndex, Ipp16s
   *pQuantLSFTable, Ipp16s * pQuantLSF);
```

Arguments

quantIndex	Pointer to the vector of indices: $L0$, $L1$, $L2$, $L3$ (see Table 8/G.729 in [ITU729]) or $L0$, $L1$, $L2$ (see also [ITU729B] , section 4.3) .
pQuantLSFTable	Pointer to the input/output table [4][10] of 4 previously quantized LSFs vectors, in Q13.
pQuantLSF	Pointer to the quantized LSF output vector [10], in Q13.

Discussion

These functions are declared in the ippsc.h file.
The function ippsLSFDecode_G729 retrieves the quantized LSF coefficients from the quantization table and indices.

The function ippsLSFDecode_G729B retrieves the quantized LSF coefficients for the SID frame. Only *L*0, *L*1, *L*2 indices are used.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *quantIndex*, *pQuantLSFTable*, or *pQuantLSF* pointer is NULL.
ippStsLSFLow	Indicates a warning of only low stability of LSF.
ippStsLSFHigh	Indicates a warning of only high stability of LSF.
ippStsLSFLowAndHigh	Indicates a warning of both low and high stability of LSF.

LSFDecodeErased_G729

Reconstructs quantized LSFs in case when a frame is erased.

```
IppStatus ippsLSFDecodeErased_G729_16s (Ipp16s maIndex, Ipp16s
   *pQuantLSFTable, Ipp16s *pQuantLSF);
```

Arguments

maIndex	Switched MA predictor (*L*0) .
pQuantLSFTable	Pointer to the input/output table [4][10] of 4 previously quantized LSFs vectors, in Q12.
pQuantLSF	Pointer to the quantized LSF output vector [10], in Q12.

Discussion

The function ippsLSFDecodeErased_G729 is declared in the ippsc.h file. This function retrieves the quantized LSF coefficients through the previously decoded switched MA-predictor index (*L*0).

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pQuantLSFTable* or *pQuantLSF* pointer is `NULL`.

LSPToLPC_G729

Converts LSP coefficients to LP coefficients.

```
IppStatus ippsLSPToLPC_G729_16s(const Ipp16s * pSrcLSP, Ipp16s * pDstLPC);
```

Arguments

pSrcLSP	Pointer to the LSP coefficients vector [10], in Q15.
pDstLPC	Pointer to the LP coefficients vector [11], in Q12.

Discussion

The function `ippsLSPToLPC_G729` is declared in the `ippsc.h` file. This function converts a set of 10th order LSP coefficients to LP coefficients.
It performs the following steps:

1. Calculates the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursive relations:

```
for i = 1 to 5
```

$$f_1(i) = -2q_{2i-1} \times f_1(i-1) + 2f_1(i-2)$$

```
    for j = i-1 down to 1
```

$$f_1^{[i]}(j) = f_1^{[i-1]}(j) - 2q_{2i-1} \times f_1^{[i-1]}(j-1) + f_1^{[i-1]}(j-2)$$

```
    end
end
```

Here the initial values are set to $f_1(0) = 1$ and $f_1(-1) = 0$.
The coefficients $f_2(i)$ are computed similarly by replacing q_{2i-1} by q_{2i} .

2. $F_1(z)$ and $F_2(z)$ are then multiplied by $1+z^{-1}$ and $1-z^{-1}$ respectively to obtain $F_1'(z)$ and $F_2'(z)$ as given by:

$$f_1'(i) = f_1(i) + f_1(i-1) \quad , i = 1,2,...5$$

$$f_2'(i) = f_2(i) - f_2(i-1) \quad , i = 1,2,...5$$

3. Finally, the function computes the LP coefficients from $f_1'(i)$ and $f_2'(i)$ as:

$$a_i = \begin{cases} 0.5 \times f_1'(i) + 0.5 \times f_2'(i) \quad , i = 1, 2, \ldots, 5 \\ 0.5 \times f_1'(11-i) - 0.5 \times f_2'(11-i) \quad , i = 6, 7, \ldots, 10 \end{cases}$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcLSP`* or *`pDstLPC`* pointer is `NULL`.

LSPQuant_G729

Quantizes the LSP coefficients.

```
IppStatus ippsLSPQuant_G729_16s(const Ipp16s * pSrcLSP, Ipp16s
   *pSrcDstPrevFreq, Ipp16s * pDstQLSP, Ipp16s * pDstQLSPIndex);
```

Arguments

`pSrcLSP`	Pointer to the LSP coefficients vector [10], in Q15.
`pSrcDstPrevFreq`	Pointer to the four previous and updated quantized frequencies vector [40], in Q13. Elements 0 to 9 are for the newest frame and elements 30 to 39 are for the oldest frame.
`pDstQLSP`	Pointer to the LSP coefficients vector [10], in Q15.
`pDstQLSPIndex`	Pointer to the combined codebook indices `L0`, `L1`, `L2` and `L3`, of length 2. Its first element represents indices `L0` and `L1` in the following combination: `L1` occupies the first 7 bits starting from LSB, whereas `L0` resides in the next 1 bit adjacent to `L1`. The second element represents indices `L2`

and `L3` in the following combination: `L3` occupies the first 5 bits starting from LSB, and `L2` resides in the next 5 bits adjacent to `L3`.

Discussion

The function `ippsLSPQuant_G729` is declared in the `ippsc.h` file. This function obtains the quantized LSP coefficients and the codebook indices. It performs the following operations:

1. Converts the LSP coefficients q_i to LSF coefficients ω_i in the normalized frequency domain [0, π] as:

$$\omega_i = \arccos(q_i) \ , \ i = 1,...,10$$

2. Uses a switched 4th order MA prediction to predict the LSF coefficients of the current frame. The difference between the computed and predicted coefficients is quantized using a two-stage vector quantizer. The first stage is a 10-dimensional VQ using codebook **L**1 with 128 entries (7 bits). The second stage is a 10 bit VQ which was implemented as a split VQ using two 5-dimensional codebooks, **L**2 and **L**3, containing 32 entries (5 bits) each. To explain the quantization process, it is convenient to first describe the decoding process. Each coefficient is obtained from the sum of two codebooks:

$$\hat{l} = \begin{cases} \mathbf{L}1_i(L1) + \mathbf{L}2_i(L2) \ , \ i = 1, \ ..., \ 5 \\ \mathbf{L}1_i(L1) + \mathbf{L}3_{i-5}(L3) \ , \ i = 6, \ ..., \ 10 \end{cases}$$

where `L1, L2` and `L3` are the codebook indices.

3. The vector to be quantized for the current frame *m* is obtained from:

$$l_i = \left[\omega_i^{(m)} - \sum_{k=1}^{4} \hat{p}_{i,k} \hat{l}_i^{(m-k)} \right] \Big/ \left(1 - \sum_{k=1}^{4} \hat{p}_{i,k} \right) \ , \ i = 1,...,10 \ ,$$

where $\hat{p}_{i,k}$ are the coefficients of the switched MA predictor. The first codebook **L**1 is searched and the entry `L1` that minimizes the (unweighted) mean-squared error is selected. This is followed by a search of the second codebook **L**2, which defines the lower part of the second stage. To avoid sharp resonances in the quantized LP

synthesis filter, the partial vector $\hat{l}_i$, i = 1,…,5, is rearranged such that adjacent coefficients have a minimum distance of *J*. The rearrangement procedure is shown below:

```
for i = 2,...,10
   if l̂[i-1] > l̂[i] - j
        l̂[i-1] = (l̂[i] + l̂[i-1] - j)/ 2
        l̂[i] = (l̂[i] + l̂[i-1] + j)/ 2
    end
end
```

where *J* is 0.0012 for the first pass. Using the selected first stage vector `L1` and the lower part of the second stage `L2`, the higher part of the second stage is searched from the codebook **L**3. Again, the rearrangement procedure is used to guarantee a minimum distance of 0.0012. The resulting vector $\hat{l}_i$, i = 1, …,10 , is rearranged to guarantee a minimum distance of 0.0006. The quantized LSF coefficients $\hat{\omega}_i^{(m)}$ for the current frame *m* are obtained from the weighted sum of previous quantizer outputs $\hat{l}_i^{(m-k)}$, and the current quantizer output $\hat{l}_i^{(m)}$:

$$\hat{\omega}_i^{(m)} = \left(1 - \sum_{k=1}^{4} \hat{p}_{i,k}\right) \hat{l}_i^{(m)} + \sum_{k=1}^{4} \hat{p}_{i,k} \hat{l}_i^{(m-k)} \quad , \; i = 1,...,10$$

There are two MA predictors. Which MA predictor to use is defined by a single bit `L0`. It is determined by the one that minimizes the weighted mean-squared error:

$$E_{lsf} = \sum_{i=1}^{10} w_i (\omega_i - \hat{\omega}_i)^2$$

The weights w_i are made adaptive as a function of the unquantized LSF coefficients,

$$w_1 = \begin{cases} 1.0 & , \; if \;\; \omega_2 - 0.04\pi - 1 > 0 \\ 10(\omega_2 - 0.04\pi - 1)^2 + 1 , & otherwise \end{cases}$$

$$w_i = \begin{cases} 1.0 & , \ if \ \omega_{i+1} - \omega_{i-1} - 1 > 0 \\ 10(\omega_{i+1} - \omega_{i-1} - 1)^2 + 1, & otherwise \end{cases}, 2 \le i \le 9$$

$$w_{10} = \begin{cases} 1.0 & , \ if \ -\omega_9 + 0.92\pi - 1 > 0 \\ 10(-\omega_9 + 0.92\pi - 1)^2 + 1, & otherwise \end{cases}$$

In addition, the weights w_5 and w_6 are multiplied by 1.2 each.

4. To avoid sharp resonance in the LP filter, the rearrangement procedure is applied twice to the resulting vector $\hat{l}_i$ with *J* = 0.0012 and 0.0006 sequentially.

5. After computing $\hat{\omega}_i$, the corresponding filter is checked for stability. This is done as follows:

1) Order the coefficients $\hat{\omega}_i$ in increasing value;

2) If $\hat{\omega}_i$ < 0.005 then $\hat{\omega}_i$ = 0.005;

3) If $\hat{\omega}_{i+1} - \hat{\omega}_i < 0.0391$ then $\hat{\omega}_{i+1} = \hat{\omega}_i + 0.0391$, i = 1, …,9;

4) If $\hat{\omega}_{10} > 3.315$ then $\hat{\omega}_{10}$ =3.315.

6. Convert the quantized LSF vectors to LSP vectors.

NOTE. *The function* `ippsLSPQuant` *is actually a combination of* `ippsLSPToLSF`, `ippsLSFQuant`, *and* `ippsLSFToLSP` *functions. The following code details the correspondence.*

```
IppStatus ippsLSPQuant_G729_16s(const Ipp16s * pSrcLSP, Ipp16s *
pSrcDstPrevFreq, Ipp16s * pDstQLSP, Ipp16s * pDstQLSPIndex) {
   __ALIGN(8) short lsf[LP_ORDER];
   __ALIGN(8) short lsp_q[LP_ORDER];
   short q_index[4];
   IppStatus sts=ippsLSPToLSF_G729_16s(pSrcLSP, lsf);
   if (sts != ippStsNoErr) return sts;
   sts = ippsLSFQuant_G729_16s(lsf, pSrcDstPrevFreq, lsf_q, q_index );
   if(sts != ippStsNoErr) return sts;
   pDstQLSPIndex[0] = (q_index[0]<<G729_NC0_B) | q_index[1];
```

```
    pDstQLSPIndex[1] = (q_index[2]<<G729_NC1_B) | q_index[3];
    ippsLSFToLSP_G729_16s(lsp_q, pDstQLSP)
    return ippStsNoErr;
}
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcLSP`*, *`pSrcDstPrevFreq`*, *`pDstQLSP`* or *`pDstQLSPIndex`* pointer is `NULL`.

LSPToLSF_G729

Converts LSP coefficients to LSF coefficients.

```
IppStatus ippsLSPToLSF_Norm_G729_16s (const Ipp16s *pLSP, Ipp16s *pLSF);
IppStatus ippsLSPToLSF_G729_16s (const Ipp16s *pLSP, Ipp16s *pLSF);
```

Arguments

`pLSP`	Pointer to the LSP vector, in Q15 in the range [-1:1].
`pLSF`	Pointer to the LSF vector. Values must be scaled by 13 bits in the range [0:π] (for `ippsLSPToLSF_G729_16s` function) or by 15 bits in the range [0:0.5] (for `ippsLSPToLSF_Norm_G729_16s` function).

Discussion

These functions are declared in the `ippsc.h` file.
The function `ippsLSPToLSF_Norm_G729` converts the LSP coefficients to LSF coefficients as follows:

$$pLSF[i] = 2^{scaleFactor} \cdot \arccos(pLSP[i]) \ , 0 \le i < 10 .$$

The scale factor is chosen such that the first LSF coefficient is normalized to the interval [0:0.5] by multiplying to 1/π.
The function `ippsLSPToLSF_G729` retains the result within the interval [0:π]. Both functions use the same approximation table to calculate the inverse cosine.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pLSP* or *pLSF* pointer is NULL.

LagWindow_G729

Applies 60Hz bandwidth expansion.

```
IppStatus ippsLagWindow_G729_32s_I (Ipp32s* pSrcDst, int len);
```

Arguments

pSrcDst	Pointer to the autocorrelation vector.
len	Number of elements in the vector.

Discussion

The function ippsLagWindow_G729 is declared in the ippsc.h file. This function applies a 60Hz bandwidth expansion to the autocorrelation vector as follows:

$$r[0] = 1.0001 \cdot r[0]$$

$$r[i] = w_{lag}[i] \cdot r[i], \; i = 0, \ldots len - 1,$$

where

$$w_{lag}[i] = \exp\left(-\frac{1}{2} \cdot \left(2\pi f_0 \cdot \frac{i}{f_s}\right)^2\right)$$

and

f_0 = 60Hz, f_s = 8000Hz.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsRangeErr	Indicates an error when *len* > 12.

Codebook Search Functions

These functions perform open loop pitch estimation using the adaptive codebook, and search in ACELP excitation (fixed) codebook for optimal signs and positions of the pulses.

OpenLoopPitchSearch_G729

Searches for an optimal pitch value.

```
IppStatus ippsOpenLoopPitchSearch_G729_16s(const Ipp16s * pSrc, Ipp16s *
   bestLag);
IppStatus ippsOpenLoopPitchSearch_G729A_16s(const Ipp16s * pSrc, Ipp16s *
   bestLag);
```

Arguments

pSrc	Pointer to the perceptually weighted speech signal vector [170].
bestLag	Pointer to the open-loop pitch search result.

Discussion

These functions are declared in the `ippsc.h` file. They use an open-loop method to get the pitch value. To reduce the complexity of the search for the best adaptive-codebook delay, the search range is limited around a candidate delay, which is obtained from maximizing the correlation coefficients of the perceptually weighted speech.
The function `ippsOpenLoopPitchSearch_G729A` is designed for G.729A codec and implements the following algorithm:
1. Overflow and underflow check:

$$nSumTemp = \sum_{j=-71}^{40} sw(2j-1) \times sw(2j-1);$$

if $nSumTemp \geq 2^{30}$, overflow happens, $sw'(i) = sw(i) >> 3$, i = -143,...,79;

else if $nSumTemp < 2^{19}$, underflow happens, $sw'(i) = sw(i) << 3$, i= -143,...,79;

else $sw'(i) = sw(i)$, i = -143,...,79 .

2. Compute the correlation:

$$R(k) = \sum_{n=0}^{39} sw'(2n) \times sw'(2n-k), \quad k = 20,...,143$$

3. In the range [20, 39], find the maximum of $R(k)$, and optimal k_1. In the range [40, 79], find the maximum of $R(k)$, and optimal k_2. In the range [80, 143], among the even k, find the maximum of $R(k)$, and suboptimal k_3. Let $k_3' = k_3$.

4. if $R(k_3'+1) > R(k_3)$, then $k_3 = k_3'+1$.

5. if $R(k_3'-1) > R(k_3)$, then $k_3 = k_3'-1$.

6. For the three optimal k_i, i = 1,2,3, normalize the three peak values of correlation:

$$R'(k_i) = \frac{R(k_i)}{\sqrt{\sum_{n=0}^{39} sw'(2n-k_i) \times sw'(2n-k_i)}}$$

7. Post process $R'(k_1)$ and $R'(k_2)$:

```
if (|2 × k2 − k3| < 5) {
    R'(k2) = R'(k2) + 0.25 × R'(k3);
}
if (|3 × k2 − k3| < 7) {
    R'(k2) = R'(k2) + 0.25 × R'(k3);
    }
if (|2 × k1 − k2| < 5) {
    R'(k1) = R'(k1) + 0.2 × R'(k2);
}
```

```
if (|3× k1 − k2| < 7) {
    R′(k1) = R′(k1) + 0.2× R′(k2);
}
```

8. Find the optimal delay k_{opt} from k_1 to k_3, which satisfies

$R'(k_{opt}) = \max\{R'(k_i),\ i = 1,...,3\}$.

The function `ippsOpenLoopPitchSearch_G729` is designed for G.729/B codec and implements the following algorithm:

1. Overflow and underflow check:

$$nSumTemp = \sum_{i=-143}^{79} sw(i) \times sw(i)$$

if $nSumTemp \geq 2^{30}$, overflow happens, $sw'(i) = sw(i) >> 3$, i = -143,...,79;

else if $nSumTemp < 2^{19}$, underflow happens, $sw'(i) = sw(i) << 3$, i= -143,...,79;

else $sw'(i) = sw(i)$, i = -143,...,79 .

2. Compute the correlation:

$$R(k) = \sum_{i=0}^{79} sw'(i) \times sw'(i-k),\ k = 20,...,143$$

3. In the range [20, 39], find the maximum of $R(k)$, and optimal $k = k_1$. In the range [40,79], find the maximum of $R(k)$, and optimal $k = k_2$. In the range [80,143], find the maximum of $R(k)$, and optimal $k = k_3$.

4. For the three optimal k_i, i = 1,2,3, normalize the three peak values of correlation:

$$R'(k_i) = \frac{R(k_i)}{\sqrt{\sum_{n=0}^{79} sw'(n-k_i) \times sw'(n-k_i)}}$$

2. Find the optimal delay k_{opt} from k_1 to k_3, according to the following relations:

$k_{opt} = k_1, \quad R'(k_{opt}) = R'(k_1)$

$if\ (R'(k_2) \geq 0.85R'(k_{opt}))\ ,\ k_{opt} = k_2, \quad R'(k_{opt}) = R'(k_2)$

$if\ (R'(k_3) \geq 0.85R'(k_{opt}))\ ,\ k_{opt} = k_3, \quad R'(k_{opt}) = R'(k_3)$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *bestLag* pointer is `NULL`.

AdaptiveCodebookSearch_G729

Searches for the integer delay and the fraction delay, and computes the adaptive vector.

```
IppStatus ippsAdaptiveCodebookSearch_G729_16s(Ipp16s valOpenDelay, const Ipp16s *
   pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse, Ipp16s *
   pSrcDstPrevExcitation, Ipp16s * pDstDelay, Ipp16s * pDstAdptVector, Ipp16s
   subFrame);
IppStatus ippsAdaptiveCodebookSearch_G729A_16s(Ipp16s valOpenDelay, const Ipp16s *
   pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse, Ipp16s *
   pSrcDstPrevExcitation, Ipp16s * pDstDelay, Ipp16s * pDstAdptVector, Ipp16s
   subFrame);
IppStatus ippsAdaptiveCodebookSearch_G729D_16s  (Ipp16s valOpenDelay, const
   Ipp16s* pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse, Ipp16s*
   pSrcDstPrevExcitation, Ipp16s * pDstDelay, Ipp16s subFrame);
```

Arguments

valOpenDelay	Open-loop delay, in the range [18,145].
pSrcAdptTarget	Pointer to the target signal for adaptive-codebook search vector [40].

pSrcImpulseResponse	Pointer to the impulse response of weighted synthesis filter vector [40], in Q12.
pSrcDstPrevExcitation	Pointer to the previous and updated excitation vector [194].
pDstDelay	Pointer to the integer delay and fraction delay vector [2].
pDstAdptVector	Pointer to the adaptive vector [40].
subFrame	Subframe number, either 0 or 1.

Discussion

These functions are declared in the `ippsc.h` file. They search for the integer delay *T* and the fraction delay *t*, using the target signal and the past filtered excitation, and compute the adaptive vector. They are applied in subframes.

The function **`ippsAdaptiveCodebookSearch_G729A`** is designed for G.729A codec and implements the following algorithm:

1. Search for the optimal delay *k*, which maximizes the following expression:

$$R_N(k) = \sum_{n=0}^{39} x(n)y_k(n) = \sum_{n=0}^{39} x(n) \sum_{i=0}^{n} u_k(i)h(n-i) = \sum_{i=0}^{39} \sum_{n=i}^{39} u_k(i)x(n)h(n-i)$$

$$= \sum_{n=0}^{39} u_k(n)x_d(n), \quad k = t_{min}, \ldots, t_{max}$$

$$x_d(n) = \sum_{i=n}^{39} x(i)h(i-n) \ , n = 0,1,...,39 .$$

Denote the optimal delay *k* as *T*.

2. If the search result *T* is less than 85 in the first subframe or it is in the second subframe, search for the fraction delay by maximizing the expression below:

$$R_{Nt}(k) = \sum_{n=0}^{39} x_d(n)u_{kt}(n) \ , t = -1,0,1 \ , k = T$$

$$u_{kt}(n) = \sum_{i=0}^{9} u_k(n - k_p - i) b_{30}(t_p + 3i) + \sum_{i=0}^{9} u_k(n - k_p + 1 + i) b_{30}(3 - t_p + 3i) ,$$

$n = 0,1,...,39$

$$k_p = \begin{cases} T , & t = 0, -1 \\ T + 1 , & t = 1 \end{cases} , \qquad t_p = \begin{cases} -t , & t = 0, -1 \\ 2 , & t = 1 \end{cases}$$

3. If the fraction result is *t*, then the new adaptive vector is computed as:

$v(n) = u_{kt}(n)$, $n = 0,1,...,39$.

The function **ippsAdaptiveCodebookSearch_G729** is designed for G.729/B codec and implements the following algorithm:

1. Search for the integer delay *k* which maximizes the following expression:

$$R(k) = \frac{\sum_{n=0}^{39} x(n) y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n) y_k(n)}}$$

where y_k is the past filter excitation at delay *k*. Denote the optimal *k* as *T*.

2. If the search result *T* is less than 85 in the first subframe or it is in the second subframe, search for the fraction delay *t* by maximizing the expression below:

$$R_t(T) = \sum_{i=0}^{3} R(T - i) b_{12}(t + 3i) + \sum_{i=0}^{3} R(T + 1 + i) b_{12}(3 - t + 3i) , \quad t = 0,1,2 .$$

3. Finally, obtain the adaptive vector by interpolating the past excitation.
The interpolation method is the same as the method used to get the new adaptive vector in the `ippsAdaptiveCodebookSearch_G729A` function.

The function **ippsAdaptiveCodebookSearch_G729D** differs from ippsAdaptiveCodebookSearch_G729 in that the number of lags in the second subframe has been reduced from 32 to 16.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcAdptTarget*, *pSrcImpulseResponse*, *pSrcDstPrevExcitation*, *pDstDelay*, or *pDstAdptVector* pointer is NULL.
ippStsRangeErr	Indicates an error when *valOpenDelay* is not in the range [18, 145], or *subFrame* is not 0 or 1.

DecodeAdaptiveVector_G729

Restores the adaptive codebook vector by interpolating the past excitation.

```
IppStatus ippsDecodeAdaptiveVector_G729_16s(const Ipp16s * pSrcDelay, Ipp16s *
   pSrcDstPrevExcitation, Ipp16s * pDstAdptVector);
IppStatus ippsDecodeAdaptiveVector_G729_16s_I(const Ipp16s * pSrcDelay, Ipp16s
   * pSrcDstPrevExcitation);
```

Arguments

pSrcDelay	Pointer to the integer and fraction delay for each subframe.
pSrcDstPrevExcitation	Pointer to the past and updated excitations vector [194].
pDstAdptVector	Pointer to the adaptive codebook vector [40].

Discussion

The function ippsDecodeAdaptiveVector_G729 is declared in the ippsc.h file. This function restores the adaptive codebook vector by interpolating the past excitation. It performs the following steps:

1. Decodes the integer part and fractional part of the pitch delay T and t.

2. Interpolates the past excitation to get the current adaptive excitation as:

$$v(n) = \sum_{i=0}^{9} u(n-T-i) \times b_{30}(t+3i) + \sum_{i=0}^{9} u(n-T+1+i) \times b_{30}(3-t+3i), n = 0,1,...,39$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDelay*, *pSrcDstPrevExcitation*, or *pDstAdptVector* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *pSrcDelay*[0] is not in the range [18, 145], or *pSrcDelay*[1] is not 0 or 1.

FixedCodebookSearch_G729

Searches for the fixed codebook vector.

```
IppStatus ippsFixedCodebookSearch_G729_16s(const Ipp16s * pSrcFixedCorr,
   Ipp16s * pSrcDstMatrix, Ipp16s * pDstFixedVector, Ipp16s * pDstFixedIndex,
   Ipp16s * pSearchTimes, Ipp16s subFrame);
IppStatus ippsFixedCodebookSearch_G729_32s16s(const Ipp16s * pSrcFixedCorr,
   Ipp32s * pSrcDstMatrix, Ipp16s * pDstFixedVector, Ipp16s * pDstFixedIndex,
   Ipp16s * pSearchTimes, Ipp16s subFrame);
IppStatus ippsFixedCodebookSearch_G729A_16s(const Ipp16s * pSrcFixedCorr,
   const Ipp16s * pSrcDstMatrix, Ipp16s * pDstFixedVector, Ipp16s *
   pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729A_32s16s(const Ipp16s * pSrcFixedCorr,
   const Ipp32s * pSrcDstMatrix, Ipp16s * pDstFixedVector, Ipp16s *
   pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729E_16s(int mode, const Ipp16s*
   pSrcFixedTarget, const Ipp16s* pSrcLtpResidual, const Ipp16s*
   pSrcImpulseResponse, Ipp16s* pDstFixedVector, Ipp16s* pDstFltFixedVector,
   Ipp16s* pDstFixedIndex);
```

```
IppStatus ippsFixedCodebookSearch_G729D_16s(Ipp16s *pSrcFixedCorr, Ipp16s
   *pSrcDstMatrix, Ipp16s *pSrcImpulseResponse, Ipp16s *pDstFixedVector,
   Ipp16s *pDstFltFixedVector, Ipp16s *pDstFixedIndex);
```

Arguments

mode	Indicates forward (*mode* = 0) of backward (*mode* = 1) LP analysis mode.
pSrcFixedCorr	Pointer to the correlation signal of fixed-codebook search vector [40], in Q13.
pSrcDstMatrix	Pointer to the Toepliz matrix of length 616 (elements of `Ipp16s` type in Q9 or `Ipp32s` type in Q25).
pSrcFixedTarget	Pointer to the input updated target speech vector.
pSrcLtpResidual	Pointer to the input residual after long term prediction vector.
pSrcImpulseResponse	Pointer to the LP synthesis filter [40].
pDstFixedVector	Pointer to the fixed-codebook vector [40], in Q13.
pDstFltFixedVector	Pointer to the output filtered fixed-codebook vector.
pDstFixedIndex	Pointer to the fixed-codebook index [2]. The first element is the sign codeword *S* and the second element is the position codeword *C*.
pSearchTimes	Pointer to the maximum search time for the remaining subframes.
subFrame	Subframe number, 0 or 1.

Discussion

These functions are declared in the `ippsc.h` file. They search for the fixed-codebook vector and corresponding vector index, respectively. They are applied in subframes.

The first function `ippsFixedCodebookSearch_G729` is designed for G.729/B codec and uses the nested-loop search approach. The Toepliz matrix is updated in this function. This function performs the following steps:

1. First, the 616 elements in the Toepliz matrix are updated as:

$$\Phi'(i, j) = \text{sign}[d(i)]\text{sign}[d(j)]\Phi(i, j)$$

where $\Phi(i, j)$ is the element of the Toepliz matrix, and $d(n)$ is the correlation signal.

2. Then, a threshold thr_3 is calculated using:

$$thr_3 = av_3 + 0.4(max_3 - av_3)$$

where max_3 and av_3 are the absolute maximum and absolute average of the first three pulses in $d(n)$.

3. Next, search for the fixed-codebook vector to minimize the term C^2 / E , and obtain the vector index, where *C* is the correlation given by:

$$C = |d(m_0)| + |d(m_1)| + |d(m_2)| + |d(m_3)| \quad (9\text{-}1)$$

and *E* is the energy obtained by:

$$\begin{aligned} E = {} & 2(\Phi'(m_0, m_0) + \Phi'(m_1, m_1) + \Phi'(m_0, m_1) \\ & + \Phi'(m_2, m_2) + \Phi'(m_0, m_2) + \Phi'(m_1, m_2) \\ & + \Phi'(m_3, m_3) + \Phi'(m_0, m_3) + \Phi'(m_1, m_3) + \Phi'(m_2, m_3)) \end{aligned} \quad (9\text{-}2)$$

The search procedure is a kind of 4-loop search, and the last loop is entered only if the pre-computed correlation has exceeded the threshold thr_3. Also, the maximum number of times the loop can be entered is fixed to 180 so that a low percentage of the codebook is searched.

The sign codeword of the index, *S*, is obtained as follows.

$$S = s_0 + 2s_1 + 4s_2 + 8s_3 \quad (9\text{-}3)$$

The position codeword of the index, *C*, is computed as:

$$C = (m_0 / 5) + 8(m_1 / 5) + 64(m_2 / 5) + 512(2(m_3 / 5) + jx) \quad , \quad (9\text{-}4)$$

where $jx = 0$ if $m_3 = 3, 8, \ldots, 38$, and $jx = 1$ if $m_3 = 4, 9, \ldots, 39$.

4. Finally, the fixed-codebook vector is calculated.

The function `ippsFixedCodebookSearch_G729A` is designed for G.729A codec and uses the iterative depth-first, tree search approach.
The functionality is as follows:

1. First, calculate 616 elements of the new Toepliz matrix, $\Phi'(i, j)$, from the old Toepliz matrix Φ, using the formula:

$$\Phi'(i, j) = \text{sign}[d(i)]\text{sign}[d(j)]\Phi(i, j) ,$$

where $\Phi(i, j)$ is the element of the Toepliz matrix, and $d(n)$ is the correlation signal.

2. Next, search for the fixed-codebook vector to minimize the term C^2/E , and obtain the vector index, where C is the correlation given by (9-1), and E is the energy obtained as (9-2).

The search is an iterative depth-first, tree search approach.

The sign codeword of the index, *S*, is computed according to (9-3) and the position codeword of the index, *C*, is obtained as given by (9-4).

3. Finally, the fixed-codebook vector is calculated.

Both `ippsFixedCodebookSearch_G729` and `ippsFixedCodebookSearch_G729A` functions search four signed pulses in positions given by following table:

Pulses	Positions
i_0	0,5,10,15,20,25,30,35
i_1	1,6,11,16,21,26,31,36
i_2	2,7,12,17,22,27,32,37
i_3	3,8,13,18,23,28,33,38 4,9,14,19,24,29,34,39

The function `ippsFixedCodebookSearch_G729E` is designed for G.729E codec and searches ten signed pulses in five tracks in positions shown in the following table:

Pulses	Positions
i_0, i_1	0,5,10,15,20,25,30,35
i_3, i_4	1,6,11,16,21,26,31,36
i_5, i_6	2,7,12,17,22,27,32,37
i_7, i_8	3,8,13,18,23,28,33,38
i_9, i_{10}	4,9,14,19,24,29,34,39

The two pulses for each track may overlap resulting in a single pulse with duplicated amplitude.

The function ippsFixedCodebookSearch_G729D is designed for G.729D codec and searches for two signed pulses in two overlapping tracks given in following table:

Pulses	Positions
i_0	1, 3, 6, 8, 11, 13, 16, 18, 21, 23, 26, 28, 31, 33, 36, 38
i_1	0, 1, 2, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 16, 17, 19, 20, 21, 22, 24, 25, 26, 27, 29, 30, 31, 32, 34, 35, 36, 37, 39

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the pointers is NULL.
ippStsRangeErr	Indicates an error when *subFrame* is not 0 or 1.

ToeplizMatrix_G729

Calculates 616 elements of the Toepliz matrix for the fixed codebook search.

```
IppStatus ippsToeplizMatrix_G729_16s(const Ipp16s * pSrcImpulseResponse,
   Ipp16s * pDstMatrix);
IppStatus ippsToeplizMatrix_G729_16s32s(const Ipp16s * pSrcImpulseResponse,
   Ipp32s * pDstMatrix);
```

Arguments

pSrcImpulseResponse	Pointer to the impulse response vector [40], in Q12.
pDstMatrix	Pointer to the elements of Toepliz matrix of length 616, (of Ipp16s type in Q9 or Ipp32s type in Q25).

Discussion

The function ippsToeplizMatrix_G729 is declared in the ippsc.h file. This function calculates 616 elements in Toepliz matrix for the fixed-codebook search. These elements can be expressed as:

$$\Phi(i, j) = \sum_{n=j}^{39} h(n-i) \times h(n-j) \text{ , } 0 \le i \le 39 \text{ , } \quad 0 \le j \le 39$$

where $h(i)$, i = 0, 1, …, 39, is the impulse response.

The function stores the calculated 616 elements in *pDstMatrix* in the following order:

1. $\Phi(m_i, m_i)$ (i = 0, 1, 2, 3), 3x8+16 = 40 elements; starting location: 0.

   $\Phi(0, 0), \Phi(5, 5), \ldots, \Phi(35, 35)$, $\Phi(1, 1), \Phi(6, 6), \ldots, \Phi(36, 36)$,

   $\Phi(2, 2), \Phi(7, 7), \ldots, \Phi(37, 37)$, $\Phi(3, 3), \Phi(8, 8), \ldots, \Phi(39, 39)$.

2. $\Phi(m_0, m_1)$, 8x8 = 64 elements; starting location: 40.

   $\Phi(0, 1), \ldots, \Phi(0, 36)$, $\Phi(5, 1), \ldots, \Phi(5, 36)$, $\Phi(10, 1), \ldots, \Phi(35, 36)$.

3. $\Phi(m_0, m_2)$, 8x8 = 64 elements; starting location: 104.

   $\Phi(0, 2), \ldots, \Phi(0, 37)$, $\Phi(5, 2), \ldots, \Phi(5, 37)$, $\Phi(10, 2), \ldots, \Phi(35, 37)$.

4. $\Phi(m_0, m_3)$, 8x16 = 128 elements; starting location: 168.

   $\Phi(0, 3), \ldots, \Phi(0, 38)$, $\Phi(5, 3), \ldots, \Phi(5, 38)$, $\Phi(10, 3), \ldots, \Phi(35, 38)$.

   $\Phi(0, 4), \ldots, \Phi(0, 39)$, $\Phi(5, 4), \ldots, \Phi(5, 39)$, $\Phi(10, 4), \ldots, \Phi(35, 39)$.

5. $\Phi(m_1, m_2)$, 8x8 = 64 elements; starting location: 296.

   $\Phi(1, 2), \ldots, \Phi(1, 37)$, $\Phi(6, 2), \ldots, \Phi(6, 37)$, $\Phi(11, 2), \ldots, \Phi(36, 37)$.

6. $\Phi(m_1, m_3)$, 8x16 = 128 elements; starting location: 360.

   $\Phi(1, 3), \ldots, \Phi(1, 38)$, $\Phi(6, 3), \ldots, \Phi(6, 38)$, $\Phi(11, 3), \ldots, \Phi(36, 38)$.

   $\Phi(1, 4), \ldots, \Phi(1, 39)$, $\Phi(6, 4), \ldots, \Phi(6, 39)$, $\Phi(11, 4), \ldots, \Phi(36, 39)$.

7. $\Phi(m_2, m_3)$, 8x16 = 128 elements; starting location: 488.

   $\Phi(2, 3), \ldots, \Phi(2, 38)$, $\Phi(7, 3), \ldots, \Phi(7, 38)$, $\Phi(12, 3), \ldots, \Phi(37, 38)$.

   $\Phi(2, 4), \ldots, \Phi(2, 39)$, $\Phi(7, 4), \ldots, \Phi(7, 39)$, $\Phi(12, 4), \ldots, \Phi(37, 39)$.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcImpulseResponse* or *pDstMatrix* pointer is `NULL`.

Codebook Gain Functions

These functions can be used to predict, quantize, decode, and control fixed- and adaptive-codebook gains both in G.729 encoding and decoding procedures.

DecodeGain_G729

Decodes the adaptive and fixed-codebook gains.

```
IppStatus ippsDecodeGain_G729_16s (Ipp32s energy, Ipp16s *pPastEnergy, const
   Ipp16s *pQuaIndex, Ipp16s *pGain);
IppStatus ippsDecodeGain_G729I_16s(Ipp32s energy,  Ipp16s valGainAttenuation,
   Ipp16s *pPastEnergy, const Ipp16s *pQuaIndex, Ipp16s *pGain);
```

Arguments

energy	Input energy of the codeword.
pPastEnergy	Pointer to the input/output vector (in Q14) of the log-energies for fixed codebook contributions of the 4 previous subframes.
pQuaIndex	NULL for frame erasure; otherwise, pointer to the vector of codebook indices: *pQuaIndex* [0] - first stage codebook index , *pQuaIndex* [1] - second stage codebook index .
valGainAttenuation	Attenuation factor for the gains in case of frame erasuse (*pQuaIndex* = NULL).
pGain	Pointer to the input/output decoded gain: *pGain*[0] - adaptive (pitch) gain, *pGain*[1] - fixed codebook gain.

Discussion

The function ippsDecodeGain_G729 is declared in the ippsc.h file. This function decodes the adaptive and fixed-codebook gains. The fixed codebook gain g_c can be expressed as follows:

$g_c = \gamma \; g_c'$,

where g_c' is a predicted gain based on previous fixed codebook energies, and γ is a correction factor. The predicted gain g_c' is obtained by predicting the log-energy of the current fixed-codebook contribution from the log-energy of the previous fixed-codebook contribution, using the 4th order MA predictor with coefficients [0.68, 0.58, 0.34, 0.19].

The adaptive codebook gain and the factor are vector-quantized using conjugate structured codebooks. The first element in each codebook represents the quantized adaptive-codebook gain and the second element represents the quantized fixed-codebook. In case of frame erasure, the gains are the attenuated versions of the previous gains.

ippsDecodeGain_G729_16s. The two-stage conjugate structured two-dimensional codebook is used: first stage in 3 bits and second stage in 4 bits. In case of frame erasure, factors 0.9 and 0.98 98 in Q15 are used for attenuation of the adaptive and fixed codebook gains, respectively.

ippsDecodeGain_G729I_16s. The new 6-bit conjugate structured codebook is used. In case of frame erasure, the factor given by *valGainAttenuation* is used for attenuation of both the adaptive and fixed codebook gains.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pPastEnergy* or *pGain* pointer is NULL.

GainControl_G729

Calculates adaptive gain control.

```
IppStatus ippsGainControl_G729_16s_I (const Ipp16s* pSrc, Ipp16s* pSrcDst,
   Ipp16s* pGain);
IppStatus ippsGainControl_G729A_16s_I (const Ipp16s* pSrc, Ipp16s* pSrcDst,
   Ipp16s* pGain);
```

Arguments

pSrc	Pointer to the source reconstructed speech vector [40].
pSrcDst	Pointer to the source post filtered signal and destination gain-scaled post filtered vector [40].
pGain	Pointer to the output adaptive gain.

Discussion

These functions are declared in the `ippsc.h` file. The function `ippsGainControl_G729` compensates the gain difference between the reconstructed speech signal *sr*, given by *pSrc* , and the filtered signal *sf*, given by *pSrcDst*. First, the gain factor *G* is calculated as follows:

$$G = \frac{\sum_{i=0}^{39} |sr[i]|}{\sum_{i=0}^{39} |sf[i]|}$$

The output gain *spf* for the post-filtered signal *pSrcDst* is calculated as:

$spf(n) = g^{(n)} \cdot sf(n),\ n = 0,..., 39,$

where

$g^{(n)} = 0.85 \cdot g^{(n-1)} + 0.15 \cdot G\ , n = 0,..., 39,\ g^{(-1)} = 1.0\ .$

The function returns $g^{(39)}$ in *pGain*.

For the function `ippsGainControl_G729A_16s`, the gain factor *G* and factors $g^{(n)}$ are calculated differently as given by:

$$G = \sqrt{\frac{\sum_{i=0}^{39} |sr[i]^2|}{\sum_{i=0}^{39} |spf[i]^2|}}$$

$$g^{(n)} = 0.9 \cdot g^{(n-1)} + 0.1 \cdot G$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSrcDst* or *pGain* pointer is NULL.

GainQuant_G729

Quantizes the codebook gain using a two-stage conjugate-structured codebook.

```
IppStatus ippsGainQuant_G729_16s(const Ipp16s * pSrcAdptTarget, const Ipp16s *
   pSrcFltAdptVector, const Ipp16s * pSrcFixedVector, const Ipp16s
   *pSrcFltFixedVector, Ipp16s *pSrcDstEnergyErr, Ipp16s * pDstQGain, Ipp16s *
   pDstQGainIndex, Ipp16s tameProcess);
IppStatus ippsGainQuant_G729D_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
   pSrcFltAdptVector, const Ipp16s* pSrcFixedVector, const Ipp16s
   *pSrcFltFixedVector, Ipp16s* pSrcDstEnergyErr, Ipp16s *pDstQGain, Ipp16s
   *pDstQGainIndex, Ipp16s tameProcess)
```

Arguments

pSrcAdptTarget	Pointer to the target signal for adaptive codebook search vector [40], in Q11.
pSrcFltAdptVector	Pointer to the filtered adaptive codebook vector [40], in Q11.
pSrcFixedVector	Pointer to the pre-weighted fixed-codebook vector [40], in Q12.
pSrcFltFixedVector	Pointer to the filtered fixed codebook vector [40], in Q12.
pSrcDstEnergyErr	Pointer to the previous predicted energy error vector [4], in Q10.

`pDstQGain`	Pointer to the quantized gain (G_p and G_c).
`pDstQGainIndex`	Pointer to the gain codebook indices, *GA* and *GB*.
`tameProcess`	Taming process indicator.

Discussion

The functions `ippsGainQuant_G729` and `ippsGainQuant_G729D` are declared in the `ippsc.h` file. These functions use a two-stage conjugate-structured codebook. The function `ippsGainQuant_G729_16s` quantizes the gain using the 7-bit codebook, whereas the function `ippsGainQuant_G729D_16s` uses the 6-bit codebook. The gain codebook search is done by minimizing the mean-squared weighted error between original and reconstructed speech:

$$E = \mathbf{x}^t\mathbf{x} + g_p^2\mathbf{y}^t\mathbf{y} + g_c^2\mathbf{z}^t\mathbf{z} - 2g_p\mathbf{x}^t\mathbf{y} - 2g_c\mathbf{x}^t\mathbf{z} + 2g_pg_c\mathbf{y}^t\mathbf{z} \quad ,$$

where $\mathbf{x}$ is the adaptive target, $\mathbf{y}$ is the filtered adaptive vector, and $\mathbf{z}$ is the filtered fixed vector; g_p is the pitch gain, g_c is the fixed gain.

The functionality is as follows:

1. Calculate the average energy minus the mean energy of the fixed-codebook contribution:

$$E = 10\log\left(\frac{1}{40}\sum_{n=0}^{39} c(n)^2\right)$$

Then calculate $Eb - E = 30(dB) - E$.

2. Calculate the predicted energy from prediction error of the previous subframes:

$$\tilde{E}^m = \sum_{i=1}^{4} b_i \hat{U}^{(m-i)}$$

$\hat{U}^m$ is calculated by the equation: $U^m = E - \tilde{E}^m$.

3. Calculate g_c by the equation: $g_c = 10^{(\tilde{E}^m + Eb - E)/20}$.

4. Calculate the following elements:

$G0 = \mathbf{y}^t\mathbf{y}$, $G1 = -2\mathbf{x}^t\mathbf{y}$, $G2 = \mathbf{z}^t\mathbf{z}$, $G3 = -2\mathbf{x}^t\mathbf{z}$, $G4 = 2\mathbf{y}^t\mathbf{z}$.

5. Calculate optimum gain G_p and G_c as:

$$G_p = -\frac{2\times G1\times G2 - G3\times G4}{4\times G0\times G2 - G4\times G4} \quad , \quad G_c = -\frac{2\times G0\times G3 - G1\times G4}{4\times G0\times G2 - G4\times G4}$$

6. Pre-select a domain in which G_p and G_c are most possibly located. The output is *index*0 and *index*1.

7. Select the gain vector in the codebook *GA* and *GB*, the range is limited by *index*0 and *index*1. The search criteria is minimizing the following expression:

$$E = G0\times G_p^2 + G2\times G_c^2 + G1\times G_p + G3\times G_c + G4\times G_p\times G_c \ .$$

8. Update U^m.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcAdptTarget`*, *`pSrcFltAdptVector`*, *`pSrcFixedVector`*, *`pSrcFltFixedVector`*, *`pSrcDstEnergyErr`*, *`pDstQGain`* or *`pDstQGainIndex`* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *`tameProcess`* is not 0 or 1.

AdaptiveCodebookContribution_G729

Updates target vector for codebook search by subtracting adaptive codebook contribution.

```
IppStatus ippsAdaptiveCodebookContribution_G729_16s (Ipp16s gain, const
   Ipp16s* pFltAdptVector, const Ipp16s* pSrcAdptTarget, Ipp16s*
   pDstAdptTarget);
```

Arguments

`gain`	Adaptive codebook gain g_p.
`pFltAdptVector`	Pointer to the input filtered adaptive codebook vector $y(n)$.

pSrcAdptTarget	Pointer to the input target vector $x(n)$.
pDstAdptTarget	Pointer to the output updated target vector $x'(n)$.

Discussion

The function ippsAdaptiveCodebookContribution_G729 is declared in the ippsc.h file. This function subtracts the adaptive codebook contribution from the target signal and stores updated target vector as follows:

$x'(n) = x(n) - g_p \cdot y(n)$, $n = 0,..., 39$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pFltAdptVector* or *pSrcAdptTarget* or *pDstAdptTarget* pointer is NULL.

AdaptiveCodebookGain_G729

Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.

```
IppStatus ippsAdaptiveCodebookGain_G729_16s(const Ipp16s * pSrcAdptTarget,
   const Ipp16s * pSrcImpulseResponse, const Ipp16s * pSrcAdptVector, Ipp16s *
   pDstFltAdptVector, Ipp16s * pResultAdptGain);
IppStatus ippsAdaptiveCodebookGain_G729A_16s(const Ipp16s * pSrcAdptTarget,
   const Ipp16s * pSrcLPC, const Ipp16s * pSrcAdptVector, Ipp16s *
   pDstFltAdptVector, Ipp16s * pResultAdptGain);
```

Arguments

pSrcAdptTarget	Pointer to the adaptive target signal vector [40].
pSrcImpulseResponse	Pointer to the impulse response of the perceptual weighting filter vector [40], in Q12.

pSrcAdptVector	Pointer to the adaptive-codebook vector [40], in Q12 (for `ippsAdaptiveCodebookGain_G729_16s`) or in Q10 (for `ippsAdaptiveCodebookGain_G729A_16s`).
pSrcLPC	Pointer to the LPC coefficients of the synthesis filter vector [11], in Q12.
pDstFltAdptVector	Pointer to the filtered adaptive-codebook vector [40].
pResultAdptGain	Pointer to the adaptive-codebook gain, in Q14.

Discussion

These functions are declared in the `ippsc.h` file. They compute the gain of the adaptive-codebook vector, and calculate the filtered adaptive-codebook vector, respectively. Both functions are applied in subframes.
The first function `ippsAdaptiveCodebookGain_G729` is designed for G.729/B codec and implements the following functionality:

1. First, convolve the adaptive-codebook vector, $v(n)$, with the impulse response, $h(n)$, to obtain the filtered adaptive-codebook vector, $y(n)$, using the formula:

$$y(n) = \sum_{i=0}^{n} v(i)h(n-i) \ , n = 0,1,...,39 \ .$$

2. Next, calculate the adaptive-codebook gain g_p as given by:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)} \ ,$$

where $x(n)$ is the adaptive target signal.

3. Finally, bound the adaptive-codebook gain g_p to the range [0, 1.2].

The second function `ippsAdaptiveCodebookGain_G729A` is designed for G.729A codec and implements the following functionality:

1. First, pass the adaptive-codebook vector, $v(n)$, through the synthesis filter, $1/A_q(z/\gamma)$, to obtain the filtered adaptive-codebook vector, $y(n)$, using the formula :

$$y(n) = v(n) - \sum_{i=1}^{10} a_i y(n-i), \quad n = 0,1,...,39 .$$

2. Next, calculate the adaptive-codebook gain g_p as:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)},$$

where $x(n)$ is the adaptive target signal.

3. Finally, bound the adaptive-codebook gain g_p to the range [0, 1.2].

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcAdptTarget`*, *`pSrcImpulseResponse`*, *`pSrcAdptVector`*, *`pSrcLPC`*, *`pDstFltAdptVector`*, or *`pResultAdptGain`* pointer is `NULL`.

Filter Functions

These functions are used to perform different types of filtering, including high pass filtering on pre-processing stage of encoding and post-processing stage of decoding, as well as the long-term and short-term postfilter in decoding.

The residual (FIR), simple synthesis (IIR), harmonic, and preemphasize filter functions may be combined to perform more complex synthesis filtering and may be used in different encode and decode stages.

ResidualFilter_G729

Implements an inverse LP filter and obtains the residual signal.

```
IppStatus ippsResidualFilter_G729_16s(const Ipp16s * pSrcSpch, const Ipp16s *
   pSrcLPC, Ipp16s * pDstResidual);
IppStatus ippsResidualFilter_G729E_16s(const Ipp16s *pCoeffs, int order, const
   Ipp16s *pSrc, Ipp16s *pDst, int len)
```

Arguments

pSrcSpch	Pointer to the input speech signal. Elements *pSrcSpch*[0...39] are the present speech signals, whereas *pSrcSpch*[-10... -1] are the history to be used.
pSrcLPC	Pointer to the LP coefficients vector [11], in Q12.
pDstResidual	Pointer to the LP residual.
pCoeffs	Pointer to the vector [*order*+1] of filter coefficients.
order	The filter order.
pSrc	Pointer to the source vector [*len*]. Values *pSrc* [-*order*,..,-1] must also be supplied.
pDst	Pointer to the output filtered residual vector [len].
len	Length of the source and destination vectors.

Discussion

The functions `ippsResidualFilter_G729` and `ippsResidualFilter_G729E` are declared in the `ippsc.h` file.

ippsResidualFilter_G729. This function filters the input speech signal through an inverse LP filter and gets the residual signal as:

$$r(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i \times s(n-i) \ , n = 0,1,...,39 \ .$$

ippsResidualFilter_G729E. This function implements the same operation as the previous function, but accepts the variable filter order.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcSpch*, *pSrcLPC*, *pDstResidual*, *pSrc*, *pCoeffs*, or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *order* is less than or equal to 0.

SynthesisFilter_G729

Reconstructs the speech signal from LP coefficients and residuals.

```
IppStatus ippsSynthesisFilter_G729_16s(const Ipp16s * pSrcResidual, const
   Ipp16s * pSrcLPC, Ipp16s * pSrcDstSpch);
IppStatus ippsSynthesisFilterZeroStateResponse_NR_16s(const Ipp16s* pSrcLPC,
   Ipp16s* pDstImp, int len, int scaleFactor);
IppStatus ippsSynthesisFilter_G729E_16s(const Ipp16s *pLPC, int order, const
   Ipp16s *pSrc, Ipp16s *pDst, int len, const Ipp16s *pMem);
IppStatus ippsSynthesisFilter_G729E_16s_I(const Ipp16s *pLPC, int order,
   Ipp16s *pSrcDst, int len, const Ipp16s *pMem);
```

Arguments

pSrcResidual	Pointer to the LP residual signal vector [40].
pSrcLPC	Pointer to the LP coefficients vector [11], in Q12.

pSrcDstSpch	Pointer to the synthesized and updated speech. Elements *pSrcDstSpch*[0...39] are the present synthesized speech, *pSrcDstSpch*[-10...-1] are the history to be used.
pLPC	Pointer to the LP coefficients vector [*order*+1].
order	The filter order (*pLPC*[*order*+1] vector must be supplied on input) .
pSrc	Pointer to the source vector [*len*].
pDst	Pointer to the destination speech vector [*len*].
pSrcDst	Pointer to the source and destination vector [*len*].
pDstImp	Pointer to the destination zero state impulse response vectore [*len*]
len	Length of the source and destination vectors.
pMem	Pointer to the filter memory vector [*order*] supplied for filtering.
scaleFactor	Scale factor for the input LP coefficients and output impulse response, accordingly.

Discussion

These functions are declared in the `ippsc.h` file.

The function `ippsSynthesisFilter_G729_16s` uses the default LP filter of 10-th order to reconstruct speech signal from the residual signal as:

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) \quad , n = 0,1,...,39 .$$

The function `ippsSynthesisFilterZeroStateResponse_NR_16s` performs the same operation as `ippsSynthesisFilter_NR_16s` for the same parameters, but uses no memory and processes the impulse input vector *pSrc* which is set as

pSrc[0] = $2^{scaleFactor}$,
pSrc[i] = 0 , i = 1,... *len* - 1 .

The functions ippsSynthesisFilter_G729E_16s and ippsSynthesisFilter_G729E_16s_I perform synthesis filtering of the order given by the parameter *order* and operate like ippsSynthesisFilter_G729_16s function for the default order 10.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the pointers is NULL.
ippStsOverflow	Indicates a warning that an overflow has occurred.
ippStsSizeErr	Indicates an error when *order* is less than or equal to 0.

LongTermPostFilter_G729

Restores the long-term information from the old speech signal.

```
IppStatus ippsLongTermPostFilter_G729_16s (Ipp16s gammaFactor, int valDelay,
   const Ipp16s *pSrcDstResidual, Ipp16s *pDstFltResidual,
   Ipp16s *pResultVoice);
IppStatus ippsLongTermPostFilter_G729A_16s(Ipp16s valDelay, const Ipp16s *
   pSrcSpch, const Ipp16s * pSrcLPC, Ipp16s * pSrcDstResidual, Ipp16s *
   pDstFltResidual);
IppStatus ippsLongTermPostFilter_G729B_16s(Ipp16s valDelay, const Ipp16s *
   pSrcSpch, const Ipp16s * pSrcLPC, Ipp16s * pSrcDstResidual, Ipp16s *
   pDstFltResidual, Ipp16s * pResultVoice, Ipp16s frameType);
```

Arguments

gammaFactor	Post filter coefficient y_p in Q15.
valDelay	Pitch delay.
pSrcSpch	Pointer to the reconstructed speech $\hat{s}(n)$, in Q15. Elements *pSrcSpch*[0...39] are the present speech signals, elements *pSrcSpch*[-10...-1] are the history that will be used.

pSrcLPC	Pointer to the LP coefficients $\hat{a}_i$ vector [11], in Q12.
pSrcDstResidual	Pointer to the residual after long term prediction vector [40], elements [-152...-1] are the long term prediction history that will be used.
pDstFltResidual	Pointer to the output filtered residual vector [40].
pResultVoice	Pointer to the voice information.
frameType	The type of the frame: 0 - stands for untransmitted frame, 1 - stands for normal speech frame, 2 - stands for SID frame.

Discussion

These functions are declared in the `ippsc.h` file. These FIR filters are used to restore the long-term relationship from old speech at the length of pitch. They are applied in subframes.

The first function **`ippsLongTermPostFilter_G729`** has the following functionality:

1. First, calculate autocorrelation $R(k)$ of $\hat{r}(n)$ and find the best integer $T0$, in the range of [int($T1$)-1, int($T1$)+1], which maximizes $R(k)$ using the following formula:

$$R(k) = \sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}(n-k) \qquad (9\text{-}6)$$

Here int($T1$) is the integer part of the pitch delay $T1$ in the first subframe.

2. Second, seek for the best fractional delay T with resolution 1/8 around $T0$. This is done by finding the delay with the highest pseudo-normalized correlation $R'(k)$:

$$R'(k) = \frac{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}_k(n)}{\sqrt{\sum_{n=0}^{39} \hat{r}_k(n) \cdot \hat{r}_k(n)}},$$

where $\hat{r}_k(n)$ is the residual at delay *k*.

3. Once the optimal delay *T* is found, determine whether the long-term filter should be disabled or not, using the following condition:

$$\text{if } \frac{R'(T)^2}{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}(n)} < 0.5 \text{ , let } g_l = 0 ; \tag{9-7}$$

$$\text{otherwise, let } g_l = \frac{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}_k(n)}{\sum_{n=0}^{39} \hat{r}_k(n) \cdot \hat{r}_k(n)} \text{ , and bound it to [0, 1.0]} \tag{9-8}$$

4. Finally, use the following FIR filter:

$$y(n) = \frac{1}{1 + \gamma_p \cdot g_l}(\hat{r}(n) + \gamma_p \cdot g_l \cdot \hat{r}(n - T)) , \tag{9-9}$$

where $\hat{r}(n)$ is the residual signal and the factor γ_p controls the amount of long-term post-filtering.

The second function **`ippsLongTermPostFilter_G729A`** is designed for G.729A codec, and performs the following operations:

1. First, get the residual $\hat{r}(n)$ from synthesized speech $\hat{s}(n)$ using the formula:

$$\hat{r}(n) = \hat{s}(n) + \sum_{i=1}^{10} \gamma_n^i \cdot \hat{a}_i \cdot \hat{s}(n - i) , \tag{9-5}$$

where $\hat{s}(n)$ is the synthesized speech, $\hat{a}_i$ are the quantized LP coefficients and $\gamma_n = 0.55$.

2. Second, calculate autocorrelation $R(k)$ of $\hat{r}(n)$ and find the best integer *T* in the range [int(*T*1)-3, int(*T*1)+3], which maximizes $R(k)$ using the formula (9-6).

Here int(*T*1) is the integer part of the pitch delay *T*1 in the first subframe.

3. Once the optimal delay *T* is found, determine whether the long-term filter should be disabled or not, using the relations (9-7) and (9-8).

6. Finally, use the FIR filter (9-9).

The third function **ippsLongTermPostFilter_G729B** is actually the combination of other functions as demonstrated by the code below:

```
{
   const short g_pst[11] = {32767,18022,9912,5451,2998,1649,907,499,274,151,83};
      /* y_n = 0.55 powered by i=0,1,…,10 in Q15 */
   short coeffs[LP_ORDER+1];          /* temporary nominator coefficients  */
   short vc;

   ippsMul_NR_16s_Sfs(g_pst, pSrcLpc, coeffs, LP_11, 15);
   ippsResidualFilter_G729_16s(pSrcSpch, coeffs, pSrcDstResidual+154);
   if (1 == frameType){
      ippsLongTermPostFilter_G729_16s (16384,valDelay, pSrcDstResidual + 154,
                pDstFltResidual, &vc); /* Harmonic filtering : g_p=0.5 in Q15*/
      *pResultVoice = (vc != 0);
   }else{
      ippsCopy_16s(pSrcDstResidual + 154,pDstFltResidual,40);
      *pResultVoice = 0;
   }
}
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcSpch`*, *`pSrcLPC`*, *`pSrcDstResidual`*, *`pDstFltResidual`* or *`pResultVoice`* pointer is `NULL`.

ippStsRangeErr	Indicates an error when *valDelay* is not in the range [18, 145] for ippsLongTermPostFilter_G729A function, or *valDelay* is not in the range [0, 143] for ippsLongTermPostFilter_G729B function, or *frameType* is not in the range [0, 2].

ShortTermPostFilter_G729

Restores speech signal from the residuals.

```
IppStatus ippsShortTermPostFilter_G729_16s(const Ipp16s * pSrcLPC, const
   Ipp16s * pSrcFltResidual, Ipp16s * pSrcDstSpch, Ipp16s *
   pDstImpulseResponse);
IppStatus ippsShortTermPostFilter_G729A_16s(const Ipp16s * pSrcLPC, const
   Ipp16s * pSrcFltResidual, Ipp16s * pSrcDstSpch);
```

Arguments

pSrcLPC	Pointer to the quantized LP coefficients vector [11], in Q12.
pSrcFltResidual	Pointer to the residual signal $x(n)$ vector [40], in Q15.
pSrcDstSpch	Pointer to the short-term filtered speech $y(n)$, in Q15. Elements *pSrcDstSpch*[0...39]are the present short-term filtered speech signals, elements *pSrcDstSpch*[-10...-1] are the history that will be used.
pDstImpulseResponse	Pointer to the generated impulse response $h_f(n)$ vector [20], in Q12.

Discussion

These functions are declared in the ippsc.h file. These two IIR filters restore the speech from the residual. They both are applied in subframes.

The first function ippsShortTermPostFilter_G729 is designed for G.729/B codec, and has the following functionality:

1. First, compute the impulse response $h_f(n)$ of the inverse filter. $\hat{A}(z)$.

2. Next, calculate the gain term g_f using the formula:

$$g_f = \sum_{0}^{10} |h_f(n)|$$

3. Finally, filter the residual through the inverse filter $\hat{A}(z)$, as follows:

$$y(n) = \frac{1}{g_f} x(n) - \sum_{i=1}^{10} \hat{a}_i y(n-i), \; n = 0,1,...,39,$$

where $x(n)$ is the residual and $\hat{a}_i$ are the weighted quantized LP coefficients.

The second function `ippsShortTermPostFilter_G729A` is designed for G.729A codec. This function filters the residual through the inverse filter $\hat{A}(z)$, using the formula:

$$y(n) = x(n) - \sum_{i=1}^{10} \hat{a}_i y(n-i), \; n = 0,1,...,39,$$

where $x(n)$ is the residual and $\hat{a}_i$ are the weighted quantized LP coefficients.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcLPC`*, *`pSrcFltResidual`*, *`pSrcDstSpch`*, or *`pDstImpulseResponse`* pointer is `NULL`.

TiltCompensation_G729

Compensates for the tilt in the short-term filter.

```
IppStatus ippsTiltCompensation_G729_16s(const Ipp16s * pSrcImpulseResponse,
   Ipp16s * pSrcDstSpch);
IppStatus ippsTiltCompensation_G729A_16s(const Ipp16s * pSrcLPC, Ipp16s *
   pSrcDstFltResidual);
IppStatus ippsTiltCompensation_G729E_16s(Ipp16s val, const Ipp16s *pSrc,
   Ipp16s *pDst);
```

Arguments

pSrcImpulseResponse	Pointer to the impulse response $h_f(n)$ vector [20], in Q12.
pSrcLPC	Pointer to the gamma weighted LP coefficients vector [22], in Q12. The first 11 elements refer to γ_n and the next 11 elements refer to γ_d.
pSrcDstSpch	Pointer to the present and tilt-compensated speech $x(n)$, in Q15. Elements *pSrcDstSpch*[0...39] are the present speech signals, element *pSrcDstSpch*[-1] is the history that will be used.
pSrcDstFltResidual	Pointer to the long-term filtered LP residual, in Q15. Elements *pSrcDstFltResidual*[0...39] are the present long-term filtered LP residual signals, element *pSrcDstFltResidual*[-1] is the history that will be used.
val	The input tilt factor.
pSrc	Pointer to the source vector [41].
pDst	Pointer to the output filtered residual vector [40]

Discussion

These functions are declared in the `ippsc.h` file. These FIR filters compensate for the tilt in the short-term filter. They are applied in subframes.

The function ippsTiltCompensation_G729 is designed for G.729/B codec, and the functionality is as follows.

1. First, calculate correlation $r_h(i)$ of the impulse response $h_f(i)$ as:

$$r_h(i) = \sum_{j=0}^{19-i} h_f(j)h_f(j+i)$$

Then obtain the tilt factor k_1' using the relation:

$$k_1' = -\frac{r_h(1)}{r_h(0)}$$

2. Next, obtain the gain term g_t using the formula:

$$g_t = 1 - |\gamma_t \cdot k_1'|,$$

where $\gamma_t = 0.9$ if k_1' is negative, and $\gamma_t = 0.2$ if k_1' is positive.

3. Finally, filter the speech using the formula:

$$y(n) = \frac{1}{g_t}(x(n) + \gamma_t \cdot k_1' \cdot x(n-1)),$$

where $x(n)$ is the input speech signal.

The function ippsTiltCompensation_G729E performs only stages 2 and 3 for a given input tilt factor *val* and filter memory specified by *pSrc*[0].

The function ippsTiltCompensation_G729A is designed for G.729A codec, and the functionality is as follows.

1. First, compute the impulse response $h_f(n)$ of the inverse filter. $\hat{A}(z)$.

2. Next, calculate correlation $r_h(i)$ of the impulse response $h_f(i)$ as:

$$r_h(i) = \sum_{j=0}^{21-i} h_f(j)h_f(j+i)$$

Then obtain the tilt factor k_1', using the relation:

$$k_1' = -\frac{r_h(1)}{r_h(0)}$$

3. Finally, filter the residual using the formula:

$y(n) = x(n) + \gamma_t \cdot k_1' \cdot x(n-1)$,

where $x(n)$ is the residual, and $\gamma_t = 0.8$ if k_1' is negative; or $\gamma_t = 0$ if k_1' is positive.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcDstSpch*, *pSrcImpulseResponse*, *pSrcLPC*, *pSrc*, *pDst*, or *pSrcDstFltResidual* pointer is `NULL`.

HarmonicFilter

Calculates the harmonic filter.

```
IppStatus ippsHarmonicFilter_16s_I (Ipp16s beta, int T, Ipp16s* pSrcDst,
   int len);
IppStatus ippsHarmonicFilter_NR_16s (Ipp16s beta, int T, const Ipp16s * pSrc,
   Ipp16s * pDst, int len);
```

Arguments

beta	The beta factor, in Q15.
T	The integer delay.
pSrc	Pointer to the source vector. Elements *pSrc*[-*T*,...,0,...*len*-1] will be used as input.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector. Elements *pSrcDst*[-*T*,...,0,...,*len*-1] will be used as input, elements *pSrcDst*[0,...,*len*-1] will be computed.
len	Number of elements in the source and destination vectors.

Discussion

These functions are declared in the `ippsc.h` file.

The function `ippsHarmonicFilter_16s_I` implements the adaptive pre-filter:

$$H(z) = \frac{1}{1 - \beta \cdot z^{-T}},$$

which can be used to enhance the harmonic component of speech.

The calculation is as follows:

$pSrcDst[n] = pSrcDst[n] + beta \cdot pSrcDst[n - T],\ 0 \le n < len$.

The function `ippsHarmonicFilter_NR_16s` implements the harmonic noise shaping filter:

$$H(z) = 1 + \beta \cdot z^{-T}$$

The calculation is as follows:

$pDst[n] = pSrc[n] + beta \cdot pSrc[n - T],\ 0 \le n < len$.

Both filters can be used to enhance the harmonic component of the input signal.

Return Value

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

HighPassFilterSize_G729

Returns the G729 high-pass filter size.

```
IppStatus ippsHighPassFilterSize_G729 (int *pSize);
```

Arguments

pSize	Pointer to the output value of the memory size.

Discussion

The function `ippsHighPassFilterSize_G729` is declared in the `ippsc.h` file. This function returns the size of memory required for the `ippsHighPassFilterInit_G729` function to operate.

Return Value

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

HighPassFilterInit_G729

Initializes the high-pass filter.

```
IppStatus ippsHighPassFilterInit_G729 (Ipp16s *pCoeff, char*pMemUpdated);
```

Arguments

`pCoeff`	Pointer to the array of length 6 containing values a_0, a_1, a_2, b_0, b_1, b_2, in Q12 or Q13.
`pMemUpdated`	Pointer to the memory allocated for the filter.

Discussion

The function `ippsHighPassFilterInit_G729` is declared in the `ippsc.h` file. This function initializes the internal data of the high-pass filter: a_0 should be equal to scaled 1 (in Q12 or Q13). The filter history data x_{-2}, x_{-1}, y_{-2}, y_{-1} are set to zero. The `ippsHighPassFilter_G729_16s_ISfs` function uses this memory for filtering.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pCoeff`* or *`pMemUpdated`* pointer is `NULL`.

HighPassFilter_G729

Performs G729 high-pass filtering.

```
IppStatus ippsHighPassFilter_G729_16s_ISfs (Ipp16s* pSrcDst, int len, int
   scaleFactor, char* pMemUpdated);
```

Arguments

pSrcDst	Pointer to the source and destination vector [*len*].
len	Number of elements in the source and destination vectors.
scaleFactor	Scale factor for output data scaling.
pMemUpdated	Pointer to the memory allocated for the filter.

Discussion

The function ippsHighPassFilter_G729 is declared in the ippsc.h file. This function performs the input signal pre-processing or the output signal post-processing using the high-pass filter:

$$H_h(z) = \frac{b_0 + b_1 \cdot z^{-1} + b_2 \cdot z^{-2}}{a_0 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2}}$$

Currently, only $a_0 = 1$ (in Q12 or Q13) is supported. The function uses the scale factor value equal to 12 for input data pre-filtering, and equal to 13 for output data post-filtering.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* or *pMemUpdated* pointer is NULL.
ippStsScaleRangeErr	Indicates an error when *scaleFactor* is not equal to 12 or 13.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

IIR16s_G729

Performs IIR filtering.

```
IppStatus ippsIIR16sLow_G729_16s(const Ipp16s *pCoeffs, const Ipp16s *pSrc,
   Ipp16s *pDst, Ipp16s *pMem);
IppStatus ippsIIR16s_G729_16s(const Ipp16s *pCoeffs, const Ipp16s *pSrc,
   Ipp16s *pDst, Ipp16s *pMem);
```

Arguments

pCoeffs	Pointer to the input vector of filter coefficients vector [20]: $b_1,\ldots, b_{10}, a_1, \ldots,a_{10}$.
pSrc	Pointer to the input signal vector [40].
pDst	Pointer to the output filtered vector [40].
pMem	Pointer to the filter memory vector [20]. This vector must be initially filled with zeroes.

Discussion

The functions `ippsIIR16sLow_G729s_16s` and `ippsIIR16s_G729_16s` are declared in the `ippsc.h` file.

These functions perform infinite impulse response (IIR) filtering using the following transfer function:

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The function `ippsIIR16sLow_G729_16s` does not check for overflow and result saturation, whereas the function `ippsIIR16s_G729_16s` makes these checks.

The filter memory is updated.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pCoeffs*, *pSrc*, *pDst*, or *pMem* pointer is NULL.

PhaseDispersionGetStateSize_G729D

Queries the memory length of the phase dispersion filter.

```
IppStatus ippsPhaseDispersionGetStateSize_G729D_16s (int *pSize);
```

Arguments

pSize	Pointer to the output phase dispersion filter memory length in bytes.

Discussion

The function ippsPhaseDispersionGetStateSize_G729D_16s reports the size of memory that the phase dispersion filter needs for proper operation.

Return Value

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

PhaseDispersionInit_G729D

Initializes the phase dispersion filter memory.

```
IppStatus ippsPhaseDispersionInit_G729D_16s
   (IppsPhaseDispersion16s_State_G729D* pPhDMem);
```

Arguments

pPhDMem	Pointer to the memory of the phase dispersion filter.

Discussion

The function ippsPhaseDispersionInit_G729D_16s sets the contents of the given buffer as memory of the phase dispersion filter in initial state.

Return Value

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

PhaseDispersionUpdate_G729D

Updates the phase dispersion filter state.

```
IppStatus ippsPhaseDispersionUpdate_G729D_16s(Ipp16s valPitchGain,
   Ipp16s valCodebookGain, IppsPhaseDispersion16s_State_G729D *pPhDMem);
```

Arguments

valPitchGain	Long term pitch gain.
valCodebookGain	Codebook gain
pPhDMem	Pointer to the memory of phase dispersion filter.

Discussion

The function ippsPhaseDispersionUpdate_G729D_16s updates the state of phase dispersion filter memory with given pitch and codebook gains.

Return Value

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

PhaseDispersion_G729D

Performs the phase dispersion filtering.

```
IppStatus ippsPhaseDispersion_G729D_16s(Ipp16s valCodebookGain, Ipp16s
   valPitchGain,  const Ipp16s *pSrcExcSignal, Ipp16s *pDstFltExcSignal,
   Ipp16s *pSrcDstInnovation, IppsPhaseDispersion16s_State_G729D *pPhDMem)
```

Arguments

valPitchGain	Input long term pitch gain.
valCodebookGain	Input codebook gain
pSrcExcSignal	Pointer to the input excitation vector [40].
pDstFltExcSignal	Pointer to the output filtered excitation vector [40].
pSrcDstInnovation	Pointer to the input/output innovative codebook vector [40].
pPhDMem	Pointer to the memory of phase dispersion filter.

Discussion

The function ippsPhaseDispersion_G729D_16s performs the phase dispersion filtering for the given pitch and codebook gains. The filter alters the innovation signal (mainly its phase), such that a new innovation signal is created with the energy being more spread over the subframe. The filtering is performed by a circular convolution using one of the three "semi-random" impulse responses according to the different amounts of spreading.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcExcSignal*, *pDstFltExcSignal*, *pSrcDstInnovation* or *pPhDMem* pointer is NULL.

Preemphasize_G729A

Computes pre-emphasis of a post filter.

```
IppStatus ippsPreemphasize_G729A_16s (Ipp16s gamma, const Ipp16s *pSrc, Ipp16s
   *pDst, int len, Ipp16s* pMem);
IppStatus ippsPreemphasize_G729A_16s_I (Ipp16s gamma, Ipp16s* pSrcDst, int
   len, Ipp16s* pMem);
```

Arguments

gamma	The filter coefficient, in Q15.
pSrc	Pointer to the source vector, in Q0.
pDst	Pointer to the destination vector, in Q0.
pSrcDst	Pointer to the source and destination vector, in Q0.
len	Number of elements in the source and destination vectors.
pMem	Pointer to the filter memory [1].

Discussion

The function `ippsPreemphasize_G729A` is declared in the `ippsc.h` file. This function computes pre-emphasis of a post filter.

The computation is performed according to the difference signal pre-emphasis equation:

$$H(z) = 1 - gamma \cdot z^{-1}$$

The filter memory *pMem* should be initialized to zero and then be constantly updated while filtering.

The function `ippsPreemphasize_G729A` performs no rounding (clipping).

Return Value

`ippStsNoErr`	Indicates no error.

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

WinHybridGetStateSize_G729E

Queries the length of the hybrid windowing module memory.

```
IppStatus ippsWinHybridGetStateSize_G729E_16s (int *pSize);
```

Arguments

pSize	Pointer to the output length of hybrid windowing module memory in bytes.

Discussion

The function ippsWinHybridGetStateSize_G729E reports the size of memory that the hybrid windowing module needs for proper operation.

Return Value

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

WinHybridInit_G729E

Initializes the hybrid windowing module memory.

```
IppStatus ippsWinHybridInit_G729E_16s (IppsWinHybridState_G729E_16s*
   pHybWinMem);
```

Arguments

pHybWinMem	Pointer to the memory of hybrid windowing module.

Discussion

The function ippsWinHybridInit_G729E sets the contents of the given buffer as the memory of the hybrid windowing module in initial state.

Return Value

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

WinHybrid_G729E

Applies the hybrid window and computes autocorrelation coefficients.

```
IppStatus ippsWinHybrid_G729E_16s32s (const Ipp16s* pSrcSynthSpeech, Ipp32s*
   pDstInvAutoCorr, IppsWinHybridState_G729E_16s* pHybWinMem);
```

Arguments

pSrcSynthSpeech	Pointer to the input synthesized speech vector [144].
pDstInvAutoCorr	Pointer to the output autocorrelation vector [31].
pHybWinMem	Pointer to the memory of hybrid windowing module.

Discussion

The function ippsWinHybrid_G729E applies the hybrid window to input synthesized speech and computes autocorrelation coefficients as follows.

1. First, the input speech vector is multiplied by the hybrid window:

$$s_{w+I}(k) = s_w(k) \cdot w(144 - k), k = 0,\ldots,144$$

where

$$w(k) = \begin{cases} \sin((k+1)*0.047783), k = 0,\ldots,34 \\ \sin(36*0.047783)*a^{(35-k)}, k = 35,\ldots,144 \end{cases}$$

Here a = 0.9928337491 so that a^{40}=0.75 and $a^{160} = 0.75^4 = 0.31640625$.

2. Second, the autocorrelation of the windowed speech is calculated by the formulae:

$$R_{rec}(n) = \sum_{k=0}^{k=79} s(k+30) \cdot s(k+30-n), n = 0,\ldots,30$$

$$R(n) = \sum_{k=0}^{k=34} s(k+110) \cdot s(k+110-n), n = 0,\ldots,30$$

3. Third, the output autocorrelation is computed and the memory is updated as follows:

$$\begin{cases} mem(k) = 0.31640625*mem(k) + R_{rec}(k) \\ pDstAutoCorr(k) = mem(k) + R(k) \end{cases}, k = 0,\ldots,30$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcSynthSpeech`*, *`pDstInvAutoCorr`* or *`pHybWinMem`* pointer is `NULL`.

RandomNoiseExcitation_G729B

Initializes a random vector with a Gaussian distribution.

```
IppStatus ippsRandomNoiseExcitation_G729B_16s (Ipp16s *pSeed, Ipp16s *pExc,
   int len);
```

Arguments

pSeed	Pointer to the input/output seed of random generator.
pExc	Pointer to the destination vector.
len	Length of the destination vector.

Discussion

The function ippsRandomNoiseExcitation_G729B_16s is declared in the ippsc.h file. This function generates random excitations as follows:

$$pEcx[n] = \frac{\sum_{i=1}^{12} \xi_{(12n+i)}}{128}, n = 0,..., len-1$$

where $\xi_{k+1} = 31821 \cdot \xi_k + 13849$, $\xi_0 = *pSeed$

The seed of the random generator ξ_k is updated.

Return Value

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

G.723.1 Related Functions

Intel IPP functions described in this section implement building blocks that can be used to create speech codecs compliant with the ITU-T Recommendation G.723.1 (see [ITU723], [ITU723A]).
The list of these functions is given in the following table:

Table 9-3 Intel IPP G.723.1 Related Functions

Function Base Name	Operation
Linear Prediction Analysis Functions	
AutoCorr_G723	Estimates the auto-correlation of a vector.
AutoCorr_NormE_G723	Estimates normal auto-correlation of a vector.

Table 9-3 Intel IPP G.723.1 Related Functions (continued)

Function Base Name	Operation
LevinsonDurbin_G723	Calculates the LP coefficients from the autocorrelation coefficients.
LPCToLSF_G723	Converts LP coefficients to LSF coefficients.
LSFToLPC_G723	Converts LSF coefficients to the LP coefficients.
LSFDecode_G723	Performs inverse quantization of LSFs.
LSFQuant_G723	Quantizes LSF coefficients.
Codebook Search Functions	
OpenLoopPitchSearch_G723	Searches for an optimal pitch value.
ACELPFixedCodebookSearch_G723	Searches the ACELP fixed codebook for the excitation.
AdaptiveCodebookSearch_G723	Searches for the close loop pitch and the adaptive gain index.
MPMLQFixedCodebookSearch_G723	Searches the MP-MLQ fixed codebook for the excitation.
ToeplizMatrix_G723	Calculates 416 elements of the Toepliz matrix for fixed codebook search.
Gain Quantization Functions	
GainQuant_G723	Implements MP-MLQ gain estimation and quantization.
GainControl_G723	Extracts delayed pitch contribution.
Filter Functions	
HighPassFilter_G723	Performs high-pass filtering of the input signal.
IIR16s_G723	Performs IIR filtering.
SynthesisFilter_G723	Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).
TiltCompensation_G723	Computes tilt compensation filter.
HarmonicSearch_G723	Searches for the harmonic delay and gain for the harmonic noise shaping filter.
HarmonicNoiseSubtract_G723	Performs harmonic noise shaping.
DecodeAdaptiveVector_G723	Restores the adaptive codebook vector from excitation, pitch, and adaptive gain.
PitchPostFilter_G723	Calculates coefficients of the pitch post filter.

Linear Prediction Analysis Functions

These functions perform LPC analysis, LSP coding (quantization) and decoding, as well as transformation between LPC and LSF coefficients.

AutoCorr_G723

Estimates the auto-correlation of a vector.

```
IppStatus ippsAutoCorr_G723_16s(const Ipp16s *pSrcSpch,  Ipp16s
   *pResultAutoCorrExp, Ipp16s *pDstAutoCorr);
```

Arguments

pSrcSpch	Pointer to the input speech signal vector [180].
pResultAutoCorrExp	Pointer to the exponent for autocorrelation coefficients.
pDstAutoCorr	Pointer to the autocorrelation coefficients vector [11].

Discussion

The function ippsAutoCorr_G723 is declared in the ippsc.h file. This function calculates the first 11 autocorrelation coefficients of the input speech signal. This function is applied to the 180 speech samples centered on the current subframe. The functionality is as follows.

1. First, apply to speech samples the Hamming windows, given by the following formula.

$$w(i) = 0.54 - 0.46\cos\left[\frac{(2i-1)}{399}\pi\right] \quad , i = 0,1,...,179$$

2. Next, calculate the autocorrelations as follows:

$$r(k) = \sum_{i=k}^{179} s(i) \times s(i-k) \quad , k = 0,1,...,10$$

3. Finally, add the autocorrelation with binomial window, given by:

$$b(0) = 1025/1024$$

$$b(i) = \exp\left[-0.5\left(\frac{2\pi f_0 i}{f_s}\right)^2\right], i = 1,...,10.$$

NOTE. *The function* `ippsAutoCorr_G723` *is actually a combination of* `ippsAutoScale`, `ippsMul_NR` *and* `ippsAutoCorr_NormE_G723` *functions. The following code details the correspondence.*

```
{
      int autoScale=3, corrScale;
      short vect[180];
      ippsAutoScale_16s(pSrcSpch, Vect,180, &autoScale );
      /* Apply the Hamming window */
      ippsMul_NR_16s_ISfs(HammWindow,Vect,180,15);
      /* Compute the autocorrelation coefficients */
      ippsAutoCorr_NormE_G723_16s(Vect,pDstAutoCorr,&corrScale);
      *pResultAutoCorrExp = corrScale+(autoScale<<1);
}
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcSpch`* or *`pDstAutoCorr`* pointer is `NULL`.

AutoCorr_NormE_G723

Estimates normal auto-correlation of a vector.

```
IppStatus ippsAutoCorr_NormE_G723_16s(const Ipp16s *pSrc, Ipp16s *pDst, int
   *pNorm);
```

Arguments

pSrc	Pointer to the source vector [180].
pDst	Pointer to the destination vector, which stores the estimated auto-correlation results of the source vector.
pNorm	Pointer to the output normalization scale factor.

Discussion

The function ippsAutoCorr_NormE_G723_16s is declared in the ippsc.h file. This function computes 11 auto-correlation coefficients from the input signal. The first correlation coefficient (energy) is multiplied by a white noise correction factor $b(0) = 1025/1024$, and the remaining 10 correlation coefficients are multiplied by the binomial window coefficients
b_n, $n = 1,...,10$ defined in the reference C-code (see [ITU723]).

The auto-correlation coefficients are additionally multiplied by the factor 2^{norm0}, where $norm0 \geq 0$ is calculated so as to make the first coefficient (energy) normalized. Thus, the resulting vector *pDst* is computed as follows:

$$pDst[n] = b_n \cdot 2^{norm0} \cdot \sum_{i=0}^{179-n} pSrc[i] \cdot pSrc[i+n], \; 0 \leq n \leq 10.$$

The function ippsAutoCorr_NormE stores the normalization scale factor *norm*0 in *pNorm*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst* or *pNorm* pointer is NULL.

LevinsonDurbin_G723

Calculates the LP coefficients from the autocorrelation coefficients.

```
IppStatus ippsLevinsonDurbin_G723_16s(const Ipp16s * pSrcAutoCorr, Ipp16s
   *pValResultSineDtct, Ipp16s *pResultResidualEnergy, Ipp16s *pDstLPC);
```

Arguments

pSrcAutoCorr	Pointer to the autocorrelation coefficients vector [11].
pValResultSineDtct	Pointer to the sine detector input/output parameter.
pResultResidualEnergy	Pointer to the output residual energy, in Q15.
pDstLPC	Pointer to the output LP coefficients vector [10], in Q13.

Discussion

The function ippsLevinsonDurbin_G723 is declared in the ippsc.h file. This function calculates the 10th order LP coefficients from the autocorrelation coefficients using Levinson-Durbin algorithm. This function also performs sine detection.

To obtain LP coefficients a_i, i = 1, 2, …, 10, the following set of equations is to be solved:

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k) \ , \ k = 1,2,...,10 \ .$$

The function ippsLevinsonDurbin_G723 performs the following steps:

1. Levinson-Durbin algorithm is applied to solve the above set of equations. This algorithm uses the recursion detailed in ippsLevinsonDurbin_G729 function.

2. If the LPC filter used in this algorithm is unstable, that is $|k_i|$ is very close to 1.0 during recursion, just set the remaining LP coefficients to zero.

3. The sine detector parameter is updated when i= 1 in the recursion, as follows:

$SineDtct = SineDtct << 1$

If $k > 0.95$, then $SineDtct = SineDtct + 1$.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcAutoCorr*, *pDstLPC*, *pValResultSineDtct*, or *pResultResidualEnergy* pointer is NULL.

LPCToLSF_G723

Converts LP coefficients to LSF coefficients.

```
IppStatus ippsLPCToLSF_G723_16s (const Ipp16s *pSrcLPC, const Ipp16s
   *pSrcPrevLSF, Ipp16s *pDstLSF);
```

Arguments

pSrcLPC	Pointer to the LPC input vector [10]: $a_1,\ldots,a_{10}$, in Q13.
pSrcPrevLSF	Pointer to the previous normalized LSF coefficients vector [10], in Q15.
pDstLSF	Pointer to the normalized LSF coefficients vector [10], in Q15.

Discussion

The function ippsLPCToLSF_G723 is declared in the ippsc.h file. This function converts a set of 10th order LP coefficients to LSF coefficients by implementing the following operations:

1. Apply the 7.5Hz bandwidth expansion.

2. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursion:

$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$

$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i)$

$i = 0...4$,

where $f_1(0) = f_2(0) = 1.0$

3. Use Chebyshev polynomials to find the roots of $F_1(z)$ and $F_2(z)$, and obtain the LSF coefficients:

$$c_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2$$

$$c_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2$$

3. If all 10 roots needed to determine LSF coefficients are not found, use the set of previous LSF coefficients.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcLPC`*, *`pSrcPrevLSF`* or *`pDstLSF`* pointer is `NULL`.

LSFToLPC_G723

Converts LSF coefficients to the LP coefficients.

```
IppStatus ippsLSFToLPC_G723_16s(const Ipp16s *pSrcLSF, Ipp16s *pDstLPC);
IppStatus ippsLSFToLPC_G723_16s_I (Ipp16s *pSrcLSFDstLPC);
```

Arguments

`pSrcLSF`	Pointer to the input LSF coefficients vector [10], in Q15.
`pDstLPC`	Pointer to the output LP coefficients vector [10], in Q13.
`pSrcLSFDstLPC`	Pointer to the input LSF and output LP coefficients vector [10], in Q15 and Q13, respectively.

Discussion

The function `ippsLSFToLPC_G723` is declared in the `ippsc.h` file. This function converts a set of 10th order LSF coefficients to the LP coefficients as follows:

1. Convert LSF coefficients to the LSP coefficients as:
$q_i = \cos(\omega_i)$, $i = 0...9$

2. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$ using the recursion:

$f_1(0) = f_1(-1) = 0$

for $i = 1$ to 5

$f_1(i) = -2q_{2i-1} * f_1(i-1) + 2f_1(i-2)$

$f_2(i) = -2q_{2i} * f_2(i-1) + 2f_2(i-2)$

for j = i-1 down to 1

$f_1^{[i]}(j) = f_1^{[i-1]}(j) - 2q_{2i-1} f_1^{[i-1]}(j-1) + f_1^{[i-1]}(j-2)$

$f_2^{[i]}(j) = f_2^{[i-1]}(j) - 2q_{2i} f_2^{[i-1]}(j-1) + f_2^{[i-1]}(j-2)$

where $f_1(0) = f_2(0) = 1.0$

3. Polynomials $F_1(z)$ and $F_2(z)$ are multiplied by $1 + z^{-1}$ and $1 - z^{-1}$, respectively, to obtain $F_1'(z)$ and $F_2'(z)$

4. The LP coefficients are calculated from the polynomials $F_1'(z)$ and $F_2'(z)$ using the following formulas:

$\alpha_{i=1..5} = 0.5 f_1'(i) + 0.5 f_2'(i)$

$\alpha_{i=6..10} = 0.5 f_1'(11-i) - 0.5 f_2'(11-i)$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcLSF*, *pDstLPC*, *pSrcLSFDstLPC* pointer is `NULL`.

LSFDecode_G723

Performs inverse quantization of LSFs.

```
IppStatus ippsLSFDecode_G723_16s (const Ipp16s *quantIndex, const Ipp16s
   *pPrevLSF, int erase, Ipp16s *pQuantLSF);
```

Arguments

quantIndex	Pointer to the input LSP VQ indices vector [3].
pPrevLSF	Pointer to the input quantized LSF for previous subframes.
erase	Indicates frame erasure.
pQuantLSF	Pointer to the output quantized LSF for previous subframes.

Discussion

The function ippsLSFDecode_G723_16s is declared in the ippsc.h file. This function performs inverse quantization of LSFs (in other words, decodes LSP VQ indices). First three VQ table entries corresponding to the transmitted index are found. The predicted vector is added to the found vector and DC vector to form the decoded vectors in three bands separately. A stability check is performed to ensure the difference between them not is less then 31.25 Hz.

For erased frame, zero predicted vectors are chosen and stability check for the difference in 62.5 Hz is applied.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *quantIndex*, *pPrevLSF* or *pQuantLSF* pointer is NULL.
ippStsLSFLow	Indicates a warning when a stability condition is not met.

LSFQuant_G723

Quantizes LSF coefficients.

```
IppStatus ippsLSFQuant_G723_16s32s(const Ipp16s* pSrcLSF, const Ipp16s*
   pSrcPrevLSF, Ipp32s* pResultQLSFIndex);
```

Arguments

pSrcLSF	Pointer to the LSF coefficients vector [10], in Q15.

pSrcPrevLSF	Pointer to the previous LSF coefficients vector [10], in Q15.
pResultQLSFIndex	Pointer to the combined index of quantized LSF coefficients. The combination is built by left-shifting the first codebook index by 16 bits and left-shifting the second codebook index by 8 bits, and then adding together the two shifted indices and the third codebook index.

Discussion

The function ippsLSFQuant_G723 is declared in the ippsc.h file. This function quantizes the LSF coefficients to obtain the codebook indices using PSVQ. It implements the following operations:

1. Calculate the diagonal weighting matrix *W*, determined from the unquantized LSF coefficients, as:

$$w_{1,1} = 1/(\omega_2 - \omega_1)$$

$$w_{10,10} = 1/(\omega_{10} - \omega_9)$$

$$w_{j,j} = 1/\min(\omega_j - \omega_{j-1}, \omega_{j+1} - \omega_j), j = 2...9$$

2. Calculate the prediction LSF coefficients as given by:

$$\omega_p = (\omega - \omega_{bc}) - 0.375(\omega_{-1} - \omega_{bc})$$

where ω are the current LSF coefficients, ω_{-1} are the previous LSF coefficients, and ω_{bc} are the DC LSF coefficients.

3. Search the codebook vector to minimize the following error:

$$E_l = (\omega_p - \omega_l)^T W (\omega_p - \omega_l),$$

where ω_l is the l-th code vector in the codebook. The quantization uses 3-3-4 split.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcLSF*, *pSrcPrevLSF*, *or pResultQLSFIndex* pointer is NULL.

Codebook Search Functions

These functions perform open loop pitch estimation using the adaptive codebook, and search in ACELP excitation (fixed) codebook for optimal signs and positions of the pulses.

OpenLoopPitchSearch_G723

Searches for an optimal pitch value.

```
IppStatus ippsOpenLoopPitchSearch_G723_16s(const Ipp16s * pSrcWgtSpch, Ipp16s
   * pResultOpenDelay);
```

Arguments

pSrcWgtSpch	Pointer to the perceptually weighted speech signal. The signal length is 265, and the pointer *pSrcWgtSpch* points to its 146th element, in Q12.
pResultOpenDelay	Pointer to the open-loop pitch search result.

Discussion

The function ippsOpenLoopPitchSearch_G723 is declared in the ippsc.h file. This function extracts the open loop pitch from the weighted speech signal. It is applied in half-frames as follows:

1. The open loop pitch is chosen such that the cross-correlation is maximized:

$$C_{ol}(j) = \frac{\left[\sum_{i=0}^{119} s(i)s(i-j)\right]^2}{\sum_{i=0}^{119} s(i-j)s(i-j)}, \quad j = 18,\ldots,142$$

2. During the search, preference is given to smaller pitch periods to avoid choosing pitch multiples. Every found pitch value is compared with the previous best. If the difference is less than 18 and $c_{ol}(j) > c_{ol}(j')$, the process is over. Otherwise, the pitch is chosen only if $c_{ol}(j)$ is greater than $c_{ol}(j')$ by 1.25db.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcWgtSpch* or *pResultOpenDelay* pointer is `NULL`.

ACELPFixedCodebookSearch_G723

Searches the ACELP fixed codebook for the excitation.

```
IppStatus ippsACELPFixedCodebookSearch_G723_16s(const Ipp16s *pSrcFixedCorr,
   const Ipp16s *pSrcMatrix, Ipp16s *pDstFixedSign, Ipp16s *pDstFixedPosition,
   Ipp16s *pResultGrid, Ipp16s *pDstFixedVector, Ipp16s *pSearchTimes);
IppStatus ippsACELPFixedCodebookSearch_G723_32s16s(const Ipp16s
   *pSrcFixedCorr, Ipp32s *pSrcDstMatrix, Ipp16s *pDstFixedSign, Ipp16s
   *pDstFixedPosition, Ipp16s *pResultGrid, Ipp16s *pDstFixedVector, Ipp16s
   *pSearchTimes);
```

Arguments

pSrcFixedCorr	Pointer to the correlation between residual and impulse response vector [60]
pSrcMatrix	Pointer to the elements of Toepliz matrix [416].
pSrcDstMatrix	Pointer to the input and output elements of Toepliz matrix [416].
pDstFixedSign	Pointer to the signs of the fixed vector [4].
pDstFixedPosition	Pointer to the positions of the fixed vector [4].
pResultGrid	Pointer to the beginning grid location.

pDstFixedVector	Pointer to the fixed vector [60].
pSearchTimes	Pointer to the input and output maximum search time

Discussion

The function ippsACELPFixedCodebookSearch_G723 is declared in the ippsc.h file. This function searches the ACELP fixed codebook for the excitation in the 5.3Kbps encoder. It is applied in subframes as follows:

1. There are four non-zero pulses in the fixed vector as given by:

$$c(n) = \sum_{k=0}^{3} \alpha_k \delta(n - m_k) \quad , \quad n = 0,...,59$$

where $\alpha_k, k = 0..3$ and $m_k, k = 0..3$ are the signs and positions of the fixed vector respectively.

2. Search for the parameters $\alpha_k, k = 0..3$ and $m_k, k = 0..3$, that minimize the following error:

$$\delta = \Phi(m_0, m_0) + \Phi(m_1, m_1) + 2\alpha_0\alpha_1\Phi(m_0, m_1) +$$
$$\Phi(m_2, m_2) + 2[\alpha_0\alpha_2\Phi(m_0, m_2) + \alpha_1\alpha_2\Phi(m_1, m_2)] +$$
$$\Phi(m_3, m_3) + 2[\alpha_0\alpha_3\Phi(m_0, m_3) + \alpha_1\alpha_3\Phi(m_1, m_3) + \alpha_2\alpha_3\Phi(m_2, m_3)]$$

where $\Phi(i, j)$ is the Toepliz matrix.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcFixedCorr*, *pSrcMatrix*, *pSrcDstMatrix*, *pSearchTimes*, *pDstFixedSign*, *pDstFixedPosition*, *pResultGrid*, or *pDstFixedVector* pointer is NULL.

AdaptiveCodebookSearch_G723

Searches for the close loop pitch and the adaptive gain index.

```
IppStatus ippsAdaptiveCodebookSearch_G723(Ipp16s valBaseDelay,
   const Ipp16s * pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse,
   Ipp16s * pSrcPrevExcitation, const Ipp32s *pSrcPrevError,
   Ipp16s * pResultCloseLag, Ipp16s * pResultAdptGainIndex, Ipp16s subFrame,
   Ipp16s sineDtct, IppSpchBitRate bitRate);
```

Arguments

valBaseDelay	Base delay, in the range [18,145].
pSrcAdptTarget	Pointer to the adaptive target signal vector [60].
pSrcImpulseResponse	Pointer to the impulse response vector [60].
pSrcPrevExcitation	Pointer to the previous excitation vector [145].
pSrcPrevError	Pointer to the previous error vector [5].
subFrame	Subframe number, from 0 to 3.
bitRate	Transmit bit rate, equal to either `IPP_SPCHBR_6300` or `IPP_SPCHBR_5300`.
sineDtct	Sine detector parameter.
pResultCloseLag	Pointer to the lag of close pitch.
pResultAdptGainIndex	Pointer to the index of the adaptive gain.

Discussion

The function `ippsAdaptiveCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches for the close loop pitch and the adaptive gain index. It is applied in subframes as follows:

1. For subframe 0 and 2, the close loop pitch lag is chosen in the range of [-1,1] around the appropriate open loop pitch. For subframe 1 and 3, the range is [-1, 3] around the open loop pitch.

2. Denote the close loop pitch as L_i , i = 0...3. The pitch predictor gains are vector quantized. For 6.3Kbps bit rate, two codebooks are used with 85 entries and 170 entries, respectively. If L_0 is less than 58 for subframe 0 and 1, or if L_2 is less than 58 for subframe 2 and 3, then the 85-entry codebook is used for pitch gain quantization. Otherwise, the 170-entry codebook is used. For 5.3Kbps bit rate, the quantization is against a 170-entry codebook.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcAdptTarget`*, *`pSrcImpulseResponse`*, *`pSrcPrevExcitation`*, *`pSrcPrevError`*, *`pResultCloseLag`*, or *`pResultAdptGainIndex`* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *`valBaseDelay`* is not in the range [18, 145], or *`subFrame`* is not in the range [0, 3], or *`bitRate`* is not a valid element of the enumerated type `IppSpchBitRate`.

MPMLQFixedCodebookSearch_G723

Searches the MP-MLQ fixed codebook for the excitation.

```
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);
```

Arguments

`valBaseDelay`	Base delay, in the range [18,145].
`pSrcImpulseResponse`	Pointer to the impulse response vector [60].

pSrcResidualTarget	Pointer to the residual target signal vector [60].
pDstFixedVector	Pointer to the fixed codebook vector [60].
pResultGrid	Pointer to the beginning grid location, 0 or 1.
pResultTrainDirac	Pointer to the flag to show if train Dirac function is used: 0-unused, 1-used.
pResultAmpIndex	Pointer to the index of the quantized amplitude.
pResultAmplitude	Pointer to the amplitude of the fixed codebook vector.
pResultPosition	Pointer to the position of the fixed codebook vector, whose amplitude has non-zero value.
subFrame	Subframe number, from 0 to 3.

Discussion

The function ippsMPMLQFixedCodebookSearch_G723 is declared in the ippsc.h file. This function searches the MP-MLQ fixed codebook for the excitation in the 6.3Kbps encoder. It is applied in subframes as follows:
1. The error is obtained by the following calculation:

$$err(n) = res(n) - G \sum_{k=0}^{M-1} \alpha_k h(n - m_k)$$

where G is the gain factor, α_k and m_k are the signs and positions of the fixed vector, respectively.

2. Search for the parameters G, α_k and m_k that minimize the mean square of the error signal $err(n)$.

3. The fixed codebook gain is obtained using the following formulae:

$$d(j) = \sum_{n=j}^{59} c(n) \cdot h(n - j)$$

$$Gm = \frac{\max_{j=0..59} \{|d(j)|\}}{\sum_{n=0}^{59} h(n) \cdot h(n)},$$

where $c(n)$ is the fixed vector.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcImpulseResponse*, *pSrcResidualTarget*, *pDstFixedVector*, *pResultGrid*, *pResultTrainDirac*, *pResultAmpIndex*, *pResultAmplitude*, *or pResultPosition* pointer is NULL.
ippStsRangeErr	Indicates an error when *subFrame* is not one of 0, 1, 2, or 3; or when *valBaseDelay* is not in the range [18,145].

ToeplizMatrix_G723

Calculates 416 elements of the Toepliz matrix for fixed codebook search.

```
IppStatus ippsToeplizMatrix_G723_16s(const Ipp16s * pSrcImpulseResponse,
   Ipp16s * pDstMatrix);
IppStatus ippsToeplizMatrix_G723_16s32s(const Ipp16s * pSrcImpulseResponse,
   Ipp32s * pDstMatrix);
```

Arguments

pSrcImpulseResponse	Pointer to the impulse response vector [60].
pDstMatrix	Pointer to the elements of Toepliz matrix vector [416].

Discussion

The function ippsToeplizMatrix_G723 is declared in the ippsc.h file. This function calculates the 416 elements in Toepliz matrix for the fixed-codebook search. Elements of the Toepliz matrix can be expressed as follows:

$$\Phi(i, j) = \sum_{n=j}^{59} h(n-i) \times h(n-j) \text{ , } i \le j \text{ , } 0 \le i \le 59 \text{ ,}$$

where $h(i)$, i = 0, 1, …, 59, is the impulse response.

The function stores the calculated 416 elements in *pDstMatrix* in the following order:

1. $\Phi(m_i, m_i)$, (i = 0, 1, 2, 3), 4x8 =32 elements; starting location: 0.
2. $\Phi(m_0, m_1)$, 8x8 = 64 elements; starting location: 32.
3. $\Phi(m_0, m_2)$, 8x8 = 64 elements; starting location: 96.
4. $\Phi(m_0, m_3)$, 8x8 = 64 elements; starting location: 160.
5. $\Phi(m_1, m_2)$, 8x8 = 64 elements; starting location: 224.
6. $\Phi(m_1, m_3)$, 8x8 = 64 elements; starting location: 288.
7. $\Phi(m_2, m_3)$, 8x8 = 64 elements; starting location: 352.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcImpulseResponse* or *pDstMatrix* pointer is `NULL`.

Gain Quantization

These functions perform fixed codebook gain estimation and gain adjustment of the postfiltered signal.

GainQuant_G723

Implements MP-MLQ gain estimation and quantization.

```
IppStatus ippsGainQuant_G723_16s (const Ipp16s *pImp, const Ipp16s *pSrc,
   Ipp16s *pDstLoc, Ipp16s *pDstAmp, Ipp32s *pMaxErr, Ipp16s *pGrid, Ipp16s
   *pAmp, int Np, int* isBest);
```

Arguments

pImp	Pointer to the input impulse response *h* vector [60].
pSrc	Pointer to the input target signal *r* vector [60].
pDstLoc	Pointer to the output pulse locations.
pDstAmp	Pointer to the output pulse amplitudes.
pMaxErr	Pointer to the input/output quantization error.
pGrid	Pointer to the output grid: 0 - if pulses are in even positions, 1 - if pulses are in odd positions.
pAmp	Pointer to the output index of maximum amplitude.
Np	Number of pulses: equal to 6 for even subframe, equal to 5 for odd subframe.
isBest	Indicates quantization result. Set to 0, if no pulses are found with quantization error better (less) than the input error.

Discussion

The function ippsGainQuant_G723_16s is declared in the ippsc.h file. This function estimates the unknown parameters, G, $\{\alpha_k\}$, $\{m_k\}$, $k = 0,\ldots,Np-1$, that minimize the mean square of the error signal:

$$err[n] = r[n] - G \cdot \sum_{k=0}^{Np-1} a_k \cdot h[n - m_k]$$

The estimation is done as follows.

First, cross correlation of the impulse response and the target vector is computed:

$$d[j] = \sum_{n=j}^{59} r[n] \cdot h[n-j], \; 0 \le n \le 59$$

The estimated maximum gain is computed as:

$$G_{max} = \frac{\max\{|d[j]|\}_{j=0..59}}{\sum_{n=0}^{59} h[n] \cdot h[n]}$$

and then is quantized by the logarithmic quantizer. The gain values selected around this quantized gain are then used to optimize the signs and locations of the pulses for both even and odd grids that yield the minimum mean square of the error signal.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pImp`*, *`pSrc`*, *`pDstLoc`*, *`pDstAmp`*, *`pMaxErr`*, *`pGrid`*, *`pAmp`*, or *`isBest`* pointer is `NULL`.

GainControl_G723

Extracts delayed pitch contribution.

```
IppStatus ippsGainControl_G723_16s_I (Ipp32s energy, Ipp16s *pSrcDst, Ipp16s
   *pGain);
```

Arguments

`energy`	The input energy coefficient.
`pSrcDst`	Pointer to the input/output post-filtered signal.
`pGain`	Pointer to the input/output gain.

Discussion

The function `ippsGainControl_G723_16s_I` is declared in the `ippsc.h` file. This function first computes the amplitude ratio g_s as:

$$g_s = \sqrt{\frac{energy}{\sum_{n=0}^{59} pSrcDst[n] \cdot pSrcDst[n]}}$$

If denominator in this expression is equal to 0, g_s is set to 1.

Then the input postfiltered signal is scaled as follows:

$pSrcDst[n] = pSrcDst[n] \cdot g_n \cdot (1+\alpha)$, $n = 0,..., 59,$

where g_n is updated using the following expression, respectively:

$g_n = (1-\alpha) \cdot g_{n-1} + \alpha \cdot g_s$, $n = 0,..., 59,$ $g_{-1} = 0$

and α is equal to 1/16.

The output gain is equal to g_{59} .

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcDst`* or *`pGain`* pointer is `NULL`.

Filter Functions

These functions implement different types of filtering, including high pass filtering on pre-processing stage of encoding and post-processing stage of decoding, as well as the postfilter in final decode stage.
Different kinds of synthesis (IIR), harmonic, and preemphasize filter functions can be combined to perform more complex filtering and may be used in different encode and decode stages.

HighPassFilter_G723

Performs high-pass filtering of the input signal.

```
IppStatus ippsHighPassFilter_G723_16s (const Ipp16s* pSrc, Ipp16s* pDst, int*
   pMem);
```

Arguments

pSrc	Pointer to the source vector [240].
pDst	Pointer to the destination vector [240].
pMem	Pointer to the filter memory vector [2]. This vector must be initially filled with zeroes.

Discussion

The function `ippsHighPassFilter_G723` is declared in the `ippsc.h` file. This function pre-processes the input signal using the following filter transfer function:

$$H_h(z) = \frac{1 - z^{-1}}{1 - \frac{127}{128} z^{-1}}$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst* or *pMem* pointer is `NULL`.

IIR16s_G723

Performs IIR filtering.

```
IppStatus ippsIIR16s_G723_16s32s (const Ipp16s *pCoeffs, const Ipp16s *pSrc,
   Ipp32s *pDst, Ipp16s *pMem);
IppStatus ippsIIR16s_G723_16s_I (const Ipp16s *pCoeffs, Ipp16s *pSrcDst,
   Ipp16s *pMem);
IppStatus ippsIIR16s_G723_32s16s_Sfs (const Ipp16s *pCoeffs, const Ipp32s
   *pSrc, int scaleFactor, Ipp16s *pDst, Ipp16s *pMem);
```

Arguments

pCoeffs	Pointer to the input vector of filter coefficients vector [20]: $b_1,\ldots, b_{10}, a_1, \ldots,a_{10}$.
pSrc	Pointer to the input signal vector [60].
pSrcDst	Pointer to the input/output signal vector [60].
scaleFactor	Scale factor.
pDst	Pointer to the output filtered vector [60].
pMem	Pointer to the filter memory vector [20]. This vector must be initially filled with zeroes.

Discussion

These functions are declared in the `ippsc.h` file.

The functions `ippsIIR16s_G723_16s32s` and `ippsIIR16s_G723_16s_I` perform infinite impulse response (IIR) filtering using the following transfer function:

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The function `ippsIIR16s_G723_32s16s_Sfs` performs IIR filtering using the transfer function:

$$H(z) = 2^{sFs} \cdot \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The filter memory is updated.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pCoeffs`*, *`pSrc`*, *`pSrcDst`*, *`pDst`*, or *`pMem`* pointer is `NULL`.

SynthesisFilter_G723

Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).

```
IppStatus ippsSynthesisFilter_G723_16s32s (const Ipp16s* pLPC, const Ipp16s*
   pSrc, Ipp32s* pDst, Ipp16s* pMem);
IppStatus ippsSynthesisFilter_G723_16s (const Ipp16s *pLPC, const Ipp16s *
   pSrc, Ipp16s * pMem, Ipp16s *pDst);
```

Arguments

`pLPC`	Pointer to the input LP coefficients a_0, a_1,…,a_{10}, in Q11.
`pSrc`	Pointer to the source vector.
`pDst`	Pointer to the filtered output.
`pMem`	Pointer to the memory supplied for filtering: short integer vector [10], initially set to zero.

Discussion

The function ippsSynthesisFilter_G723 is declared in the ippsc.h file. This function computes the filter given by:

$$H(z) = \hat{A}^{-1}(z) = \frac{1}{a_0 + \sum_{i=1}^{10} a_i \cdot \hat{z}^{-i}}$$

This function is applied after the residual filter in computing the perceptually weighted speech signal:

$$pDst[n] = pSrc[n] \cdot pLPC[0] + \sum_{i=1}^{10} pLPC[i] \cdot pMem[n-i+1] \quad , n = 0,...,59$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pLPC*, *pDst*, or *pMem* pointer is NULL.
ippStsOverflow	Indicates a warning that at least one result value was saturated.

TiltCompensation_G723

Computes tilt compensation filter.

```
IppStatus ippsTiltCompensation_G723_32s16s (Ipp16s val, const Ipp32s* pSrc,
   Ipp16s *pDst);
```

Arguments

val	The first-order partial correlation coefficient, in Q15.
pSrc	Pointer to the source vector [61].
pDst	Pointer to the output filtered vector [60].

Discussion

The function ippsTiltCompensation_G723 is declared in the ippsc.h file. This function computes the tilt compensation filter as:

$H_t(z) = (1 - val \cdot z^{-1})$

$pDst[i] = pSrc[i+1] + pSrc[i] \cdot val\ ,\ i = 0,...,59$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.

HarmonicSearch_G723

Searches for the harmonic delay and gain for the harmonic noise shaping filter.

```
IppStatus ippsHarmonicSearch_G723_16s(Ipp16s valOpenDelay, const Ipp16s
   *pSrcWgtSpch, Ipp16s *pResultHarmonicDelay, Ipp16s *pResultHarmonicGain);
```

Arguments

valOpenDelay	Open loop pitch, in the range [18,145].
pSrcWgtSpch	Pointer to the weighted speech vector [205]. The pointer points to the 146th element, in Q12.
pResultHarmonicDelay	Pointer to the output harmonic delay.
pResultHarmonicGain	Pointer to the output harmonic gain, in Q15.

Discussion

The function ippsHarmonicSearch_G723 is declared in the ippsc.h file. This function searches for the harmonic delay and harmonic gain for the harmonic noise shaping filter, using the weighted speech and open loop pitch as input.
This function is applied in subframes as follows:

1. Find the index L to optimize the following expression:

$$C_{pw}(j) = \left[\sum_{i=0}^{59} s(i)s(i-j) \right]^2 / \left(\sum_{i=0}^{59} s(i-j)s(i-j) \right), \quad j = pitch-3..pitch+3$$

2. Calculate the optimal filter gain as follows:

$$G_{opt} = \sum_{i=0}^{59} s(i)s(i-j) / \sum_{i=0}^{59} s(i-j)s(i-j)$$

G_{opt} is saturated to [0,1].

3. Calculate the energy of the weighted speech samples and the harmonic gain:

$$E = \sum_{i=0}^{59} s^2(i)$$

$$\beta = \begin{cases} 0.3125 G_{opt}, & if \ -10\log 10(1 - C_L / E) \geq 2.0 \\ 0, & otherwise \end{cases}$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcWgtSpch`*, *`pResultHarmonicDelay`*, or *`pResultHarmonicGain`* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *`valOpenDelay`* is not in the range [18, 145].

HarmonicNoiseSubtract_G723

Performs harmonic noise shaping.

```
IppStatus ippsHarmonicNoiseSubtract_G723_16s_I(Ipp16s val, int T, const Ipp16s
   *pSrc, Ipp16s *pSrcDst);
```

Arguments

val	The input harmonic filter coefficient, in Q15.
T	The input harmonic filter lag.
pSrc	Pointer to the input zero impulse response vector [60] of the combined filter.
pSrcDst	Pointer to the input/output harmonic noise weighted speech vector [60].

Discussion

The function ippsHarmonicNoiseSubtract_G723_16s_I is declared in the ippsc.h file. This function subtracts the harmonic shaped vector *pSrc* from vector *pSrcDst*, as follows:

$$pSrcDst[n] = pSrcDst[n] - (pSrc[n] + val \cdot pSrc[n - T]) \text{ , } n = 0,..., 59$$

This operation is used for ringing subtraction, which is performed by subtracting the zero impulse response from the harmonic weighted speech vector to obtain the target vector:

$$t[n] = w[n] - z[n]$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pSrcDst* pointer is NULL.

DecodeAdaptiveVector_G723

Restores the adaptive codebook vector from excitation, pitch, and adaptive gain.

```
IppStatus ippsDecodeAdaptiveVector_G723_16s(Ipp16s valBaseDelay, Ipp16s
   valCloseLag, Ipp16s valAdptGainIndex, const Ipp16s *pSrcPrevExcitation,
   Ipp16s *pDstAdptVector, IppSpchBitRate bitRate);
```

Arguments

valBaseDelay	Base delay, in the range [18,145].
valCloseLag	Lag of close pitch.
valAdptGainIndex	Index of adaptive gain.
pSrcPrevExcitation	Pointer to the past excitations vector [145].
bitRate	Transmit bit rate, equal to either `IPP_SPCHBR_6300` or `IPP_SPCHBR_5300`.
pDstAdptVector	Pointer to the adaptive codebook vector [60].

Discussion

The function `ippsDecodeAdaptiveVector_G723` is declared in the `ippsc.h` file. This function decodes the adaptive vector from excitation, close loop pitch, and adaptive gain index. It is applied in subframes as follows:

1. Obtain the pitch as:

$$L = L_o + L_c - 1 ,$$

where L_o is the open loop pitch and L_c is the close loop pitch lag.

2. Obtain the residual from the previous excitation using the following relations:

res(0) = *exci*(143-*L*);

res(1) = *exci*(143-*L*+1);

When decoding a second subframe, the sum (L_o+L_c) can be equal to 146, so elements *exci*(-2) and *exci*(-1) must be available.

$res(i+2) = exci(145 - L + (i\%L))$, $i = 0...61$

3. Select the adaptive codebook, G_b, from the 85-or 170-entry codebooks.

4. Obtain the adaptive vector using the formula:

$$c(i) = \sum_{j=0}^{d} res(i+j)G_b(20k+i) \text{ , } i = 0...59$$

where k is the adaptive gain index.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcPrevExcitation`* or *`pDstAdptVector`* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *`valBaseDelay`* is not in the range [18, 145]; or *`valCloseLag`* is not in the range [0, 2]; or *`valAdptGainIndex`* is not in the range [0, 169], or *`bitRate`* is not a valid element of the enumerated type `IppSpchBitRate`.

PitchPostFilter_G723

Calculates coefficients of the pitch post filter.

```
IppStatus ippsPitchPostFilter_G723_16s(Ipp16s valBaseDelay, const Ipp16s
   *pSrcResidual, Ipp16s *pResultDelay, Ipp16s *pResultPitchGain, Ipp16s
   *pResultScalingGain, Ipp16s subFrame, IppSpchBitRate bitRate);
```

Arguments

`valBaseDelay`	Base delay, in the range [18,145].
`pSrcResidual`	Pointer to the residual signal vector [365]. This pointer points to the 146th element.
`pResultDelay`	Pointer to the delay of the pitch post filter.

pResultPitchGain	Pointer to the gain of the pitch post filter, in Q15.
pResultScalingGain	Pointer to the scaling gain of the pitch post filter, in Q15.
subFrame	Subframe number, from 0 to 3.
bitRate	Transmit bit rate, equal to either IPP_SPCHBR_6300 or IPP_SPCHBR_5300.

Discussion

The function ippsPitchPostFilter_G723 is declared in the ippsc.h file. This function calculates the coefficients of the pitch post filter. It is applied in subframes as follows:

1. Search for the forward pitch lag M_f from the forward cross-correlation:

$$C_f = \sum_{i=0}^{59} res(i)res(i+M_f) \text{ , } M_1 \le M_f \le M_2$$

where $M_1 = valBaseDelay - 3$ and $M_2 = valBaseDelay + 3$.

2. Search for the backward pitch lag M_b from the backward cross-correlation:

$$C_b = \sum_{i=0}^{59} res(i)res(i-M_b) \text{ , } M_1 \le M_b \le M_2$$

3. If C_f or C_b is negative, then the corresponding weight and delay are set to zero.

4. Calculate the relevant residual energy as follows:

$$D_f = \sum_{i=0}^{59} res(i+M_f)res(i+M_f)$$

$$D_b = \sum_{i=0}^{59} res(i-M_b)res(i-M_b)$$

$$T_{en} = \sum_{i=0}^{59} res(i)res(i)$$

5. Obtain the optimal gain: $g = C / D$

6. Calculate the scaling gain using the following formula:

$$g_s = \sqrt{\sum_{i=0}^{59} res^2(i) / \sum_{i=0}^{59} res'^2(i)}$$

where $res'(i)$ is the post filtered residual.

7. Calculate the total gain of the post filter:

$g_p = \gamma_{ltp} \cdot g_p \cdot g$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrcResidual*, *pResultDelay*, *pResultPitchGain*, *or pResultScalingGain* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *valBaseDelay* is not in the range [18, 145], or *subFrame* is not 0 or 3, or *bitRate* is not a valid element of the enumerated type `IppSpchBitRate`.

GSM-AMR Related Functions

This part of the chapter describes Intel IPP functions that can be combined to construct a bit-exact implementation of the European Telecommunications Standards Institute (ETSI) Global System for Mobile Communications (GSM) Adaptive Multi-Rate (AMR) ETSI EN 301 704 GSM 06.90 version 7.5.0 Release 2001 speech codec, more commonly known as the "GSM-AMR 06.90" codec. The primitives are primarily concerned with the well-defined, computationally expensive core operations that comprise the codec portion of the GSM-AMR system.

The GSM 06.90 AMR speech codec comprises an adaptive multi-rate algorithm that represents efficiently telephone-bandwidth digital speech using compressed data rates of 4.75, 5.15, 5.90, 6.70, 7.40, 7.95, 10.2, and 12.2 kilobits per second (kbps). The GSM-AMR system adaptively controls speech codec bit rates such that output quality is maximized for a given set of channel conditions.

Table 9-4 shows the functional grouping of the GSM-AMR primitives.

Table 9-4 Summary and Functional Groupings of the GSM-AMR Codec Primitives

Function Subset	Function Name
Basic functions	Interpolate_GSMAMR
	FFTFwd_RToPerm_GSMAMR
LP analysis	AutoCorr_GSMAMR
	LevinsonDurbin_GSMAMR
	LPCToLSP_GSMAMR
	LSPToLPC_GSMAMR
	LSFToLSP_GSMAMR
	LSPQuant_GSMAMR
	QuantLSPDecode_GSMAMR

Table 9-4 Summary and Functional Groupings of the GSM-AMR Codec Primitives

Adaptive-codebook search	OpenLoopPitchSearchNonDTX_GSMAMR OpenLoopPitchSearchDTXVAD1_GSMAMR OpenLoopPitchSearchDTXVAD2_GSMAMR ImpulseResponseTarget_GSMAMR AdaptiveCodebookSearch_GSMAMR AdaptiveCodebookDecode_GSMAMR AdaptiveCodebookGain_GSMAMR
Fixed-codebook search	AlgebraicCodebookSearch_GSMAMR FixedCodebookDecode_GSMAMR
Discontinuous transmission	Preemphasize_GSMAMR VAD1_GSMAMR VAD2_GSMAMR EncDTXSID_GSMAMR EncDTXHandler_GSMAMR EncDTXBuffer_GSMAMR, DecDTXBuffer_GSMAMR
Post-processing	PostFilter_GSMAMR

Sections that follow provide details on the primitives that implement each of the GSM-AMR functional blocks summarized in Table 9-4, specifically:

Section Basic Functions describes some general primitives.

Section LP Analysis and Quantization 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.

Section Discontinuous Transmission (DTX) is concerned with primitives for discontinuous transmission (DTX), including functions that implement voice activity detection (VAD) algorithms 1 and 2, as well as primitives for comfort noise generation (CNG), and DTX buffer management.

Section Post Processing describes post-processing primitives.

Basic Functions

Interpolate_GSMAMR

Computes the weighted sum of two vectors.

```
IppStatus ippsInterpolate_GSMAMR_16s (const Ipp16s *pSrc1, const Ipp16s
   *pSrc2, Ipp16s *pDst, int len);
```

Arguments

pSrc1	Pointer to the first input vector[*len*].
pSrc2	Pointer to the second input vector[*len*].
pDst	Pointer to the output vector[*len*].
len	Length of the input and output vectors.

Discussion

The function `ippsInterpolate_GSMAMR` is declared in `ippsc.h` file. This function computes the weighted sum of two vectors as:

pDst[i] = (*pSrc1*[i]>>2) + (*pSrc2*[i] - (*pSrc2*[i]>>2)); $0 \le i < len$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.

FFTFwd_RToPerm_GSMAMR

Computes the forward fast Fourier transform (FFT) of a real signal.

```
IppStatus ippsFFTFwd_RToPerm_GSMAMR_16s_I (Ipp16s *pSrcDst);
```

Arguments

pSrcDst	Pointer to the input and output vector[128].

Discussion

The function `ippsFFTFwd_RToPerm_GSMAMR` is declared in `ippsc.h` file. This function computes the forward FFT of a real signal. This transform is the same as done by the general signal processing FFT function `ippsFFTFwd_RToPerm_16s_Sfs` with the following settings: `order` = 7, `flag` = `IPP_FFT_DIV_FWD_BY_N`, `scale factor` = –1. The result of function operation may be different because the rounding mode used in `ippsFFTFwd_RToPerm_16s_Sfs` does not guarantee a bit-exact operation of GSM AMR codec (see Fourier Transform Functions in chapter 7).

Return Value

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

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

Estimates the autocorrelation sequence for a block of samples.

```
IppStatus ippsAutoCorr_GSMAMR_16s32s(const Ipp16s * pSrcSpch, Ipp32s *
   pDstAutoCorr, IppSpchBitRate mode);
```

Arguments

pSrcSpch	Pointer to the input speech vector (240 samples), represented using Q15.0. This should be aligned on an 8-byte boundary.
pDstAutoCorr	Pointer to the autocorrelation coefficients, of length 22. For 12.2 kbps mode, elements 0 ~ 10 contain the first set of autocorrelation lags, and elements 11 ~ 21 contain the second set of autocorrelation lags. For all other modes there is only one set of autocorrelation lags contained in vector elements 0 ~ 10.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Discussion

The function `ippsAutoCorr_GSMAMR` is declared in `ippsc.h` file. This function estimates the autocorrelation sequence for a block of 240 samples (30 ms). For the12.2 kbps mode, the 160 samples associated with the current 20 ms frame are combined with the last 80 samples from the previous frame, and two autocorrelation sequences are estimated. For all other bit rates, 160 samples from the current frame are combined with the last 40 samples from the previous frame as well as the first 40 samples from the next frame, and only one autocorrelation sequence is estimated. In particular, the estimates are formed as follows:

1. Tapered windowing – asymmetric tapered windows are applied to the input speech. Different windows are selected depending on the bit rate. For 12.2 kbps frames, unique tapered windows are applied for each of the two autocorrelation estimates, that is,

$$w_1(n) = \begin{cases} 0.54 - 0.46\cos(\frac{n\pi}{159}), n = 0,1,...,159 \\ 0.54 + 0.46\cos(\frac{(n-160)\pi}{79}), n = 160,161,...,239 \end{cases}$$

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 w_1 nor w_2 has any look ahead. For all bit rates other than 12.2 kbps, a window centered on the current frame is applied, i.e.,

$$w_3(n) = \begin{cases} 0.54 - 0.46\cos(\frac{2n\pi}{399}), n = 0,1,...,199 \\ \cos(\frac{2(n-200)\pi}{159}), n = 200,201,...,239 \end{cases}$$

2. Estimation of autocorrelation lags – autocorrelation lags are estimated from the windowed speech samples $s(i)$, $i = 0, 1, \ldots, 239$, as follows

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k), \quad k = 0, 1, \ldots, 10$$

3. Bandwidth expansion – the following binomial lag window is applied to the autocorrelation sequence(s) obtained in step 2:

$$bi(i) = \exp[-0.5 \times (\frac{2\pi f_0 i}{f_s})^2], i = 0,1,...,10$$

where f_0 = 60 Hz and f_s = 8000 Hz.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when the *`pSrcSpch`* or *`pDstAutoCorr`* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *`mode`* is not a valid element of the enumerated type `IppSpchBitRate`.

LevinsonDurbin_GSMAMR

Calculates the LP coefficients using the Levinson-Durbin algorithm.

```
IppStatus ippsLevinsonDurbin_GSMAMR(const Ipp32s * pSrcAutoCorr,
   Ipp16s * pSrcDstLpc);
IppStatus ippsLevinsonDurbin_GSMAMR_32s16s (const Ipp32s * pSrcAutoCorr,
   Ipp16s * pSrcDstLpc);
```

Arguments

pSrcAutoCorr	Pointer to the autocorrelation coefficients, a vector of length 11.
pSrcDstLpc	Pointer to the LP coefficients associated with the previous frame, a vector of length 11, represented using Q3.12. On output, updated to point to the LP coefficients associated with the current frame, a vector of length 11, represented using Q3.12.

Discussion

The function `ippsLevinsonDurbin_GSMAMR` is declared in `ippsc.h` file. This function calculates the 10th-order LP coefficients from the autocorrelation lags using the Levinson-Durbin algorithm. The detailed steps performed by the primitive are as follows:

1. Minimization of the prediction residual in the mean-square sense yields a system of linear equations that can be solved efficiently using the Levinson-Durbin algorithm to yield a set of LP coefficients, a_i, i = 1, 2, …, 10, that is

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k) \ , k = 1,2,...,10 \ .$$

2. The Levinson-Durbin algorithm performs the following recursion to obtain the prediction coefficients:

$$E^{[0]} = r(o)$$

for $i = 1$ to 10

$$a_0^{[i-1]} = 1$$

$$k_i = -\left[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j)\right] / \ E^{[i-1]}$$

$$a_i^{[i]} = k_i$$

for $j = 1$ to i-1

$$a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$$

end

$$E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$$

end

3. Unstable synthesis filter exception handling: if autocorrelation analysis yields an unstable LPC synthesis filter, i.e., if $|k_i|$ is close to or larger than 1.0, then the LP coefficients associated with the previous frame should replace the LP coefficients estimated on the current frame.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when the *`pSrcAutoCorr`* or *`pSrcDstLpc`* pointer is `NULL`.

LPCToLSP_GSMAMR

Converts LP coefficients to line spectrum pairs.

```
IppStatus ippsLPCToLSP_GSMAMR_16s(const Ipp16s * pSrcLpc, const Ipp16s *
   pSrcPrevLsp, Ipp16s * pDstLsp);
```

Arguments

`pSrcLpc`	Pointer to the eleven-element LP coefficient vector, represented using Q3.12.
`pSrcPrevLsp`	Pointer to the ten-element LSP coefficient vector associated with the previous frame, represented using Q0.15.
`pDstLsp`	Pointer to the ten-element LSP coefficient vector, represented using Q0.15.

Discussion

The function `ippsLPCToLSP_GSMAMR` is declared in `ippsc.h` file. This function converts a set of 10th-order LP coefficients to an equivalent set of line spectrum pairs (LSPs). The functionality is as follows:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$ using the recursive relations

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i),\ i = 0,1,...,4$$

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.

Return Value

`ippStsNoErr`	Indicates no error.

ippStsBadArgErr	Indicates an error when the *pSrcLpc*, *pSrcPrevLsp* or *pDstLsp* pointer is NULL.

LSPToLPC_GSMAMR

Converts LSPs to LP coefficients.

```
IppStatus ippsLSPToLPC_GSMAMR_16s(const Ipp16s * pSrcLsp, Ipp16s * pDstLpc);
```

Arguments

pSrcLsp	Pointer to the ten-element LSP coefficient vector, represented using Q0.15.
pDstLpc	Pointer to the eleven-element LP coefficient vector, represented using Q3.12.

Discussion

The function ippsLSPToLPC_GSMAMR is declared in ippsc.h file. This function converts a set of 10th-order LSPs to LP coefficients. The functionality is as follows:

1. Calculate the polynomial coefficients of $F_1(z)$ and $F_2(z)$, using the recursive relations

```
for i = 1 to 5
```

$$f_1(i) = -2q_{2i-1} \times f_1(i-1) + 2f_1(i-2)$$

```
for j = i-1 down to 1
```

$$f_1(j) = f_1(j) - 2q_{2i-1} \times f_1(j-1) + f_1(j-2)$$

```
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

$$a_i = \begin{cases} 0.5\times f_1'(i) + 0.5\times f_2'(i), i = 1,2,...,5 \\ 0.5\times f_1'(11-i) - 0.5\times f_2'(11-i), i = 6,7,...,10 \end{cases}$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when the *`pSrcLsp`* or *`pDstLpc`* pointer is `NULL`.

LSFToLSP_GSMAMR

Converts Line Spectral Frequencies to LSP coefficients.

```
IppStatus ippsLSFToLSP_GSMAMR_16s (const Ipp16s *pLSF, Ipp16s *pLSP) ;
```

Arguments

`pLSF`	Pointer to the vector [10] of LSF normalized by factor $1/\pi$ in Q14, so given in the range [0,1].
`pLSP`	Pointer to the LSP vector [10], in Q15 in the range [-1,1].

Discussion

The function `ippsLSPToLSF_GSMAMR` is declared in `ippsc.h` file. This function converts the Line Spectral Frequencies (LSF) to the LSP coefficients as given by:

$$lsp_n = \cos(lsf_n), 1 \le n \le 10$$

Return Value

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

LSPQuant_GSMAMR

Quantizes the LSP coefficient vector.

```
IppStatus ippsLSPQuant_GSMAMR_16s(const Ipp16s * pSrcLsp, Ipp16s *
   pSrcDstPrevQLsfResidual, Ipp16s * pDstQLsp, Ipp16s * pDstQLspIndex,
   IppSpchBitRate  mode);
```

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. On output, points to the ten-element quantized LSF residual for the current frame, represented using Q0.15.
pDstQLsp	Pointer to the 20-element quantized LSP vector, represented using Q0.15. For 12.2 kbps frames, elements 0 to 9 contain the first quantized LSP set, and elements 10 to 19 contain the second quantized LSP set. For all other bit rates there is only one LSP set contained in elements 0 to 9.

pDstQLspIndex	Pointer to the five-element vector of quantized LSP indices. For 12.2Kbps frames, all five elements contain valid data; for all other bit rates, only the first three elements contain valid indices (see the following discussion of SMQ and SVQ for the various modes).
mode	Bit rate specifier. The enumerated values of `IPP_SPCHBR_4750` to `IPP_SPCHBR_12200` are valid.

Discussion

The function `ippsLSPQuant_GSMAMR` is declared in `ippsc.h` file. This function quantizes the LSP coefficient vector, then obtains quantized LSP codebook indices. The functionality can be summarized as follows:

1. LSP-to-LSF conversion – the LSPs are mapped to line spectrum frequencies (LSFs). For 12.2 kbps frames, two sets of LSPs are converted to LSFs. For all other bit rates, a single set of LSPs is converted using the 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

$$w_i = \begin{cases} 3.347 - \frac{1.547}{450} d_i, & d_i < 450 \\ 1.8 - \frac{0.8}{1050}(d_i - 450), & otherwise \end{cases}$$

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.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when the *`pSrcLsp`*, *`pSrcDstPrevQLsfResidual`*, *`pDstQLsp`*, or *`pDstQLspIndex`* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *`mode`* is not a valid element of the enumerated type `IppSpchBitRate`.

QuantLSPDecode_GSMAMR

Decodes quantized LSPs.

```
IppStatus ippsQuantLSPDecode_GSMAMR_16s(const Ipp16s * pSrcQLspIndex, Ipp16s *
   pSrcDstPrevQLsfResidual, Ipp16s * pSrcDstPrevQLsf, Ipp16s *
   pSrcDstPrevQLsp, Ipp16s * pDstQLsp, Ipp16s bfi, IppSpchBitRate mode);
```

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. On output, points to the ten-element updated LSF residual, represented using Q0.15.
pSrcDstPrevQLsf	Pointer to the ten-element quantized LSF vector from the previous frame, represented using Q0.15. On output, points to the ten-element updated quantized LSF vector, represented using Q0.15.
pSrcDstPrevQLsp	Pointer to the ten-element quantized LSP vector from the previous frame, represented using Q0.15. On output, points to the ten-element updated quantized LSP vector, represented using Q0.15.
pDstQLsp	Pointer to a 40-element vector containing four subframe LSP sets. Two sets are generated by interpolation for 12.2 kbps frames; for all other bit rates three sets are generated by interpolation. All elements are represented in using Q0.15.
bfi	Bad frame indicator; “0” signifies a good frame; all other values signify a bad frame.
mode	Bit rate specifier. The enumerated values of `IPP_SPCHBR_4750` to `IPP_SPCHBR_12200` are valid.

Discussion

The function `ippsQuantLSPDecode_GSMAMR` is declared in `ippsc.h` file. This function decodes quantized LSPs from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized LSPs from previous quantized LSPs using linear interpolation. The functionality can be summarized as follows:

1. If no errors are detected on the current frame, obtain the quantized LSPs from the codebook indices and the previous quantized residual using inverse LSP quantization.
2. If errors are detected on the current frame, quantized LSFs are obtained using the following rate-dependent interpolation scheme:

$$\begin{cases} lsf\_q1(i) = lsf\_q2(i) = \alpha \times past\_lsf\_q(i) + (1-\alpha) \times mean\_lsf(i) & 12.2\,\text{kbit/s mode} \\ lsf\_q(i) = \alpha \times past\_lsf\_q(i) + (1-\alpha) \times mean\_lsf(i) & otherwise \end{cases}$$

where $i = 0, 1, \ldots, 9$, $\alpha = 0.95$, *lsf_q*1 and *lsf_q*2 (for 12.2 kbps) are two sets of quantized LSF vectors for current frame, *past_lsf_q* is *lsf_q*2 of the previous frame, and *mean_lsf* is the average LSF vector. Note that there is only one set of quantized LSF coefficients for rates other than 12.2 kbps. The corresponding quantized LSPs are obtained by LSF-to-LSP conversion.

3. Linear interpolation is applied to generate four sets of quantized LSPs from the decoded set(s) of LSPs and the quantized LSPs from the previous frame.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when the *`pSrcQLspIndex`*, *`pSrcDstPrevQLsfResidual`*, *`pSrcDstPrevQLsf`*, *`pSrcDstPrevQLsp`* or *`pDstQLsp`* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *`mode`* is not a valid element of the enumerated type `IppSpchBitRate`.

Adaptive Codebook 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.

Open-Loop Pitch Search (OLP)

Three functions are provided for OLP search. Use of these functions is mutually exclusive, that is, only one is appropriate for an application during any given frame type. The appropriate choice of an OLP search function depends upon the state of the DTX and VAD modules. The OLP search functions should be applied as follows:

Encoder Mode	Appropriate Function
DTX disabled	`ippsOpenLoopPitchSearchNonDTX_GSMAMR`
DTX, VAD 1 enabled	`ippsOpenLoopPitchSearchDTXVAD1_GSMAMR`
DTX, VAD 2 enabled	`ippsOpenLoopPitchSearchDTXVAD2_GSMAMR`

The OLP search functions extract a pitch estimate from a weighted version of the input speech. For 5.15 and 4.75 kbps frames, the search is performed once per frame. For all other modes, the search is performed twice per frame. A unique search is employed for 10.2 kbps frames. The OLP search details are as follows:

1. If the transmission bit rate is 10.2 kbps,
   a. Compute a windowed autocorrelation of weighted speech, that is,

$$R(k) = w(k) \times \sum_{n=0}^{79} sw(n) \times sw(n-k), \quad 20 \le k \le 143$$

where the sequence $sw(n)$ contains weighted speech, and w is the weighting function given by

$$w(k) = wl(k) \times wn(k)$$

In this weighting function, *wl* emphasizes low pitch, and is defined in terms of the table *cw*, while *wn* is a sequence of neighboring emphasis lag coefficients associated with the previous frame, i.e.,

$$wn(k) = \begin{cases} cw(|T_{old} - k| + 20) & weightflag = 1 \\ 1 & otherwise \end{cases}$$

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

$$g = \frac{\sum_{n=0}^{79} sw(n) \times sw(n - T_{op})}{\sum_{n=0}^{79} sw(n) \times sw(n)} - 0.4$$

In addition,

$$v = \begin{cases} 1 & g > 0 \\ 0.9v & otherwise \end{cases}$$

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 are found in three different ranges for the correlations given by

$$R(k) = \sum_{n=0}^{length-1} sw(n) \times sw(n - k)$$

For 5.15 and 4.75 kbps frames, *length* = 160. For all other bit rates (except 10.2 kbps), *length* = 80.

b. Normalize the three maximum correlations according to

$$M_i = \frac{M_i}{\sqrt{\sum_{n=0}^{length-1} sw^2(n - T_i)}} \qquad i= 1,2,3$$

c. Determine the best open-loop lag using the following rule:

$$T_{op} = T_1, \quad M(T_{op}) = M_1$$
$$if\,(M_2 > 0.85M(T_{op}))$$
$$M(T_{op}) = M_2, \quad and\ T_{op} = T_2$$
$$if\,(M_3 > 0.85M(T_{op}))$$
$$T_{op} = T_3$$

Each of the OLP search functions implements the rate-dependent search algorithms described above. Next, details are given for non-DTX, VAD1, and VAD2 OLP search primitives.

OpenLoopPitchSearchNonDTX_GSMAMR

Computes the open-loop pitch lag.

```
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);
```

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. On output, points to the updated median filtered pitch lag vector (5 previous voiced speech half-frames), represented using Q15.0. This argument is valid only for 10.2 kbps frames.
pValResultVvalue	Pointer to the adaptive parameter *v* described in Open-Loop Pitch Search (OLP) , represented using Q0.15. Used also as an output argument. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevPitchLag	Pointer to the five-element vector that contains the pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five previous voiced half-frames. This argument is used only for 10.2 kbps frames.
pSrcDstPrevWgtSpch	Pointer to a 143-element vector containing perceptually weighted speech from the previous frame, represented using Q15.0. On output, points to the updated 143-element vector of perceptually weighted speech, represented using Q15.0.

pDstOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
pDstOpenLoopGain	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Discussion

The function `ippsOpenLoopPitchSearchNonDTX_GSMAMR` is declared in `ippsc.h` file. This function computes the open-loop pitch lag (as well as optimal pitch gain for 10.2 kbps frames only) when both DTX and VAD are disabled. The search algorithm is described in Open-Loop Pitch Search (OLP) .

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when any of the input or output pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *mode* is not a valid element of the enumerated type `IppSpchBitRate`.

OpenLoopPitchSearchDTXVAD1_GSMAMR

Extracts an open-loop pitch lag estimate (VAD 1 scheme is enabled).

```
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);
```

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. On output, points to the updated tone flag for the VAD module.
pValResultPrevMidPitchLag	Pointer to a vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.

pValResultVvalue	Pointer to the adaptive parameter *v*, represented using Q0.15. On output, points to the updated adaptive parameter *v*, represented using Q0.15. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevPitchLag	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevWgtSpch	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output, points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.
pResultMaxHpCorr	Pointer to the correlation maximum.
pDstOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
pDstOpenLoopGain	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Discussion

The function `ippsOpenLoopPitchSearchDTXVAD1_GSMAMR` is declared in `ippsc.h` file. This function extracts an open-loop pitch lag estimate from the weighted input speech when the VAD 1 scheme is enabled using a version of the pitch search algorithm described in Open-Loop Pitch Search (OLP) , that is modified as follows:

1. For 10.2 kbps frames, the following modification is applied after the best open-loop pitch is found:
   a. Update the tone flag (when initialized or reset, it is set to 0) in the following way

$$toneflag >>= 1$$

$$DelayEnergy = \sum_{n=0}^{79} sw^2(n - T_{op})$$

$$MaxCorr = \sum_{n=0}^{79} sw(n - T_{op}) \times sw(n)$$

$$if\ (0.325 \times DelayEnergy < MaxCorr) \quad toneflag = toneflag \mid 0x4000$$

   b. On the second OLP search for the current frame, find the maximum of the high-pass filtered autocorrelations, i.e.,
   $maxhpcorr = \max(R(k) \times 2 - R(k-1) - R(k+1) \mid k = 142,..., 24)$

Then, *maxhpcorr* is normalized by *NormFactor* = *frameEnergy* - *frameCorr*:

$$frameEnergy = \sum_{n=0}^{79} sw^2(n) \ , \qquad rameCorr = \sum_{n=0}^{79} sw(n) \times sw(n-1)$$

2. For all other bit rates, the following modifications are applied:
   a. Before the open-loop pitch search, update the tone flag as follows:

$$toneflag = toneflag \gg 1$$

   If the bit rate is either 5.15 or 4.75 kbps, update the tone flag as follows:

$$toneflag = toneflag \gg 1, \quad toneflag = toneflag \mid 0x2000$$

   b. After finding three open-loop pitch candidates, update the tone flag as follows:
   if ($DelayEnergy \times 0.65 < MaxCorr$) $toneflag = toneflag \mid 0x4000$

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.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when any of the input or output pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *`mode`* is not a valid element of the enumerated type `IppSpchBitRate`.

OpenLoopPitchSearchDTXVAD2_GSMAMR

Extracts an open-loop pitch lag estimate (VAD 2 scheme is enabled).

```
IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR(const Ipp16s * pSrcWgtLpc1,
   const Ipp16s * pSrcWgtLpc2, const Ipp16s * pSrcSpch,
   Ipp16s * pValResultPrevMidPitchLag, Ipp16s * pValResultVvalue,
   Ipp16s * pSrcDstPrevPitchLag, Ipp16s * pSrcDstPrevWgtSpch,
   Ipp32s * pResultMaxCorr, Ipp32s pResultWgtEnergy, Ipp16s * pDstOpenLoopLag,
   Ipp16s * pDstOpenLoopGain, IppSpchBitRate mode);
IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR_16s32s(const Ipp16s *
   pSrcWgtLpc1, const Ipp16s * pSrcWgtLpc2, const Ipp16s * pSrcSpch, Ipp16s *
   pValResultPrevMidPitchLag, Ipp16s * pValResultVvalue, Ipp16s *
   pSrcDstPrevPitchLag, Ipp16s * pSrcDstPrevWgtSpch, Ipp32s * pResultMaxCorr,
   Ipp32s pResultWgtEnergy, Ipp16s * pDstOpenLoopLag, Ipp16s *
   pDstOpenLoopGain, IppSpchBitRate mode);
```

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. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
pValResultVvalue	Pointer to the adaptive parameter *v*, represented using Q0.15. On output, points to the updated adaptive parameter *v*, represented using Q0.15. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevPitchLag	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
pSrcDstPrevWgtSpch	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output,

	points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.
pResultMaxCorr	Pointer to the correlation maximum.
pResultWgtEnergy	Pointer to the pitch delayed energy of the weighted speech signal, as described above. The output may be scaled, and the Q representation is not given.
pDstOpenLoopLag	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
pDstOpenLoopGain	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Discussion

The function `ippsOpenLoopPitchSearchDTXVAD2_GSMAMR` is declared in `ippsc.h` file. This function extracts an open-loop pitch lag estimate from the weighted input speech when the VAD 2 scheme is enabled, using a version of the pitch search algorithm described in Open-Loop Pitch Search (OLP) that is modified in the following way:

After finding the best open-loop pitch, extract from the weighted speech the maximum correlation *MaxCorr* and the delayed energy *DelayEnergy*. For both 4.75 and 5.15 kbps frames, the computation is carried for 160 samples. For all other bit rates, the computation is carried for only 80 samples using the relations

$$MaxCorr = \sum_{n=0}^{length-1} sw(n) \times sw(n - T_{op})$$

$$DelayEnergy = \sum_{n=0}^{length-1} sw^2(n - T_{op})$$

The *MaxCorr* and *DelayEnergy* values corresponding to the two half-frame open-loop pitch lag estimates are combined to obtain whole-frame estimates of *MaxCorr* and *DelayEnergy* (except during 4.75 and 5.15 kbps frames, when the OLP search is performed only once per frame).

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when any of the input or output pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *`mode`* is not a valid element of the enumerated type `IppSpchBitRate`.

ImpulseResponseTarget_GSMAMR

Computes the impulse response and target signal required for the adaptive codebook search.

```
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);
```

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.
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.

Discussion

The function `ippsImpulseResponseTarget_GSMAMR` is declared in `ippsc.h` file. This function computes the impulse response and target signal required for the adaptive codebook search. This function is performed on a subframe basis using the following approach:

1. The impulse response $h(n)$ of the weighted synthesis filter,

$$H(z)W(z) = A(z/\gamma_1)/[\hat{A}(z)A(z/\gamma_2)] \ ,$$

   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, i.e.,

$$res_{LP}(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i s(n-i)$$

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error when the *pSrcSpch*, *pSrcWgtLpc1*, *pSrcWgtLpc2*, *pSrcQLpc*, *pSrcSynFltState*, *pSrcWgtFltState*, *pDstImpulseResponse*, *pDstLpResidual*, or *pDstAdptTarget* pointer is NULL.

AdaptiveCodebookSearch_GSMAMR

Performs the adaptive codebook search.

```
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);
```

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.
pValResultPrevIntPitchLag	Pointer to the current integer pitch lag.
pSrcDstExcitation	Pointer to the updated 194-element excitation vector. Elements 0 ~ 153 contain past excitations, and are represented using Q15.0. Elements 154 ~ 193 contain the 40-sample adaptive codebook vector, *v*.
pResultFracPitchLag	Pointer to the fractional pitch lag obtained during the adaptive codebook search.
pResultAdptIndex	Pointer to the coded closed-loop pitch index.
pDstAdptVector	Pointer to the 40-sample adaptive codebook vector, represented using Q15.0.
subFrame	Subframe index.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.

Discussion

The function `ippsAdaptiveCodebookSearch_GSMAMR` is declared in `ippsc.h` file. This function performs the adaptive codebook search. The adaptive codebook search consists of a closed-loop pitch search followed by computation of an adaptive excitation vector. The adaptive excitation vector is obtained by interpolating the past excitation at the fractional pitch lag obtained during the closed-loop pitch search. The adaptive codebook is searched on every subframe. A detailed description of the adaptive codebook search procedure is given next:

1. A closed-loop pitch analysis is performed in the neighborhood of the open-loop pitch estimate, T_{op}, on a subframe basis. In the first and third subframes (only the first subframe for 5.15 and 4.75 kbps modes), the search neighborhood is rate-dependent. For 12.2 kbps frames, the search range is T_{op} ±3, and is bounded

by 18…143. For 5.15 or 4.75 kbps frames, the search range is T_{op} ±5, and is bounded by 20…143. For all other bit rates, the search range is T_{op} ±3, and is bounded by 20…143. For the second and fourth subframes, the search neighborhood surrounds T_1, the nearest integer to the fractional pitch lag of the previous subframe, and the neighborhood boundaries are also rate-dependent. For 12.2 kbps frames, the search range is [T_1 – 5 3/6, T_1 + 4 3/6]. For 7.95 kbps frames, the search range is [T_1 – 10 2/3, T_1 + 9 2/3]. For 10.2 or 7.40 kbps frames, the search range is [T_1 – 5 2/3, T_1 + 4 2/3]. For all other bit rates, the search range is [T_1 – 5, T_1 + 4].

2. The optimum integer pitch search minimizes the mean-square weighted error between the original and synthesized speech. This is achieved by maximizing the normalized cross-correlation given by

$$R(k) = \frac{\sum_{n=0}^{39} x(n)y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n)y_k(n)}}$$

where $x(n)$ is the target signal, and $y_k(n)$ is the past filtered excitation at delay k (past excitation convolved with the impulse response $h(n)$). The convolution $y_k(n)$ is computed for the first delay $t_{\min}$ in the searched range. For other delays in the range $k = t_{\min} + 1,\ldots, t_{\max}$, it is updated using the recursive relation

$$y_k(n) = y_{k-1}(n-1) + u(-k)h(n)$$

where $u(n)$, n = -154,…,39 , is the excitation history buffer. To simplify the search, the prediction residual is copied to $u(n)$, n = 0,…39, making it valid for all delays.

3. The fractional pitch search is performed by interpolating $R(k)$ and searching for its maximum. Fractional delay interpolation is achieved using an FIR filter, b_{24}, based on a hamming windowed *sinc* function truncated at ±23 and padded with zeros at ±24, i.e.,

$$R_t(k) = \sum_{i=0}^{3} R(k-i)b_{24}(t+6i) + \sum_{i=0}^{3} R(k+1+i)b_{24}(6-t+6i)$$

where *t* corresponds to the fractional delay. If the bit rate is 12.2 kbps, then the delay is –1/2 to 1/2 with a resolution of 1/6; otherwise the fraction is –2/3 to 2/3 with a resolution of 1/3.

4. The adaptive codebook vector $v(n)$ is computed by interpolating the past excitation signal $u(n)$ at the integer delay k and fractional delay t, i.e.,

$$v(n) = \sum_{i=0}^{9} u(n-k-i)b_{60}(t+6i) + \sum_{i=0}^{9} u(n-k+1+i)b_{60}(6-t+6i)$$

The interpolation filter is based on a Hamming windowed *sinc* function truncated at ±59 and padded with zeros at ±60.

5. For the first and the third subframes (only the first subframe during 4.75 and 5.15 kbps modes), the pitch lag bit allocation is rate-dependent. For 12.2 kbps frames, the pitch lag is encoded with 9 bits. For all other bit rates, the pitch lag is encoded with 8 bits. For the second and fourth subframes, pitch is encoded with 6 bits for the 12.2 and 7.95 kbps modes; 5 bits for 10.2 or 7.4 kbps modes, and 4 bits for all other modes.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when the *`pSrcTarget`*, *`pSrcImpulseResponse`*, *`pSrcOpenLoopLag`*, *`pValResultPrevIntPitchLag`*, *`pSrcDstExcitation`*, *`pResultFracPitchLag`*, *`pResultAdptIndex`* or *`pDstAdptVector`* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when *`mode`* is not a valid element of the enumerated type `IppSpchBitRate`.
`ippStsSizeErr`	Indicates an error when *`subFrame`* is not in the range [0, 3].

AdaptiveCodebookDecode_GSMAMR

Decodes the adaptive codebook parameters.

```
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);
```

Arguments

valAdptIndex	Adaptive codebook index.
pValResultPrevIntPitchLag	Pointer to the previous integer pitch lag. Used as an output argument also.
pValResultLtpLag	Pointer to the LTP-Lag value. Used as an output argument also.
pSrcDstExcitation	Pointer to the 194-element excitation vector. Elements 0 ~ 153 contain the past excitation, represented using Q15.0. Elements 154 ~ 193 are used as a buffer whenever the subframe length exceeds the pitch lag. On output, elements 154 - 193 are updated to contain the adaptive codebook vector.
pResultIntPitchLag	Pointer to the integer pitch.
pDstAdptVector	Pointer to the 40-sample adaptive codebook vector, represented using Q15.0.
subFrame	Subframe index.
bfi	Bad frame indicator. “0” signifies a good frame; any other value signifies a bad frame.
inBackgroundNoise	Flag set when the previous frame is considered to contain background noise and only shows minor energy level changes.

voicedHangover	Counter used to monitor the time since a frame was presumably voiced.
mode	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Discussion

The function ippsAdaptiveCodebookDecode_GSMAMR is declared in ippsc.h file. This function decodes the adaptive codebook parameters transmitted by the encoder, and then applies them to interpolate an adaptive codebook vector. If errors are detected on the received frame, previously received parameters are used to approximate the parameters of the current frame and the adaptive codebook vector interpolation procedure is carried with the approximated parameter set. Adaptive codebook vectors are decoded for every subframe. Details of this primitive are given next:

1. If no errors are detected on the current frame, integer and fractional pitch lags are extracted from the adaptive codebook indices.
2. If errors are detected, the integer pitch is recovered either from the previous integer pitch or the *LTP-Lag*, and the fractional pitch is set to zero. The *LTP-Lag* value is replaced by the integer pitch of the 4th subframe of the previous frame (12.2 Kbps mode) or slightly modified values based on the last correctly received value (all other modes).
3. The same adaptive codebook interpolation procedure described in section 13.4.3 is applied to obtain the adaptive codebook vector.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error when the *pValResultPrevIntPitchLag*, *pValResultLtpLag*, *pSrcDstExcitation*, *pResultIntPitchLag*, or *pDstAdptVector* pointer is NULL.
ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.
ippStsSizeErr	Indicates an error when *subFrame* is not in the range [0, 3].

AdaptiveCodebookGain_GSMAMR

Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.

```
IppStatus ippsAdaptiveCodebookGain_GSMAMR_16s (const Ipp16s * pSrcAdptTarget,
   const Ipp16s* pSrcFltAdptVector, Ipp16s* pResultAdptGain);
IppStatus ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s (const Ipp16s*
   pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, Ipp16s* pResultAdptGain,
   Ipp16s* pResultAdptGainCoeffs);
```

Arguments

pSrcAdptTarget	Pointer to the adaptive target signal vector [40].
pSrcFltAdptVector	Pointer to the filtered adaptive-codebook vector [40], in Q12.
pResultAdptGain	Pointer to the output adaptive-codebook gain g_p, in Q14.
pResultAdptGainCoeffs	Pointer to the output gain coefficients vector [4] , in Q15.

Discussion

The functions `ippsAdaptiveCodebookGain_GSMAMR` and `ippsAdaptiveCodebookGainCoeffs_GSMAMR` are declared in `ippsc.h` file.

`ippsAdaptiveCodebookGain_GSMAMR_16s`. This function calculates the adaptive-codebook gain g_p as given by:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)}$$

bounded by $0 \le g_p \le 1.2$, in Q14,

where x is the adaptive target signal vector, and y is the filtered adaptive-codebook vector.

See also the `ippsAdaptiveCodebookGain_G729` function.

ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s. This function calculates the adaptive-codebook gain g_p in the same way as the ippsAdaptiveCodebookGain_GSMAMR_16s function does, and additionally returns the gain in a different representation given by the following formula:

$$g_p = \frac{c_{xy}\,2^{\exp_{xy}}}{c_{yy}\,2^{\exp_{yy}}}, 1/2 \le |c_{xy}|, |c_{yy}| < 1, Q15$$

The mantissas c_{xy} , c_{yy} and exponents exp_{xy} , exp_{yy} for both the normalized denominator and normalized numerator are returned in the *pResultAdptGainCoeffs* vector:

pResultAdptGainCoeffs[0] = c_{yy},

pResultAdptGainCoeffs[1] = exp_{yy},

pResultAdptGainCoeffs[2] = c_{xy},

pResultAdptGainCoeffs[3] = exp_{xy}

If $c_{xy} < 4$, then zero gain is returned.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcAdptTarget*, *pSrcFltAdptVector*, *pResultAdptGain* or *pResultAdptGainCoeffs* pointer is NULL.

Fixed Codebook Search

This section describes primitives that are concerned with the fixed codebook, including primitives that perform the following functions:

- Fixed (algebraic) codebook search
- Fixed codebook vector decode

AlgebraicCodebookSearch_GSMAMR

Searches the algebraic codebook.

```
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);
```

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. On output, points to the updated 40-element impulse response vector, which is obtained by filtering original impulse response $h(n)$ through the pre-filter $F_E(z)$. It is represented using Q3.12.
pDstFixedVector	Pointer to the 40-element fixed codebook vector $c(n)$, represented using Q2.13.
pDstFltFixedVector	Pointer to the 40-element filtered fixed codebook vector $z(n)$, which is obtained by convolving the impulse response with the fixed codebook vector, represented using Q2.13.
pDstEncPosSign	Pointer to the ten-element buffer that contains the encoded positions and signs of optimal pulses. For 12.2 kbps mode, 10 short words are used to store the result of this encoding. For the 10.2 kbps mode, 7 short words are used. For all other modes, only 2 short words are used.
subFrame	Subframe index, which ranges from 0 to 3.
mode	Bit rate specifier. Values between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` are valid.
pBuffer	Pointer to internal working buffer, of length 1K.

Discussion

The functions `ippsAlgebraicCodebookSearch_GSMAMR` and `ippsAlgebraicCodebookSearchEX_GSMAMR` are declared in `ippsc.h` file. These functions search the algebraic codebook by minimizing the mean square error between the weighted input speech and the weighted synthesized speech. After the fixed codebook vector has been obtained, it is filtered through the weighted synthesis filter to obtain a fixed codebook vector. The positions and signs of the optimal pulses are encoded respectively according to the GSM06.90 specification. Algebraic codebook search is applied on each subframe.

These two functions work identically with the following exception: `ippsAlgebraicCodebookSearchEX_GSMAMR` uses an internal working buffer pointed by *`pBuffer`* allocated by user, but `ippsAlgebraicCodebookSearch_GSMAMR` allocates this internal working buffer in stack.

Details of the algebraic search are given next.

1. Prior to fixed codebook search, if the integer part of the closed-loop pitch is less than the subframe length, filter the impulse response $h(n)$ through the pre-filter $F_E(z)$, i.e.,
$$h(n) = h(n) - \beta h(n - T) \ , \ n = T,..., 39$$
where β is the bounded and quantized adaptive gain, and T is the integer part of the closed pitch.
2. Compute backward filtered target vector **d** by
$$d(n) = \sum_{i=n}^{39} x_2(i) \times h(i-n) \ , \ 0 \le n \le 39$$
where $x_2(n)$ is the fixed target signal used for fixed codebook search.
3. For 12.2 Kbps and 10.2 Kbps modes, the signal $b(n)$ which is used for presetting the pulse amplitudes is computed:
$$b(n) = \frac{res_{LTP}(n)}{\sqrt{\sum_{i=0}^{39} res_{LTP}(i) res_{LTP}(i)}} + \frac{d(n)}{\sqrt{\sum_{i=0}^{39} d(i) d(i)}} \ , \ n = 0,..., 39$$
For other modes, $b(n)$ is equal to the signal $d(n)$.
4. Calculate the symmetric Toepliz matrix Φ for each mode, the element of it is computed by
$$\Phi(i, j) = \sum_{n=j}^{39} h(n-i) \times h(n-j) \ , i \le j \ , \ 0 \le i \le 39$$

5. Search the optimal pulse positions for each mode. An efficient non-exhaustive analysis-by-synthesis search technique is used for 12.2Kbps, 10.2Kbps, 7.95Kbps, 7.4Kbps, 6.70 Kbps modes. An exhaustive analysis-by-synthesis search technique is used for other modes. GSM 06.90 (clause 5.7) contains complete details.
6. Construct the fixed codebook vector $c(n)$ according to optimal pulse positions, then convolve it with $h(n)$ to get filtered fixed codebook vector $z(n)$

$$z(n) = \sum_{i=0}^{39} c(i) \times h(n-i) \quad , \quad 0 \le n \le 39$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when any of the input or output pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *`mode`* is not a valid element of the enumerated type `IppSpchBitRate`.

FixedCodebookDecode_GSMAMR

Decodes the fixed codebook vector.

```
IppStatus ippsFixedCodebookDecode_GSMAMR_16s(const Ipp16s * pSrcFixedIndex,
   Ipp16s * pDstFixedVector, Ipp16s subFrame, IppSpchBitRate mode);
```

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.
`pDstFixedVector`	Pointer to the 40-element fixed codebook vector.
`subFrame`	Subframe index.

mode	Bit rate specifier. Values between IPP_SPCHBR_4750 and IPP_SPCHBR_12200 are valid.

Discussion

The function ippsFixedCodebookDecode_GSMAMR is declared in ippsc.h file. This function decodes the fixed codebook vector from the received fixed codebook index.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error when the *pSrcFixedIndex* or *pDstFixedVector* pointer is NULL.
ippStsRangeErr	Indicates an error when *mode* is not a valid element of the enumerated type IppSpchBitRate.
ippStsSizeErr	Indicates an error when *subFrame* is not in the range [0, 3].

Discontinuous Transmission (DTX)

This section describes primitives that are concerned with discontinuous transmission (DTX), including primitives that perform the following functions:

- Signal pre-emphasis prior to VAD option 2
- VAD Decision Function for Option 1
- VAD Decision Function for Option 2
- Parameter Extraction for the SID frame
- DTX Handler
- DTX Buffering

Each of these primitives is described next.

Preemphasize_GSMAMR

Computes pre-emphasis of an input signal in VAD option 2.

```
IppStatus ippsPreemphasize_GSMAMR_16s (Ipp16s gamma, const Ipp16s *pSrc,
   Ipp16s *pDst, int len, Ipp16s* pMem);
```

Arguments

gamma	The filter coefficient, in Q15.
pSrc	Pointer to the source vector, in Q0.
pDst	Pointer to the destination vector, in Q0.
len	Number of elements in the source and destination vectors.
pMem	Pointer to the filter memory [1].

Discussion

The function `ippsPreemphasize_GSMAMR` is declared in the `ippsc.h` file. This function computes pre-emphasis of the input signal prior to frequency domain conversion in Voice Activity Detector option 2. The function `ippsPreemphasize_GSMAMR` performs the same operation as the ippsPreemphasize_G729A function does, but has a slightly different order to deliver accuracy required by the GSM-AMR transcoding standard.

Return Value

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

VAD1_GSMAMR

Implements the VAD functionality corresponding to VAD option 1.

```
IppStatus ippsVAD1_GSMAMR_16s(const Ipp16s pSrcSpch, IppGSMAMRVad1State *
   pValResultVad1State, Ipp16s * pResultVadFlag, Ipp16s maxHpCorr,
   Ipp16s toneFlag);
```

Arguments

pSrcSpch	Pointer to the input speech signal, of length 160, in Q0.
pValResultVad1State	On input, pointer to the VAD Option 1 history variables. On output, points to the updated VAD Option 1 history variables. The structure `IppGSMAMRVad1State` is defined below.
pResultVadFlag	Pointer to the VAD flag of this frame. If it is set to “1”, it indicates the presence of signals that should be transmitted. If set to “0”, there is no signals in this frame needed to be transmitted.
maxHpCorr	*best_corr_hp* value of previous frame, which is the maximum normalized value of the high pass filtered correlation. This value is the output of the open-loop pitch search function.
toneFlag	Tone flag, which indicates the presence of information tones or signals containing very strong periodic component. This value is the output of the open-loop pitch search function.

Discussion

The function `ippsVAD1_GSMAMR` is declared in `ippsc.h` file. This function implements the VAD functionality corresponding to VAD option 1 of *ETSI GSM 06.94*. It is used to indicate whether each 20ms frame contains signals that should be transmitted - for example, speech, music or information tones. The structure `IppGSMAMRVad1State` contains the history variables of VAD Option 1. These variables are initialized before the beginning of the encoder, and can be only updated by this f unction. Refer to *ETSI GSM 06.94* VAD Option 1 specification for details of the implementation.

Structure definition of `IppGSMAMRVad1State`:

`typedef struct{`	**Description**
`Ipp16s pPrevSignalLevel[9];`	Signal level vector of *level*[*n*] previous frame.
`Ipp16s pPrevSignalSublevel[9];`	Intermediate signal sublevel vector of previous frame.
`Ipp16s pPrevAverageLevel[9];`	Average signal level vector *ave_level*[*n*] of previous frame.
`Ipp16s pBkgNoiseEstimate[9];`	Background noise estimate vector *back_est*[*n*] of previous frame.
`Ipp16s pFifthFltState[6];`	The history state of the three 5th order filters of filter bank.
`Ipp16s pThirdFltState[5];`	The history state of the five 3rd order filters of filter bank.
`Ipp16s burstCount;`	Burst counter *burst_count* which counts length of a speech burst, used by VAD hangover addition.
`Ipp16s hangCount;`	Hang counter *hang_counter* which is used by VAD hangover addition.
`Ipp16s statCount;`	Stationary counter variable *stat_count* which is used in background noise estimation.
`Ipp16s vadReg;`	Value that indicates intermediate VAD decision.
`Ipp16s complexHigh;`	*complex_high* value which is used as intermediate complex signal decision.
`Ipp16s complexLow;`	*complex_low* value which is used as intermediate complex signal decision.
`Ipp16s complexHangTimer;`	*complex_hang_timer* which is used as hangover initiator by Complex Activity Estimation.
`Ipp16s complexHangCount;`	*complex_hang_count* which is used as hangover counter by VAD hangover addition.
`Ipp16s complexWarning;`	*complex_warning* flag.
`Ipp16s corrHp;`	The high-pass filtered value of *best_corr_hp.*
`Ipp16s pitchFlag;`	Pitch flag which indicates the presence of vowel sounds and other periodic signals.

`}IppGSMAMRVad1State.`

Note. **VAD option 1 history variables initialization**:
Whenever the Encoder is reset, all elements in `pPrevSignalLevel, pPrevSignalSublevel, pPrevAverageLevel` vector should be set as 150, `corrHp` should be set as 13106. While all other variables should be initialized as 0.
For the detail usage of these history variables, please refer to *ETSI GSM 06.94*.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when any of the input or output pointers is `NULL`.

VAD2_GSMAMR

Implements the VAD functionality corresponding to VAD option 2.

```
IppStatus ippsVAD2_GSMAMR_16s(const Ipp16s * pSrcSpch, IppGSMAMRVad2State *
   pValResultVad2State, Ipp16s * pResultVadFlag, Ipp16s ltpFlag);
```

Arguments

`pSrcSpch`	Pointer to the input speech frame, of length 160, in Q0.
`pValResultVad2State`	On input, pointer to the VAD Option 2 history variables. On output, points to the updated VAD Option 2 history variables. The structure `IppGSMAMRVad2State` is defined below.
`pResultVadFlag`	Pointer to the Boolean flag *`VAD_flag`*. If it is set to “1”, it indicates the presence of signals that should be transmitted. If set to “0”, there is no signals in this frame needed to be transmitted.
`ltpFlag`	*LTP_flag* value in *GSM 06.94* equation (4.24), which is generated by the comparison of the long-term prediction to a constant threshold *`LTP_THLD`*.

Discussion

The function `ippsVAD2_GSMAMR` is declared in `ippsc.h` file. This function implements the VAD functionality corresponding to VAD option 2 of *ETSI GSM 06.94*. It is used to indicate whether each 20ms frame contains signals that should be transmitted. For example, speech, music, or information tones. The structure `IppGSMAMRVad2State` contains the history variables of VAD Option 2.

Please refer to *ETSI GSM 06.94* VAD Option 2 specification for details.

Structure definition of `IppGSMAMRVad2State`:

`typedef struct{`	**Description**
`Ipp32s pEngyEstimate[16];`	Channel energy estimates vector $\boldsymbol{E}_{ch}$ of current half-frame, which is calculated during the previous half-frame according to the equation *ETSI GSM 06.94 (4.4)*.
`Ipp32s pNoiseEstimate[16];`	Channel noise estimate vector $\boldsymbol{E}_{n}$ of current half-frame, which is calculated during the previous half-frame according to the equation *ETSI GSM 06.94 (4.26)* .
`Ipp16s pLongTermEngyDb[16];`	Channel average long-term spectral estimate vector $\bar{E}_{dB}$, 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 $\psi$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;`	

Note. All elements in the structure should be initialized to zero whenever encoder is reset.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when any of the input or output pointers is `NULL`.

EncDTXSID_GSMAMR

Extracts parameters for the SID frame.

```
IppStatus ippsEncDTXSID_GSMAMR_16s(const Ipp16s * pSrcLspBuffer, const
   Ipp16s * pSrcLogEnergyBuffer, Ipp16s * pValResultLogEnergyIndex,
   Ipp16s * pValResultDtxLsfRefIndex, Ipp16s * pSrcDstQLsfIndex,
   Ipp16s * pSrcDstPredQErr, Ipp16s * pSrcDstPredQErrMR122, Ipp16s sidFlag);
```

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 to log energy index of last frame, in Q2.13. On output, points to the log energy index of current frame, in Q2.13.
pValResultDtxLsfRefIndex	Pointer to the LSF quantization reference index of last frame. On output, points to the LSF quantization reference index of current DTX frame.
pSrcDstQLsfIndex	Pointer to the LSF residual quantization indices of last frame, in the length of 3. On output, points to the LSF residual quantization indices of current frame, in the length of 3.

pSrcDstPredQErr	Pointer to the fixed gain prediction error of four previous subframes for non-12.2 Kbps modes, in the length of 4, in Q5.10. On output, points to the updated fixed gain prediction error for non 12.2 Kbps modes, in the length of 4, in Q5.10.
pSrcDstPredQErrMR122	Pointer to the fixed gain prediction error of four previous subframes for 12.2 Kbps, in the length of 4, in Q5.10. On output, points to the updated fixed gain prediction error for 12.2 Kbps mode, in the length of 4, in Q5.10.
sidFLag	The SID flag of the current frame. If it is set to 1, the current frame is a SID frame, and the function will extract the LSF and energy parameters. If it is set to 0, the LSF and energy parameters will copy from previous frame.

Discussion

The function `ippsEncDTXSID_GSMAMR` is declared in `ippsc.h` file. This function is called only when the current frame is a DTX frame. It extracts the needed parameters for the SID frame (that is, the LSF quantization parameter and the energy index parameter). If the SID flag is off, no operation is needed, and all the parameters are copied from last frame. If the SID flag is on, the functionality is as below:

1. Compute the log energy index

   $$EnLogIndex = \frac{1}{8}\sum_{n=0}^{7} EnLog(t-n) \ ,$$

   where $EnLog(t)$ is the log energy of the current frame in log2 scale.

2. Update the fixed gain prediction error for 12.2 Kbps mode and other modes respectively with the log energy index:
   $PredErr(i) = EnLogIndex$
   $PredErrMR122(i) = (EnLogIndex) / (20 \times \log10(2))$
   $i = 0,..., 3$

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.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when any of the input or output pointers is `NULL`.

EncDTXHandler_GSMAMR

Determines the SID flag of current frame.

```
IppStatus ippsEncDTXHandler_GSMAMR_16s(Ipp16s * pValResultHangOverCount,
   Ipp16s *pValResultDtxElapsedCount, Ipp16s * pValResultUsedMode,
   Ipp16s * pResultSidFlag, Ipp16s vadFlag);
```

Arguments

`pValResultHangOverCount`	Pointer to the DTX hangover count. When initialized or reset, it is set to 0. On output, points to the updated DTX hangover count.

pValResultDtxElapsedCount	Pointer to elapsed frame count since last non-DTX frame. When initialized or reset, it is set 0. On output, points to the updated elapsed frame count since last non-DTX frame.
pValResultUsedMode	Pointer to the transmission mode. At the input stage, the mode is one of the bit rate modes ranging from 4.75 Kbps to 12.2 Kbps. At the output stage, this value is either unchanged or set to the DTX frame mode.
pResultSidFlag	Pointer to the output SID flag, “1” indicates a SID frame, and “0” indicates a non-SID frame.
vadFlag	This is the VAD flag of the current frame, if it is set 1, the current frame is marked with voiced, and if it is set to 0, it is marked with unvoiced.

Discussion

The function `ippsEncDTXHandler_GSMAMR` is declared in `ippsc.h` file. This function determines the SID flag of current frame, and it determines whether the current frame should use DTX encoding.

1. Update the elapsed frame count since last SID frame: *DTX_ElapsedCount* = *DTX_ElapsedCount* + 1(Bounded to 0~0x7fff).
2. If the VAD flag of current frame is 1 (voiced frame), the DTX hangover count is set to 7, and this function ends.
3. If the VAD flag of current frame is 0 (unvoiced frame), and the DTX hangover count is 0, the transmission mode of this frame is set to DTX frame mode, and the elapsed frame count since last DTX frame is set to 0, the SID flag is set to 1.
4. If the VAD flag of current frame is 0 (unvoiced frame), but the DTX hangover count is not 0, then decrease DTX hangover count by 1, and if *DTX_HangOver_Count* + *DTX_ElapsedCount* < 30
   The SID flag is set to 0, and the transmission mode is set to “MRDTX”.

Return Value

`ippStsNoErr`	Indicates no error.

ippStsBadArgErr	Indicates an error when any of the input or output pointers is NULL.

EncDTXBuffer_GSMAMR, DecDTXBuffer_GSMAMR

Buffer the LSP (or LSF) coefficients and previous log energy coefficients.

```
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);
```

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. On output, points to the current memory update index. It is circularly increased between 0 and 7.
pSrcDstLspBuffer	Pointer to the LSP coefficients of eight previous frames, in the length of 80, in Q0.15. On output, points to the LSP coefficients of eight most recent frames (including current frame), in the length of 80, in Q0.15.

pSrcDstLogEnergyBuffer	Pointer to the logarithm energy coefficients of eight previous frames, in the length of 8, in Q5.10. On output, points to the log energy coefficients of eight most recent frames (including current frame), in the length of 8, in Q5.10.

Discussion

The functions ippsEncDTXBuffer_GSMAMR and ippsDecDTXBuffer_GSMAMR are declared in ippsc.h file. These functions buffer the LSP (or LSF) coefficients and previous log energy coefficients. These LSPs (or LSFs) and energy coefficients will be used for SID frame to extract necessary parameters. The memory update index indicates which part of the buffer will be updated, and it saves the cost for some memory copy. The log energy is computed as follows:

$$EnLog = \log 2\left(\frac{1}{N}\sum_{n=0}^{N-1} s^2(n)\right),$$

where N is the frame length, and s is the input speech signal.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error when any of the input or output pointers is NULL.

Post Processing

PostFilter_GSMAMR

Filters the synthesized speech.

```
IppStatus ippsPostFilter_GSMAMR_16s(const Ipp16s * pSrcQLpc,
   const Ipp16s * pSrcSpch, Ipp16s * pValResultPrevResidual,
   Ipp16s * pValResultPrevScalingGain, Ipp16s * pSrcDstFormantFIRState,
   Ipp16s * pSrcDstFormantIIRState, Ipp16s * pDstFltSpch, IppSpchBitRate mode);
```

Arguments

pSrcQLpc	Pointer to the reconstructed LP coefficients, in the length of 44, in Q3.12.
pSrcSpch	Pointer to the start position of the input speech signal for current frame, in the length of 160, in Q15.0.
pValResultPrevResidual	On entry, pointer to the last output of the FIR filter of the formant filter for previous subframe, in Q15.0. It is the input of the tilt compensation filter. On exit, points to the last output of the FIR filter of the formant filter for this subframe (in Q15.0) and is the output of the tilt compensation filter. This value is initialized to 0 and can only be updated by this function.
pValResultPrevScalingGain	Pointer to the scaling factor *b* of the last signal for the previous subframe, in Q3.12. On output, points to the scaling factor *b* of the last signal for this subframe, in Q3.12
pSrcDstFormantFIRState	Pointer to the state of the FIR part of the formant filter, in the length of 10, in Q15.0. On output, points to the updated state of the FIR part of the formant filter, in the length of 10, in Q15.0.

pSrcDstFormantIIRState	Pointer to the state of the IIR part of the formant filter, in the length of 10, in Q15.0. On output, points to the updated state of the IIR part of the formant filter, in the length of 10, in Q15.0.
pDstFltSpch	Pointer to the filtered speech, in the length of 160, in Q15.0.
mode	The source transmission mode, for this function, the value between `IPP_SPCHBR_4750` and `IPP_SPCHBR_12200` is valid.

Discussion

The function `ippsPostFilter_GSMAMR` is declared in `ippsc.h` file. This function filters the synthesized speech to enhance reconstruction quality. This primitive implements the following procedure:

1. Get the weighted LP coefficients of the formant postfilter:

$$H_f(z) = \frac{\hat{A}(z/\gamma_n)}{\hat{A}(z/\gamma_d)}$$

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:

$$k_1' = \frac{r_h(1)}{r_h(0)} \quad , \quad r_h(i) = \sum_{j=0}^{21-i} h(j) \times h(j+i)$$

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:

$HI(z) = \hat{A}(z/\gamma_d)$

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:

$$\hat{s}'(n) = \beta_{sc}(n)\hat{s}_f(n)$$

The scaling factor is updated sample-by-sample:

$$\beta_{sc}(n) = 0.9\beta_{sc}(n-1) + 0.1\gamma_{sc}$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when any of the input or output pointers is `NULL`, or the input variable *mode* is out of range.

GSM Full Rate Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following the GSM 06.10, 06.11, 06.12, 06.31, and 06.32 recommendations. The list of these functions is given in the following table:

Table 9-5 Intel IPP GSM Full Rate Related Functions

Function Base Name	Operation
RPEQuantDecode_GSMFR	Performs APCM inverse quantization.
Deemphasize_GSMFR	Performs de-emphasis filtering.
ShortTermAnalysisFilter_GSMFR	Performs short-term analysis filtering.
ShortTermSynthesisFilter_GSMFR	Performs short-term synthesis filtering.
HighPassFilter_GSMFR	Filters input speech signal through a high-pass filter.
Schur_GSMFR	Estimates the reflection coefficients by Schur recursion.
WeightingFilter_GSMFR	Calculates the weighting filter.
Preemphasize_GSMFR	Computes pre-emphasis of a speech signal.

RPEQuantDecode_GSMFR

Performs APCM inverse quantization.

```
IppStatus ippsRPEQuantDecode_GSMFR_16s (const Ipp16s *pSrc, Ipp16s ampl,
   Ipp16s amplSfs, Ipp16s *pDst);
```

Arguments

pSrc	Pointer to the input vector [13] of the RPE samples.
ampl	The block amplitude.
amplSfs	Scale factor of the block amplitude.
pDst	Pointer to the output reconstructed long-term residual vector [13].

Discussion

The function ippsRPEQuantDecode_GSMFR is declared in ippsc.h file. This function performs APCM inverse quantization of the input RPE samples. The output reconstructed long-term residual vector is formed as

$$pDst[i] = (2 * pSrc[i] - 7) \cdot \frac{ampl}{2^{amplSfs}}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsRangeErr	Indicates an error when *amplSfs* is less then 0.

Deemphasize_GSMFR

Performs de-emphasis filtering.

```
IppStatus ippsDeemphasize_GSMFR_16s_I (Ipp16s *pSrcDst, int len,
   Ipp16s *pMem);
```

Arguments

pSrcDst	Pointer to the input short-term synthesized signal and output post-processed speech vector [*len*].
len	Length of the input residual and output speech vectors.
pMem	Pointer to the filter memory element.

Discussion

The function ippsDeemphasize_GSMFR is declared in ippsc.h file. This function performs de-emphasis of the input synthesized signal by filtering it through the filter with the following transfer function:

$$H(z) = \frac{1}{1 - \alpha z^{-1}}$$

where $\alpha = 0.86$ (28180 in Q15).

The initial memory of the filter will be set to zero.

The filtered speech signal is scaled up by multiple of 2 and then stored in *pSrcDst* with truncation of the three least significant bits.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst* or *pMem* pointer is NULL.
ippStsRangeErr	Indicates an error when *len* is less or equal to 0.

ShortTermAnalysisFilter_GSMFR

Performs short-term analysis filtering.

```
IppStatus ippsShortTermAnalysisFilter_GSMFR_16s_I (const Ipp16s *pRC,
   Ipp16s *pSrcDstSpch, int len, Ipp16s *pMem);
```

Arguments

pRC	Pointer to the input reflection coefficients vector [8]: $a_1, a_2, \ldots, a_8$.
pSrcDstSpch	Pointer to the input pre-processed speech and output short term residual vector [*len*].
len	Length of the input speech and output residual vectors.
pMem	Pointer to the filter memory vector [8]: $m_0, m_1, \ldots, m_7$.

Discussion

The function ippsShortTermAnalysisFilter_GSMFR is declared in ippsc.h file. This function performs filtering of the input pre-processed speech vector $s(n)$ and stores the result in the output short-term residual vector $r(n)$ as given below:

$$r_0 = s(n)$$
$$r_i = r_{i-1} + a_i \cdot m_{i-1}, i = 1,...,8$$
$$m_0 = s(n)$$
$$m_i = m_{i-1} + a_i \cdot r_{i-1}, i = 1,...,7$$
$$r(n) = r_8$$

where m_i , i = 0,...,7 is the filter memory and r_i , i = 0,...,8 is the reusable local memory.

The initial filter memory vector will be zeroed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRC*, *pSrcDstSpch* or *pMem* pointer is NULL.
ippStsRangeErr	Indicates an error when *len* is less or equal to 0.

ShortTermSynthesisFilter_GSMFR

Performs short-term synthesis filtering.

```
IppStatus ippsShortTermSynthesisFilter_GSMFR_16s (const Ipp16s *pRC, const
   Ipp16s *pSrcResidual, Ipp16s *pDstSpch, int len, Ipp16s *pMem);
```

Arguments

pRC	Pointer to the input reflection coefficients vector [8]: $a_1, a_2, \ldots, a_8$.
pSrcResidual	Pointer to the input reconstructed short term residual vector [*len*].
pDstSpch	Pointer to the output speech vector [*len*].
len	Length of the input residual and output speech vectors.
pMem	Pointer to the filter memory vector [8]: $m_0, m_1, \ldots, m_7$.

Discussion

The function ippsShortTermSynthesisFilter_GSMFR is declared in ippsc.h file. This function performs filtering of the input reconstructed short-term residual $r(n)$ and stores the result in the output speech vector $s(n)$ as given below:

$$s_0 = r(n)$$
$$s_i = s_{i-1} - a_{9-i} \cdot m_{8-i}, i = 1,...,8$$
$$m_{8-i} = m_{7-i} - a_{9-i} \cdot s_i, i = 1,...,7$$
$$s(n) = s_8$$
$$m_0 = s(n)$$

where m_i, $i = 0,...,7$ is the filter memory and s_i, $i = 0,...,8$ is the reusable local memory.

The initial filter memory vector will be zeroed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pRC*, *pSrcResidual*, *pMem*, or *pDstSpch* pointer is NULL.
ippStsRangeErr	Indicates an error when *len* is less or equal to 0.

HighPassFilter_GSMFR

Performs high-pass filtering of the input speech signal.

```
IppStatus ippsHighPassFilter_GSMFR_16s (const Ipp16s *pSrc, Ipp16s *pDst,
   int len, int *pMem);
```

Arguments

pSrc	Pointer to the source speech vector [*len*].
pDst	Pointer to the destination filtered vector [*len*].
len	Length of the source and destination vectors.
pMem	Pointer to the filter memory vector [2].

Discussion

The function `ippsHighPassFilter_GSMFR` is declared in `ippsc.h` file. This function filters the input speech signal according to the transfer function:

$$H(z) = 0.5 \cdot \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

where $\alpha = 0.99899$.

The initial filter memory will be set to zero.

Return Value

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

Schur_GSMFR

Estimates the reflection coefficients by Schur recursion.

```
IppStatus ippsSchur_GSMFR_32s16s (const Ipp32s *pSrc, Ipp16s *pDst,
   int dstLen);
```

Arguments

pSrc	Pointer to the input autocorrelation vector [*dstLen*+1].
pDst	Pointer to the output reflection coefficients vector [*dstLen*].
dstLen	The number of reflection coefficients to estimate.

Discussion

The function ippsSchur_GSMFR is declared in ippsc.h file. This function implements the Schur algorithm according to GSM 06.10 clause 4.2.5. See also ippsSchur function.

Return Value

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

WeightingFilter_GSMFR

Calculates the weighting filter.

```
IppStatus ippsWeightingFilter_GSMFR_16s (const Ipp16s *pSrc, Ipp16s *pDst,
   int dstLen);
```

Arguments

pSrc	Pointer to the input long-term residual vector [-5,..,*dstLen*+4].
pDst	Pointer to the filtered output vector [*dstLen*].
dstLen	The number of filtered elements to calculate.

Discussion

The function ippsWeightingFilter_GSMFR is declared in ippsc.h file. This function performs filtering of the input signal by symmetric FIR filter with predefined taps given below:

taps[i]=[-134, -374, 0, 2054, 5741, 8192, 5741, 2054, 0, -374, -134], i=0,..,10

$$dst[n] = \sum_{i=0}^{10} taps[i] \cdot src[n+i-5], n = 0,.., dstlen-1$$

The result of filtering is stored in *pDst*.

Return Value

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

Preemphasize_GSMFR

Computes pre-emphasis of a speech signal.

```
IppStatus ippsPreemphasize_GSMFR_16s(const Ipp16s *pSrc, Ipp16s *pDst,
   int *pMem, int len);
```

Arguments

pSrc	Pointer to the offset-free input speech signal.
pDst	Pointer to the pre-emphasized output signal.

pMem	Pointer to the filter memory value.
len	Length of the input and output signals.

Discussion

The function ippsPreemphasize_GSMFR is declared in ippsc.h file. This function computes pre-emphasis of the input speech according to the difference signal pre-emphasis equation:

$H(z) = 1 - \gamma z^{-1}$

for γ = -0.86 and *pSrc*[-1] = *pMem*[0].

The result of filtering is stored in *pDst*. The memory value *pMem*[0] is updated by *pSrc*[n-1]. For proper use of this function in GSM Full Rate codec, the memory value will be initialized to zero.

The function ippsPreemphasize_GSMFR performs NR rounding (see Rounding mode).

Return Value

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

G.722.1 Related Functions

This chapter describes Intel IPP functions that can be used in implementing speech codecs following ITU-T recommendations G.722.1.
The list of these functions is given in the following table:

Table 9-6 Intel IPP G.722.1 Related Functions

Function Base Name	Operation
DCTFwd_G722, DCTInv_G722	Computes the forward or inverse discrete cosine transform (DCT) of a signal.
DecomposeMLTToDCT	Decomposes the MLT transform input signal to the form of the DCT input signal.
DecomposeDCTToMLT	Decomposes IDCT output signal to the form of the MLT transform output signal.
HuffmanEncode_G722	Performs Huffman encoding of the quantized amplitude envelope indexes.

DCTFwd_G722, DCTInv_G722

Computes the forward or inverse discrete cosine transform (DCT) of a signal.

```
IppStatus ippsDCTFwd_G722_16s (const Ipp16s *pSrc, Ipp16s *pDst);
IppStatus ippsDCTInv_G722_16s (const Ipp16s *pSrc, Ipp16s *pDst);
```

Arguments

pSrc	Pointer to the source vector[320].
pDst	Pointer to the destination vector[320]

Discussion

The functions `ippsDCTFwd_G722` and `ippsDCTInv_G722` are declared in `ippsc.h` file. These functions compute the forward and inverse discrete cosine transform (DCT) of length 320 as follows:

$$y(m) = \alpha \sum_{n=0}^{319} \cos\left(\frac{\pi}{320} \cdot (n + 0.5) \cdot (m + 0.5)\right) \cdot x(n) \text{ , for forward DCT}$$

where $\alpha = 2/320$;

$$x(n) = \beta \sum_{m=0}^{319} \cos\left(\frac{\pi}{320} \cdot (m + 0.5) \cdot (n + 0.5)\right) \cdot y(m) \text{ , for inverse DCT (IDCT) .}$$

where $\beta = 0.81$.

The formulae are the same except for the different scaling that is applied to meet the bit exact requirement.

Return Value

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *`pSrc`* or `pDst` pointer is `NULL`.

DecomposeMLTToDCT

Decomposes the MLT transform input signal to the form of the DCT input signal.

```
IppStatus ippsDecomposeMLTToDCT_G722_16s(const Ipp16s *pSrcSpch, Ipp16s
   *pSrcDstSpchOld, Ipp16s *pDstSpchDecomposed);
```

Arguments

`pSrcSpch`	Pointer to the source vector [320].
`pSrcSpchOld`	Pointer to the source/destination vector [320] of speech samples of the previous frame.
`pDstSpchDecomposed`	Pointer to the destination vector [320].

Discussion

The function `ippsDecomposeMLTToDCT_G722` is declared in `ippsc.h` file.
This function decomposes the input signal of the modulated lapped transform (MLT) to the form that fits DCT, so that the MLT transform may be performed in two steps: first, decomposition of the input signal, and second, DCT of the decomposed signal.

The decomposed speech signal is computed as follows:

$$v(n) = w(159-n)x(159-n) + w(160+n)x(160+n), 0 \le n \le 159$$

$$v(n+160) = w(319-n)(320+n) - w(n)x(639-n), 0 \le n \le 159$$

where

$$w(n) = \sin\left(\frac{\pi}{640}(n+0.5)\right), 0 \le n \le 319$$

and

$$x(n) = \begin{cases} pSrcDstSpchOld[n], 0 \le n < 319 \\ pSrcSpch(n-320), 320 \le n < 639 \end{cases}$$

The input signal *pSrcSpch* is stored in *pSrcDstSpchOld* for use in the next frame.

Return Value

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *pSrcSpch* or *pDstSpch* or *pSrcSpchOld* pointer is `NULL`.

DecomposeDCTToMLT

Decomposes IDCT output signal to the form of the MLT transform output signal.

```
IppStatus ippsDecomposeDCTToMLT_G722_16s(const Ipp16s *pSrcSpchDecomposed,
   Ipp16s *pSrcDstSpchDecomposedOld, Ipp16s *pDstSpch);
```

Arguments

pSrcSpchDecomposed	Pointer to the source vector [320].
PsrcSpchDecomposedOld	Pointer to the source/destination vector [160] of decomposed speech samples of the previous frame.
pDstSpch	Pointer to the destination vector [320].

Discussion

The function `ippsDecomposeDCTToMLT_G722` is declared in `ippsc.h` file. This function decomposes the output signal of the Inverse DCT to the form that fits the output of the Inverse Modulated Lapped Transform (IMLT), so that the IMLT transform may be performed in two steps: first, Inverse Discrete Cosine Transform (IDCT), and second, decomposition of the IDCT output.

The decomposed signal is computed as follows:

$$pDstSpch(n) = w(n)u(159-n) + w(319-n)u\_old(n), 0 \le n \le 159$$

$$pDstSpch(n+160) = w(160+n)u(n) - w(159-n)u\_old(159-n), 0 \le n \le 159$$

where

$$w(n) = \sin\left(\frac{\pi}{640}(n+0.5)\right), 0 \le n \le 319$$

u(n) = pSrcSpchDecomposed[*n*], and
u_old(n)=pSrcDstDecomposedOld[*n*].

The unused high half of the input signal is stored as *u_old*() for use in the next frame:

$$u\_old(n) = u(n+160), 0 \le n \le 159$$

Return Value

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *`pSrcSpch`* or *`pDstSpch`* or *`pSrcSpchOld`* pointer is `NULL`.

HuffmanEncode_G722

Performs Huffman encoding of the quantized amplitude envelope indexes.

```
IppStatus ippsHuffmanEncode_G722_16s32u(int category, int qntAmpEnvIndex,
   const Ipp16s *pSrcMLTCoeffs, Ipp32u *pDstCode, int *pCodeLength);
```

Arguments

`category`	The category of the Modulated Lapped Transform (MLT) region in the range of [0-7].
`qntAmpEnvIndex`	The quantized amplitude envelope index in the range of [0-63].
`pSrcMLTCoeffs`	Pointer to the source vector [20] of raw MLT coefficients.
`pDstCode`	Pointer to the output Huffman code.
`pCodeLength`	Pointer to the output Huffman code length in bits.

Discussion

The function `ippsHuffmanEncode_G722` is declared in `ippsc.h` file. This function performs Huffman encoding of the quantized index of the amplitude envelope of one of the twenty regions of the MLT coefficients.

Return Value

`ippStsNoErr`	Indicates no error.

IppStsNullPtrErr	Indicates an error when the *pSrcMLTCoeffs* or *pDstCode* or *pCodeLength* pointer is NULL.
IppStsScaleRangeErr	Indicates an error when *category* or *qntAmpEnvIndex* is out of proper range.

G.726 Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following ITU- T recommendations G.726 with Annex A.

The list of these functions is given in the following table:

Table 9-7 Intel IPP G.726.1 Related Functions

Function Base Name	Operation
EncodeGetStateSize_G726	Informative function, returns the number of bytes needed for encoder memory.
EncodeInit_G726	Initializes the memory for the ADPCM encoder.
Encode_G726	Performs ADPCM compression of the uniform PCM input.
DecodeGetStateSize_G726	Informative function, returns the number of bytes needed for decoder memory.
DecodeInit_G726	Initializes the memory for the G726 decoder.
Decode_G726	Performs decompression of the ADPCM bit-stream.

EncodeGetStateSize_G726

Informative function, returns the number of bytes needed for encoder memory.

```
IppStatus ippsEncodeGetStateSize_G726_16s8u (unsigned int* pEncSize);
```

Arguments

pEncSize	Pointer to the output memory size in bytes.

Discussion

The function `ippsEncodeGetStateSize_G726` is declared in `ippsc.h` file. This function gets information about the amount of memory needed to process the G.726 ADPCM compression.

Return Value

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *`pEncSize`* pointer is `NULL`.

EncodeInit_G726

Initializes the memory for the ADPCM encoder.

```
IppStatus ippsEncodeInit_G726_16s8u (IppsEncoderState_G726_16s* pEncMem,
   IppSpchBitRate rate);
```

Arguments

`pEncMem`	Pointer to the input memory buffer of size needed to properly initialize the G.726 encoder.
`rate`	Encode bit rate of the G.726 encoder, must be one of `IPP_SPCHBR_16000`, `IPP_SPCHBR_24000`, `IPP_SPCHBR_32000`, or `IPP_SPCHBR_40000`.

Discussion

The function `ippsEncodeInit_G726` is declared in `ippsc.h` file. This function initializes the memory given by the pointer *`pEncMem`* to enable G.726 ADPCM compression starting from the reset state.

Return Value

`ippStsNoErr`	Indicates no error.

ippStsBadArgErr	Indicates an error when *rate* is not equal to one of the admissible encoding bit rates: IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000.
IppStsNullPtrErr	Indicates an error when the *pEncMem* pointer is NULL.

Encode_G726

Performs ADPCM compression of the uniform PCM input.

```
IppStatus ippsEncode_G726_16s8u (IppsEncoderState_G726_16s* pEncMem,
   const Ipp16s *pSrc, Ipp8u *pDst, unsigned int len);
```

Arguments

pEncMem	Pointer to the memory buffer that has been initialized for ADPCM encode.
pSrc	Pointer to the uniform PCM input speech vector.
pDst	Pointer to the ADPCM bit-stream output vector.
len	The length of input/output vectors.

Discussion

The function ippsEncode_G726 is declared in ippsc.h file. This function performs ADPCM compression of the 14-bit uniform PCM speech input (Recommendation G.726, Annex A) with the bit rate on which the G.726 encoder (with memory pointed to by *pEncMem*) was initialized to operate. Each byte of the output vector contains ADPCM compressed value of two, three, four, or five binary digits for 16, 24, 32 or 40 Kbit/s bit-rate ADPCM compression, respectively.

The Mu-Law or A-Law PCM input should be expanded to 14-bit uniform PCM prior to ADPCM compression (see Recommendation G.726). This expansion may be done, for example, by first applying the functions `ippsMuLawToLin_8u16s` or `ippsALawToLin_8u16s` which expand 8-bit Mu-Law or A-Law PCM, respectively, into linear 16-bit PCM.

The linear 16-bit PCM input must be shifted two bits to the right (divided by four) to achieve the 14-bit uniform PCM input appropriate for the `ippsEncode_G726` function.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when the len is less or equal to 0.
`IppStsNullPtrErr`	Indicates an error when the *`pEncMem`*, *`pSrc`*, or *`pDst`* pointer is `NULL`.

DecodeGetStateSize_G726

Informative function, returns the number of bytes needed for decoder memory.

```
IppStatus ippsDecodeGetStateSize_G726_8u16s (unsigned int* pDecSize);
```

Arguments

`pDecSize`	Pointer to the output memory size in bytes.

Discussion

The function `ippsDecodeGetStateSize_G726` is declared in `ippsc.h` file. This function reports the amount of memory needed to process the G.726 ADPCM decompression.

Return Value

`ippStsNoErr`	Indicates no error.
`IppStsNullPtrErr`	Indicates an error when the *`pDecSize`* pointer is `NULL`.

DecodeInit_G726

Initializes the memory for the G726 decoder.

```
IppStatus ippsDecodeInit_G726_8u16s (IppsDecoderState_G726_16s* pDecMem,
   IppSpchBitRate rate, IppPCMLaw law);
```

Arguments

pDecMem	Pointer to the input memory buffer of size needed to properly initialize the G.726 decoder.
rate	Decode bit rate of the G.726 decoder, must be one of IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000.
law	Output speech PCM law: must be one of IPP_PCM_MULAW, IPP_PCM_ALAW, or IPP_PCM_LINEAR.

Discussion

The function ippsDecodeInit_G726 is declared in ippsc.h file. This function initializes the memory given by the pointer *pDecMem* to enable ADPCM decompression starting from the reset state.

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error when *rate* is not equal to one of the admissible decoding bit rates: IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000; or *law* is not equal to one of the acceptable output PCM values IPP_PCM_MULAW, IPP_PCM_ALAW, or IPP_PCM_LINEAR.
IppStsNullPtrErr	Indicates an error when the *pDecMem* pointer is NULL.

Decode_G726

Performs decompression of the ADPCM bit-stream.

```
ppStatus ippsDecode_G726_8u16s (IppsDecoderState_G726_16s* pDecMem, const
   Ipp8u *pSrc, Ipp16s *pDst, unsigned int len)
```

Arguments

pDecMem	Pointer to the memory buffer that has been initialized for G.726 ADPCM decoder.
pSrc	Pointer to the input vector that contains two, three, four, or five binary digits per byte for 16, 24, 32, or 40 Kbit/s ADPCM bit-stream, respectively.
pDst	Pointer to the 16-bit linear PCM speech output vector.
len	The length of input/output vectors.

Discussion

The function `ippsDecode_G726` is declared in `ippsc.h` file. This function performs decompression of the ADPCM bit-stream input (Recommendation G.726) into 16-bit linear PCM. The input bit-stream must be ADPCM compressed on the bit rate for which the G.726 decoder was initialized to operate.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsNullPtrErr`	Indicates an error when the *pDecMem*, *pSrc*, or *pDst* pointer is `NULL`.

G.728 Related Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following ITU-T* recommendation G.728 with Annexes I, H.

The list of these functions is given in the following table:

Table 9-8 Intel IPP G.728 Related Functions

Function Base Name	Operation
IIRGetStateSize_G728	Gets the size of IIR state structure to be used.
IIR_Init_G728	Initializes the IIR state structure.
IIR_G728	Applies IIR filter to multiple samples.
SynthesisFilterGetStateSize_G728	Gets the size of synthesis filter state structure.
SynthesisFilterInit_G728	Initializes the synthesis filter state structure.
SyntesisFilter_G728	Applies the synthesis filter to multiple samples.
CombinedFilterGetStateSize_G728	Gets the size of combined filter state structure.
CombinedFilterInit_G728	Initializes the combined filter state structure.
CombinedFilter_G728	Applies the combined filter to multiple samples.
PostFilterGetStateSize_G728	Gets the size of the post filter state structure.
PostFilterInit_G728	Initializes the post filter state structure.
PostFilter_G728	Applies the post filter to multiple samples.
WinHybridGetStateSize_G728	Gets the size of hybrid windowing module state structure.
WinHybridInit_G728	Initializes the hybrid windowing module state structure.
WinHybrid_G728	Applies the hybrid windowing.
LevinsonDurbin_G728	Calculates LP coefficients from the autocorrelation coefficients.
CodebookSearch_G728	Searches the codebook for the best code vector.
ImpulseResponseEnergy_G728	Implements shape codebook vector convolution and energy calculation.

IIRGetStateSize_G728

Gets the size of IIR state structure to be used.

```
IppStatus ippsIIR16sGetStateSize_G728_16s (int *pSize);
```

Arguments

pSize	Pointer to the output IIR state size value.

Discussion

The function ippsIIR16sGetStateSize_G728 is declared in ippsc.h file. This function returns the minimal size of memory to be allocated for proper use of the IIR filter.

Return Value

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

IIR_Init_G728

Initializes the IIR state structure.

```
IppStatus ippsIIR16sInit_G728_16s (IppsIIRState16s_G728_16s *pMem );
```

Arguments

pMem	Pointer to the memory allocated for IIR filter.

Discussion

The function ippsIIR16sInit_G728 is declared in ippsc.h file. This function initializes the IIR state structure using the given memory block.

Return Value

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

IIR_G728

Applies IIR filter to multiple samples.

```
IppStatus ippsIIR16s_G728_16s (const Ipp16s *pCoeffs,
   const Ipp16s * pSrcQntSpeech, Ipp16s * pDstWgtSpeech, int len,
   IppsIIRState16s_G728_16s *pMem );
```

Arguments

pCoeffs	Pointer to the filter coefficients vector [20]: b_0 , …, b_9 , a_0 , …, a_9 (in Q14).
pSrcQntSpeech	Pointer to the source vector [*len*].
pDstWgtSpeech	Pointer to the destination vector [*len*].
len	The number of source and destination samples.
pMem	Pointer to the IIR filter state structure.

Discussion

The function `ippsIIRState16s_G728` is declared in `ippsc.h` file. This function calculates the synthesized speech output by filtering the input quantized speech through the IIR filter one at a time according to the transfer function:

$$\frac{1+\sum_{i=0}^{9} b_i \cdot z^{-i}}{1+\sum_{i=0}^{9} a_i \cdot z^{-i}}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the *pCoeffs*, *pSrcQntSpeech* , *pDstWgtSpeech* or *pMem* pointers is NULL.

SynthesisFilterGetStateSize_G728

Gets the size of synthesis filter state structure.

```
IppStatus ippSynthesisFilterGetStateSize_G728_16s (int *pSize);
```

Arguments

pSize	Pointer to the output size value of the synthesis filter state structure.

Discussion

The function ippSynthesisFilterGetStateSize_G728 is declared in ippsc.h file. This function returns the minimal size of memory to be allocated for proper use of the synthesis filter.

Return Value

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

SynthesisFilterInit_G728

Initializes the synthesis filter state structure.

```
IppStatus ippSynthesisFilterInit_G728_16s (IppsSynthesisFilterState_G728_16s
   *pMem);
```

Arguments

pMem	Pointer to the memory allocated for synthesis filter.

Discussion

The function ippSynthesisFilterInit_G728 is declared in ippsc.h file. This function initializes synthesis filter state structure using the given memory block.

Return Value

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

SyntesisFilter_G728

Applies the synthesis filter to multiple samples.

```
IppStatus ippsSyntesisFilterZeroInput_G728_16s (const Ipp16s* pCoeffs, Ipp16s*
   pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech, Ipp16s* pSpeechSfs,
   IppsSynthesisFilterState_G728_16s *pMem);
```

Arguments

pCoeffs	Pointer to the filter coefficients vector [51]: a_0 , …, a_{50} in Q14.
pSrcDstExc	Pointer to the input/output gain-scaled excitation vector [5].
excSfs	The input scale of the previous gain-scaled excitation vector.
pDstSpeech	Pointer to the output quantized speech vector [5].
pSpeechSfs	The output scale of the quantized speech vector.
pMem	Pointer to the synthesis filter state structure.

Discussion

The function ippsSyntesisFilterZeroInput_G728 is declared in ippsc.h file. This function computes the decoded speech vector as the sum of the zero-input response and the zero-state response of the synthesis filter according to the transfer function:

$$\frac{1}{1+\sum_{i=1}^{50} a_i \cdot z^{-i}}$$

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the *pCoeffs*, *pSrcDstExc*, *pDstSpeech*, *pSpeechSfs* or *pMem* pointer is NULL.

CombinedFilterGetStateSize_G728

Gets the size of combined filter state structure.

```
IppStatus ippsCombinedFilterGetStateSize_G728_16s (int *pSize);
```

Arguments

pSize	Pointer to the output size value of the combined filter state structure.

Discussion

The function ippsCombinedFilterGetStateSize_G728 is declared in ippsc.h file. This function returns the minimal size of memory to be allocated for proper use of the combined filter.

Return Value

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSize* pointer is NULL.

CombinedFilterInit_G728

Initializes the combined filter state structure.

```
IppStatus ippCombinedFilterInit_G728_16s (IppsCombinedFilterState_G728_16s
   *pMem);
```

Arguments

pMem	Pointer to the memory allocated for the combined filter.

Discussion

The function ippCombinedFilterInit_G728 is declared in ippsc.h file. This function initializes combined filter state structure using the given memory block.

Return Value

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

CombinedFilter_G728

Applies the combined filter to multiple samples.

```
IppStatus ippsCombinedFilterZeroInput_G728_16s(const Ipp16s* pSyntCoeff, const
   Ipp16s* pWgtCoeff, Ipp16s* pDstWgtZIR, IppsCombinedFilterState_G728_16s*
   pMem);
IppStatus ippsCombinedFilterZeroState_G728_16s(const Ipp16s* pSyntCoeff, const
   Ipp16s* pWgtCoeff, Ipp16s* pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech,
   Ipp16s* pSpeechSfs, IppsCombinedFilterState_G728_16s* pMem);
```

Arguments

pSyntCoeff	Pointer to the filter coefficients vector [50]: a_1 , …, a_{50} in Q14.
pWgtCoeff	Pointer to the filter coefficients vector [20]: B_1 , …, B_{10} , A_1 , …, A_{10} in Q14.
pSrcDstExc	Pointer to the output gain-scaled excitation vector [5].
excSfs	The input scale of the previous gain-scaled excitation vector.
pDstWgtZIR	Pointer to the output zero input response vector [5] of the combined filter.
pDstSpeech	Pointer to the output quantized speech vector [5].
pSpeechSfs	The output scale of the quantized speech vector.
pMem	Pointer to the combined filter state structure.

Discussion

The functions `ippsCombinedFilterZeroInput_G728` and `ippsCombinedFilterZeroState_G728` are declared in `ippsc.h` file.

`ippsCombinedFilterZeroInput_G728_16s`. This function calculates the zero-input response of the combined filter by superposing two filters, specifically, the 50s-order synthesis filter and the 10s-order IIR filter according to the transfer function

$$\frac{1}{1+\sum_{i=1}^{50} a_i \cdot z^{-i}} \cdot \frac{1+\sum_{i=1}^{10} B_i \cdot z^{-i}}{1+\sum_{i=1}^{10} A_i \cdot z^{-i}}$$

`ippsCombinedFilterZeroState_G728_16s`. This function first performs filtering of the gain-scaled excitation vector through the zero-state combined filter (see the above function). The memory of combined filter is then updated by adding zero-state responses of the synthesis and the IIR filters which it is combined of. The quantized speech output vector is obtained as a by-product of the memory updates.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSyntCoeff`*, *`pWgtCoeff`*, *`pSrcDstExc`*, *`pDstWgtZIR`*, *`pDstSpeech`*, *`pSpeechSfs`* or *`pMem`* pointer is `NULL`.

PostFilterGetStateSize_G728

Gets the size of the post filter state structure.

```
IppStatus ippsPostFilterGetStateSize_G728_16s (int *pSize);
```

Arguments

`pSize`	Pointer to the output size value of the post filter state structure.

Discussion

The function `ippsPostFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the post filter.

Return Value

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

PostFilterInit_G728

Initializes the post filter state structure.

```
IppStatus ippPostFilterInit_G728_16s (IppsPostFilterState_G728_16s *pMem);
```

Arguments

pMem	Pointer to the memory allocated for the post filter.

Discussion

The function ippPostFilterInit_G728 is declared in ippsc.h file. This function initializes the post filter state structure using the given memory block.

Return Value

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

PostFilter_G728

Applies the post filter to multiple samples.

```
IppStatus ippsPostFilter_G728_16s (Ipp16s gl, Ipp16s glb, Ipp16s kp, Ipp16s
   tiltz, const Ipp16s *pCoeffs, const Ipp16s *pSrc, Ipp16s *pDst,
   IppsPostFilterState_G728_16s *pMem);
```

Arguments

gl	The LTP scaling factor.
glb	The LTP product term.
kp	The LTP lag, pitch period of the current frame.
tiltz	The STP tilt-compensation coefficient.
pCoeffs	Pointer to the post filter coefficients vector [20]: B_1 , …, B_{10} , A_1 , …, A_{10}.
pSrc	Pointer to the input speech vector [5]; elements -*kp*,…,-1 must be given as memory of the LTP filter.
pDst	Pointer to the output post-filtered speech vector [5].
pMem	Pointer to the post filter state structure.

Discussion

The function `ippsPostFilter_G728` is declared in `ippsc.h` file. This function performs filtering of input samples by one at a time according to the transfer function that is comprised of LTP filter and STP filter parts:

$$gl \cdot (1 - \mathrm{glb} \cdot z^{-(p)}) \cdot \frac{1 - \sum_{i=1}^{10} B_i \cdot z^{-i}}{1 - \sum_{i=1}^{10} A_i \cdot z^{-i}} \cdot (1 + tiltz \cdot z^{-1})$$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pCoeffs`*, *`pSrc`*, *`pDst`* or *`pMem`* pointer is `NULL`.

WinHybridGetStateSize_G728

Gets the size of hybrid windowing module state structure.

```
IppStatus ippsWinHybridGetStateSize_G728_16s (int M, int L, int N, int DIM, int
   *pSize);
```

Arguments

`M`	The input length of the LPC window.
`L`	The input adaptation cycle size in samples.
`N`	The input number of non-recursive window samples.
`DIM`	The input block size used for block scaling of the input speech by the `ippsWinHybrid_G728` function.
`pSize`	Pointer to the output size value of the hybrid windowing module state structure.

Discussion

The function ippsWinHybridGetStateSize_G728 is declared in ippsc.h file. This function returns the minimal size of memory to be allocated for proper use of the hybrid windowing module according to the given window parameters.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSize* pointer is NULL.
ippStsSizeErr	Indicates an error when *L*, *M* or *N* are less than or equal to zero.

WinHybridInit_G728

Initializes the hybrid windowing module state structure.

```
IppStatus ippWinHybridInit_G728_16s (const Ipp16s *pWinTab, int M, int L,
   int N, int DIM, Ipp16s a2L, IppsWinHybridState_G728_16s *pMem);
```

Arguments

pWinTab	The input vector of windowing coefficients.
M	The input length of LPC window. Must be not less than 10.
L	The input adaptation cycle size in samples.
N	The input number of non-recursive window samples.
a2L	The a^{2L} multiple used in calculation of the recursive component in hybrid windowing module.
N	The input number of non-recursive window samples.

Discussion

The function ippWinHybridInit_G728 is declared in ippsc.h file. This function initializes the hybrid windowing module state structure using the given memory block. The recursive component and the previous speech samples are zeroed.

Return Value

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

WinHybrid_G728

Applies the hybrid windowing.

```
IppStatus ippsWinHybridBlock_G728_16s(Ipp16s bfi, const Ipp16s *pSrc, const
   Ipp16s *pSrcSfs, Ipp16s *pDst, IppsWinHybridState_G728_16s *pMem);
IppStatus ippsWinHybrid_G728_16s(Ipp16s bfi, const Ipp16s *pSrc, const
   Ipp16s *pSrcSfs, Ipp16s *pDst, IppsWinHybridState_G728_16s *pMem);
```

Arguments

bfi	The input bad frame indicator: “1” signifies a bad frame, any other value signifies a good frame.
pSrc	Pointer to the input speech vector [20].
pSrcSfs	Pointer to the input vector [5] of scale factors for elements of the input speech vector. Each element of the *pSrcSfs* vector may specify a separate scale factor for a block of input speech vector elements. Thus, *pSrcSfs*[0] is the scale factor of the elements *pSrc*[0],..*pSrc*[*DIM*-1]; *pSrcSfs*[1] is the scale factor for *pSrc*[*DIM*],..*pSrc*[2**DIM*-1], and so on, where *DIM* is a block length that must be defined by the WinHybridInit_G728() function.
pDst	Pointer to the output reflection coefficients vector [M+1], where M is the length of the LPC window defined in WinHybridInit_G728() function.
pMem	Pointer to the post filter state structure.

Discussion

The function `ippsWinHybrid_G728` is declared in `ippsc.h` file. This function first applies the window to the input speech vector and then calculates the autocorrelation coefficients needed in LPC analysis by the formulae:

$$R_m(i) = r_m(i) + \sum_{k=m-N}^{m-1} s_m(k)s_m(k-i), m = 0,..,M$$

where the recursive component of adaptation cycle is calculated as follows:

$$r_m(i) = \alpha^{2L} \cdot r_{m-L}(i) + \sum_{k=m-L-N}^{m-N-1} s_m(k)s_m(k-i)$$

The recursive part is calculated using the data calculated in previous adaptation cycle and stored in module memory.

A *white noise correction* is applied to autocorrelation coefficients by increasing the energy as follows:

$$r_m(0) = \frac{257}{256} \cdot r_m(0)$$

The recursive component and previous speech samples are stored in the module memory and may be used in the next adaptation cycle.

If the bad frame indicator is on, then only 10 autocorrelation coefficients are calculated and output.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrcSfs`*, *`pSrc`*, *`pDst`* or *`pMem`* pointer is `NULL`.

LevinsonDurbin_G728

Calculates LP coefficients from the autocorrelation coefficients.

```
IppStatus ippsLevinsonDurbin_G728_16s_Sfs(const Ipp16s *pSrcAutoCorr, int
   order, Ipp16s *pDstLPC, Ipp16s *pDstResidualEnergy, Ipp16s
   *pDstScaleFactor);
IppStatus ippsLevinsonDurbin_G728_16s_ISfs(const Ipp16s *pSrcAutoCorr, int
   numSrcLPC, int order, Ipp16s *pSrcDstLPC, Ipp16s *pSrcDstResidualEnergy,
   Ipp16s *pSrcDstScaleFactor);
```

Arguments

pSrcAutoCorr	Pointer to the input autocorrelation coefficients vector [*order*+1].
order	The input number of LP coefficients to calculate.
numSrcLPC	The input number of pre-calculated LPCs.
pDstLPC	Pointer to the output LPC vector [*order*].
pSrcDstLPC	Pointer to the input/output LPC vector [*order*].
pDstResidualEnergy	Pointer to the output residual energy.
pSrcDstResidualEnergy	Pointer to the input/output residual energy of the pre-calculated LPC.
pDstScaleFactor	Pointer to the output scale factor of the LPC vector.
pSrcDstScaleFactor	Pointer to the input scale factor of the pre-calculated LPC and the output scale factor of the output LPC vector.

Discussion

The functions `ippsLevinsonDurbin_G728_16s_Sfs` and `ippsLevinsonDurbin_G728_16s_ISfs` are declared in `ippsc.h` file.

ippsLevinsonDurbin_G728_16s_Sfs. This function may be used to calculate Linear Prediction (LP) coefficients by solving the following set of linear equations:

$$\sum_{i=0}^{order-1} a_i \cdot r(|i-k|) = r(k), k = 1,2,\ldots,order$$

where a_i , $i = 0,1,\ldots,order-1$ are the LP coefficients to be calculated and stored in the output LPC vector. The description of the Levinson-Durbin algorithm used by this function may be found in `ippsLevinsonDurbin_G729` function.

Both `ippsLevinsonDurbin_G728` and `ippsLevinsonDurbin_G729B` calculate mathematically the same, but not bit exact LPCs. The difference is that the `ippsLevinsonDurbin_G729B` function outputs LPCs in Q12, while the `ippsLevinsonDurbin_G728` function automatically rescales the LPCs if overflow occurs.

ippsLevinsonDurbin_G728_16s_Isfs. This function may be used to continue Levinson-Durbin recursion for bigger *order*. The LPCs, their scale factor and the residual energy of the previous recursion and the additional autocorrelation coefficients are used to resume recursion.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the input or output pointers is `NULL`.
`ippStsRangeErr`	Indicates an error when *order* is less than or equal to 0, or when *order* < *numSrcLPC* for the in-place function.

CodebookSearch_G728

Searches the codebook for the best code vector.

```
IppStatus ippsCodebookSearch_G728_16s(const Ipp16s* pSrcCorr, const Ipp16s*
   pSrcEnergy, int* pDstShapeIdx, int* pDstGainIdx, short* pDstCodebookIdx,
   IppSpchBitRate rate);
```

Arguments

pSrcCorr	Pointer to the input vector [5] of time-reversed convolution of the target signal.
pSrcEnergy	Pointer to the input vector [128] of the energy of convolved shape codevector.
pDstShapeIdx	Pointer to the output best 7-bit shape codebook index.
pDstGainIdx	Pointer to the output best 3-bit gain codebook index.
pDstCodebookIdx	Pointer to the output best codebook index to be transmitted.
rate	Input coding bit rate.

Discussion

The function ippsCodebookSearch_G728 is declared in ippsc.h file. This function implements the “Error calculator and best codebook index selector” block used to search through the gain codebook and the shape codebook for the best combination of the gain and shape codebook indexes.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcCorr*, *pSrcEnergy* , *pDstShapeIdx*, *pDstGainIdx* or *pDstCodebookIdx* pointer is NULL.
IppStsRangeErr	Indicates an error when *rate* is not one of the acceptable values IPP_SPCHBR_16000, IPP_SPCHBR_12800 or IPP_SPCHBR_9600 for the 16, 12.8 or 9.6 Kbit/s coding bit rates, respectively.

ImpulseResponseEnergy_G728

Implements shape codebook vector convolution and energy calculation.

```
IppStatus ippsImpulseResponseEnergy_G728_16s(const Ipp16s *pSrcImpResp, Ipp16s
   *pDstEnergy);
```

Arguments

pSrcImpResp	Pointer to the input impulse response vector [5] of the synthesis and weighted filter.
pDstEnergy	Pointer to the output energy vector [128] of the convolved shape codevector.

Discussion

The function ippsImpulseResponseEnergy_G728 is declared in ippsc.h file. This function implements the “Shape codevector convolution and energy calculator” block used to calculate the energy of the convolved shape codevector.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcImpResp* or *pDstEnergy* pointer is NULL.

Audio Coding Functions 10

The subset of Intel® IPP for audio coding includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio encoder and decoder (see [ISO14496]), MP3 encoder and decoder. These functions implement pipeline blocks with large computational complexity.

The current set of functions is sufficient to implement a portable optimized MPEG-4 AAC Main profile decoder and a portable optimized MPEG-1, 2 Layer III encoder (see [ISO11172] and [ISO13818]).

Also, this chapter includes description of companding functions that perform the μ-law and A-law companding in compliance with the CCITT G.711 specification (see [CCITT]).

The full list of functions is given in Table 10-1.

Table 10-1 Intel IPP Audio Coding functions

Function Base Name	Operation
Interleaved to Multi-row Format Conversion Functions	
`Interleave`	Converts signal from non-interleaved to interleaved format.
`Deinterleave`	Converts signal from interleaved to non-interleaved format.
Spectral Data Prequantization Functions	
Pow34	Raises a vector to the power of 3/4
`Pow43`	Raises a vector to the power of 4/3
Scale Factors Calculation Functions	
`CalcSF`	Restores actual scale factors from the bit stream values
Mantissa Conversion and Scaling Functions	

Table 10-1 Intel IPP Audio Coding functions (continued)

Function Base Name	Operation
ApplySF_I	Applies scale factors to spectral bands in accordance with spectral bands boundaries
MakeFloat	Converts mantissa and exponent arrays to float arrays
Modified Discrete Cosine Transform Functions	
MDCTFwdInitAlloc, MDCTInvInitAlloc	Initializes modified discrete cosine transform specification structure
MDCTFwdFree, MDCTInvFree	Closes modified discrete cosine transform specification structure
MDCTFwdGetBufSize, MDCTInvGetBufSize	Gets the size of MDCT work buffer
MDCTFwd, MDCTInv	Computes forward or inverse modified discrete cosine transform (MDCT) of a signal
Block Filtering Functions	
FIRBlockInitAlloc	Initializes FIR block filter state
FIRBlockFree	Closes FIR block filter state
FIRBlockOne	Filters vector of sample through FIR block filter
Frequency Domain Prediction Functions	
FDPInitAlloc	Initializes predictor state
FDPFree	Closes FDP state
ResetFDP	Resets predictors for all spectral lines
ResetFDP_SFB	Resets predictor-specific information in some scale factor bands
ResetFDPGroup	Resets predictors for group of spectral lines
FDPFwd	Performs frequency domain prediction procedure and calculates prediction error
FDPInv	Retrieves input signal from prediction error, using frequency domain prediction procedure
Huffman Algorithm Functions	
GetSizeHDT	Calculates size of the Huffman Decode table
BuildHDT	Builds Huffman decode table
DecodeVLC	Decodes value from Huffman table
GetSizeHET	Calculates size required for input Huffman table

Table 10-1 Intel IPP Audio Coding functions (continued)

Function Base Name	Operation
BuildHET	Builds Huffman table
HuffmanCountBits	Calculates size required for encoding quantized values
EncodeVLC	Encodes block of spectral values by using the specified Huffman table.
GetSizeHET_VLC	Calculates the size necessary for the input Huffman table in an internal format.
BuildHET_VLC	Builds a Huffman table in the internal format.
CountBits	Calculates the size in bits necessary to encode the input array.
EncodeBlock	Encodes input array.
Vector Quantization Functions	
CdbkInitAlloc	Initializes the codebook structure.
CdbkFree	Closes the IppsCdbkState_VQ_32f structure created in CdbkInitAlloc.
PreSelect_VQ	Selects candidates for the nearest code vector of codebooks.
MainSelect_VQ	Finds optimal indexes with minimal distortion.
IndexSelect_VQ	Finds optimal vector set for specified number of codebooks.
VectorReconstruction_VQ	Reconstructs vectors from indexes.
Companding Functions	
MuLawToLin	Decodes samples from 8-bit μ-law encoded format to linear samples.
LinToMuLaw	Encodes the linear samples using 8-bit μ-law format and stores them in a vector.
ALawToLin	Decodes the 8-bit A-law encoded samples to linear samples.
LinToALaw	Encodes the linear samples using 8-bit A-law format and stores them in an array.
MuLawToALaw	Converts samples from 8-bit μ-law encoded format to 8-bit A-law encoded format.
ALawToMuLaw	Converts samples from 8-bit A-law encoded format to 8-bit μ-law encoded format.
MP3 Encoder Functions	
AnalysisPQMF_MP3	Implements stage 1 of MP3 hybrid analysis filterbank.

Table 10-1 Intel IPP Audio Coding functions (continued)

Function Base Name	Operation
MDCTFwd_MP3	Implements stage 2 of the MP3 hybrid analysis filterbank.
PsychoacousticModelTwo_MP3	Implements the ISO/IEC 11172-3 psych-acoustic model recommendation 2 to estimate the masked threshold and perceptual entropy associated with a block of PCM audio input.
JointStereoEncode_MP3	Transforms independent left and right channel spectral coefficient vectors into combined mid/side (MS) and/or intensity (IS) mode coefficient vectors suitable for quantization.
Quantize_MP3	Quantizes the spectral coefficients generated by the analysis filterbank.
PackScalefactors_MP3	Applies noiseless coding to the scale factors and then packs the output into the bitstream buffer.
HuffmanEncode_MP3	Applies lossless Huffman encoding to the quantized samples and packs the output into the bitstream buffer.
PackFrameHeader_MP3	Packs the content of the frame header into the bitstream.
PackSideInfo_MP3	Packs the side information into the bitstream buffer.
BitReservoirInit_MP3	Initializes all elements of the bit reservoir state structure.
MP3 Decoder Functions	
UnpackFrameHeader_MP3	Unpacks the audio frame header.
UnpackSideInfo_MP3	Unpacks the side information from the input bitstream for use during the decoding of the associated frame.
UnpackScaleFactors_MP3	Unpacks scale factors.
HuffmanDecode_MP3 HuffmanDecodeSfb_MP3 HuffmanDecodeSfbMbp_MP3	Decodes Huffman data.
ReQuantize_MP3 ReQuantizeSfb_MP3	Requantizes the decoded Huffman symbols.
MDCTInv_MP3	Performs the first stage of hybrid synthesis filter bank.
SynthPQMF_MP3	Performs the second stage of hybrid synthesis filter bank.
MPEG-2 Primitive Functions	
UnpackADIFHeader_AAC	Gets the AAC ADIF format header.
UnpackADTSFrameHeader_AAC	Gets ADTS frame header from the input bitstream.

Table 10-1 Intel IPP Audio Coding functions (continued)

Function Base Name	Operation
DecodePrgCfgElt_AAC	Gets program configuration element from the input bitstream.
DecodeChanPairElt_AAC	*Gets channel_pair_element from the input bitstream.*
NoiselessDecoder_LC_AAC	Decodes all the data for one channel.
DecodeDatStrElt_AAC	Gets data stream element from the input bitstream.
DecodeFillElt_AAC	Gets the fill element from the input bitstream.
QuantInv_AAC	Performs inverse quantization of Huffman symbols for current channel in-place.
DecodeMsStereo_AAC	Processes MS stereo for pair channels in-place.
DecodeIsStereo_AAC	Processes intensity stereo for pair channels.
DeinterleaveSpectrum_AAC	Deinterleaves the coefficients for short block.
DecodeTNS_AAC	Decodes for Temporal Noise Shaping in-place.
MDCTInv_AAC	Maps the time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples.
MPEG-4 Primitive Functions	
DecodeMainHeader_AAC	Gets main header information and main layer information from bit stream.
DecodeExtensionHeader_AAC	Get extension header information and extension layer information from bit stream.
DecodePNS_AAC	Implements perceptual noise substitution coding within an ICS.
LongTermReconstruct_AAC	Uses Long Term Reconstruct to reduce the redundancy of a signal between successive coding frames.
MDCTFwd_AAC	Generates spectrum coefficient of PCM samples.
EncodeTNS_AAC	Performs reversion of TNS in the Long Term Reconstruct loop in-place.
LongTermPredict_AAC	Gets the predicted time domain signals in the Long Term Reconstruct (LTP) loop.
NoiseLessDecode_AAC	Performs noiseless decoding.
LtpUpdate_AAC	Performs required buffer update in the Long Term Reconstruct (LTP) loop.

Interleaved to Multi-row Format Conversion Functions

This section describes functions that enable you to perform transformations between interleaved and non-interleaved forms of multichannel signal. A signal in the interleaved form is a vector with interleaved samples for different channels. In the non-interleaved form, samples for each channel are stored in a separate vector.

NOTE. *Although you may use both aligned and unaligned memory for arrays, you should expect slower performance when memory is not aligned.*

Interleave

Converts signal from non-interleaved to interleaved format.

```
IppStatus ippsInterleave_16s(const Ipp16s** pSrc, int ch_num, int len,
    Ipp16s* pDst);
IppStatus ippsInterleave_32f(const Ipp32f** pSrc, int ch_num, int len,
    Ipp32f* pDst);
```

Arguments

pSrc	Array of pointers to the vectors [*len*] containing samples for particular channels.
ch_num	Number of channels.
len	Number of samples in each channel.
pDst	Pointer to the destination vector [*ch_num* * *len*] in interleaved format.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsInterleave` transforms the signal from the non-interleaved to interleaved format according to the formula

$pDst[i] = pSrc[i \text{ div } ch\_num][i \bmod ch\_num], 0 \le i < ch\_num * len,$

where `div` is the integer part of the quotient and `mod` is the remainder.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* or *ch_num* is less than or equal to 0.
`ippStsMisalignedBuf`	Indicates misaligned data. Supply aligned data for better performance.

Deinterleave

Converts signal from interleaved to non-interleaved format.

```
IppStatus ippsDeinterleave_16s(const Ipp16s* pSrc, int ch_num, int len,
    Ipp16s** pDst);
IppStatus ippsDeinterleave_32f(const Ipp32f* pSrc, int ch_num, int len,
    Ipp32f** pDst);
```

Arguments

pSrc	Pointer to vector [*ch_num* * *len*] of interleaved samples.
ch_num	Number of channels.
len	Number of samples in each channel.
pDst	Array of pointers to the vectors [*len*] to be filled with samples of particular channels.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsDeinterleave` transforms the input signal from interleaved to non-interleaved format according to the formula

$pDst[i][j] = pSrc[i + j * ch\_num], 0 \le i < ch\_num, 0 \le j < len.$

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* or *ch_num* is less than or equal to 0.
`ippStsMisalignedBuf`	Indicates misaligned data. Supply aligned data for better performance.

Spectral Data Prequantization Functions

MPEG-1, 2 Layer III and MPEG-4 AAC audio encoders raise the spectral data to the power of 3/4 before quantization to provide a more consistent signal-to-noise ratio over the range of quantized values. The re-quantizers in the decoders linearize the values by raising their output to the power of 4/3.

Pow34

Raises a vector to the power of 3/4.

```
IppStatus ippsPow34_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int len);
IppStatus ippsPow34_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the input data vector [*len*].
pDst	Pointer to the output data vector [*len*].

len	Number of elements in the input and output vectors.

Discussion

This function is declared in the ippac.h header file. The function ippsPow34 performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = |pSrc[i]|^{\frac{3}{4}}, 0 \le i < len$$

and stores the result in *pDst*.

For example, in MPEG-1 Layer III the quantization of the complete vector of spectral values is done according to the following formula:

$$ix(i) = nint\left(\left(\frac{|xr(i)|}{\sqrt[4]{2^{qquant + quantanf}}}\right)^{0.75} - 0.0946\right), \text{ where}$$

ix is the array of quantized values, *i* is the number of values in the array, *nint* is the function used to round non-integer values to the nearest integer value, *xr* is the vector of the magnitudes of the spectral values, *qquant* is the quantizer step size information, and *quantanf* is a constant that depends on the spectral flatness measure.

You can use the function ippsPow34_32f16s for this operation. However, if the maximum of all quantized values is outside the table range or the overall bit sum exceeds the available bits, you should repeat this operation several times with a new *qquant* value and the same *xr* array.

Alternatively, you can use the function ippsPow34_32f to calculate the power of 3/4 for *xr* only once before the inner iteration loop and use it during every quantization iteration with the constant multiplier and convert the resulting values to short.

$$multiplier = \left(\frac{1}{\sqrt[4]{2^{qquant + quantanf}}}\right)^{0.75}.$$

This operation is supported by the function ippsMulC_Low_32f16s.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.

ippStsMisalignedBuf	Indicates misaligned data. Supply aligned data for better performance.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsNanArg	Indicates a warning when NaN is encountered in the input data vector.
ippStsOutOfRangeErr	Indicates an error when the value of any element of the source array is more than 1 000 000.

Pow43

Raises a vector to the power of 4/3.

```
IppStatus ippsPow43_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the input data vector [*len*].
pDst	Pointer to the output data vector [*len*].
len	Number of elements in the input and output vectors.

Discussion

This function is declared in the ippac.h header file. The function ippsPow43 performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = sign(pSrc[i]) * |pSrc[i]|^{\frac{4}{3}}, 0 \le i < len$$

and stores the result in *pDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.

`ippStsMisalignedBuf`	Indicates misaligned data. Supply aligned data for better performance.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

NOTE. *The input values in the `pSrc` array should not be greater than 8206. The function does not check if the input values meet this requirement in order not to cause performance slowdown. If the input values in the array exceed the limit, the function may operate incorrectly. No error code is returned in this case.*

Scale Factors Calculation Functions

In MPEG-2, 4 GA AAC decoder scale factors extracted from the bitstream require an additional restoring procedure. The function `CalcSF` restores actual scale factors from values transmitted in the bit stream.

CalcSF

Restores actual scale factors from the bit stream values.

```
IppStatus ippsCalcSF_16s32f(const Ipp16s* pSrc, int offset, Ipp32s* pDst,
    int len);
```

Arguments

pSrc	Pointer to the input data array.
offset	Scale factors offset.
pDst	Pointer to the output data array.
len	Number of elements in the vector.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsCalcSF` restores actual scale factors from the values *pSrc* transmitted in the bit stream, using the common scale factor offset. Computation is performed according to the following formula

$$pDst[i] = 2^{\frac{1}{4}(pSrc[i]-offset)}, 0 \le i < len.$$

Restored scale factors are written into *pDst*.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pDst* pointer is `NULL`.

`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.

NOTE. *The input values in the `pSrc` array should be in the range $|pSrc[i] - offset| \le 128$. Otherwise the function may operate incorrectly. No error message is returned in this case.*

Mantissa Conversion and Scaling Functions

The scale application procedure is necessary for MPEG-2, 4 GA AAC decoder to restore the original spectral values from a set of the inversely quantized spectral coefficients. The whole spectrum is divided into a set of scale factor bands. Since the widths differ from band to band, the band positions are defined by the offset vector.

The mantissa conversion procedure is necessary for the AC3 decoder to restore the original spectral values from a set of the exponents and mantissas in the bitstream.

ApplySF_I

Applies scale factors to spectral bands in accordance with spectral bands boundaries.

```
IppStatus ippsApplySF_32f_I(Ipp32f* pSrcDst, const Ipp32f* pSF,
    const int *pBandOffset, int bands_number);
```

Arguments

`pSrcDst`	Pointer to the input and output data array. The size of array must be not less than *`pBandOffset`*`[`*`bands_number`*`]`.
`pSF`	Pointer to the data array containing scale factors. The size of the array must be not less than *`bands_number`*.

pBandsOffset	Pointer to the vector of band offsets. The size of array must be not less than *bands_number* + 1.
bands_number	Number of bands to which scale factors are applied.

Discussion

This function is declared in the ippac.h header file. The function ippsApplySF_I computes scaled values from the input vector, the vector of scale factors, and the vector of band offsets. Operations are performed in-place.

This function applies the set of scale factors *pSF* with *bands_number* elements to the bands constructed from the input vector *pSrc*. Band boundaries are defined by the vector of the band offsets *pBandOffset*. All values in each band are multiplied by the corresponding scale factor.

NOTE. *The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of pSrcDst vector is contained in the pBandsOffset[bands_number] element.*

Return Value

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

MakeFloat

Converts mantissa and exponent arrays to float arrays.

```
IppStatus ippsMakeFloat_16s32f (Ipp32s* inmant, Ipp32s* inexp, Ipp32s size,
    Ipp32f* outfloat);
```

Arguments

inmant	Array of mantissas.
inexp	Array of exponents.
size	Number of array elements.
outfloat	Array of resulting float arrays.

Discussion

This function is declared in the `ippac.h` header file. This function converts the mantissa and exponent arrays decoded from the bitstream to the float array of spectral samples by the formula:

$outfloat[i] = inmant[i] \times 2^{-in\exp[i]-15}$,
where $i = 0...size$.

The conversion serves to improve application performance when decoding bitstreams in the AC3 format.

Return Value

`ippStsNoErr`	Indicates no error.

Modified Discrete Cosine Transform Functions

This section describes Intel IPP functions that compute the modified discrete cosine transform (MDCT) of a signal. MDCT is a lapped orthogonal transform widely used in different audio codecs, such as MPEG-1, MPEG-2, AC-3, and AAC.

MDCTFwdInitAlloc, MDCTInvInitAlloc

Initializes modified discrete cosine transform specification structure.

```
IppStatus ippsMDCTFwdInitAlloc_32f(IppsMDCTFwdSpec_32f** pMDCTSpec,
    int length);
IppStatus ippsMDCTInvInitAlloc_32f(IppsMDCTInvSpec_32f** pMDCTSpec,
    int length);
```

Arguments

pMDCTSpec	Pointer to MDCT specification structure to be created.
length	Number of samples in MDCT. Since this set of functions was designed specially for audio coding, only the following values of length are supported: 12, 36, and 2^k, where $k \geq 5$. These values are the only values that appear in audio coding.

Discussion

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdInitAlloc` and `ippsMDCTInvInitAlloc` create and initialize MDCT specification structure *pMDCTSpec* with the specified transform length *length*.

`ippsMDCTFwdInitAlloc`. The function `ippsMDCTFwdInitAlloc` initializes the forward MDCT specification structure.

`ippsMDCTInvInitAlloc`. The function `ippsMDCTInvInitAlloc` initializes the inverse MDCT specification structure.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMDCTSpec* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *length* does not belong to the above set of admissible values.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

MDCTFwdFree, MDCTInvFree

Closes modified discrete cosine transform specification structure.

```
IppStatus ippsMDCTFwdFree_32f(IppsMDCTFwdSpec_32f* pMDCTSpec);
IppStatus ippsMDCTInvFree_32f(IppsMDCTInvSpec_32f* pMDCTSpec);
```

Arguments

pMDCTSpec	Pointer to the MDCT specification structure to be closed.

Discussion

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdFree` and `ippsMDCTInvFree` close the MDCT structure *pMDCTSpec* by freeing all memory associated with the specification created by `ippsMDCTFwdInitAlloc` or `ippsMDCTInvInitAlloc` functions.

Call either `ippsMDCTFwdFree` or `ippsMDCTInvFree` after the transform is completed.

ippsMDCTFwdFree. The function `ippsMDCTFwdFree` closes the forward MDCT specification structure.

ippsMDCTInvFree. The function `ippsMDCTInvFree` closes the inverse MDCT specification structure.

Return Value

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

MDCTFwdGetBufSize, MDCTInvGetBufSize

Gets the size of MDCT work buffer.

```
IppStatus ippsMDCTFwdGetBufSize_32f(const IppsMDCTFwdSpec_32f* pMDCTSpec, int*
    pSize);
IppStatus ippsMDCTInvGetBufSize_32f(const IppsMDCTInvSpec_32f* pMDCTSpec, int*
    pSize);
```

Arguments

pMDCTSpec	Pointer to the MDCT specification structure.
pSize	Address of the MDCT work buffer size value in bytes.

Discussion

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdGetBufSize` and `ippsMDCTInvGetBufSize` get the work buffer size of the MDCT described by the specification structure *pMDCTSpec* and store the result in *pSize*.

`ippsMDCTFwdGetBufSize`. The function `ippsMDCTFwdGetBufSize` gets the size of the work buffer for the forward MDCT.

`ippsMDCTInvGetBufSize`. The function `ippsMDCTInvGetBufSize` gets the size of the work buffer for the inverse MDCT.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pMDCTSpec* pointer or *pSize* value is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification structure *pMDCTSpec* is invalid.

MDCTFwd, MDCTInv

Computes forward or inverse modified discrete cosine transform (MDCT) of a signal.

```
IppStatus ippsMDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst,const
    IppsMDCTFwdSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst,const
    IppsMDCTInvSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsMDCTFwdSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f_I(Ipp32f* pSrcDst, const IppsMDCTInvSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
```

Arguments

pSrc	Pointer to the input data array.
pDst	Pointer to the output data array.
pSrcDst	Pointer to the input and output data array for the in-place operations.
pMDCTSpec	Pointer to the MDCT specification structure.
pBuffer	Pointer to the MDCT work buffer.

Discussion

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwd` and `ippsMDCTInv` compute the forward and inverse modified discrete cosine transform (MDCT), respectively.

In the following definition of MDCT, N denotes the length and $n_0 = (N/2 + 1)/2$.

For the forward MDCT, $x(n)$ is *pSrc*[*n*] and $y(k)$ is *pDst*[*k*], whereas for the inverse MDCT $x(n)$ is *pDst*[*n*] and $y(k)$ is *pSrc*[*k*].

The forward MDCT is defined by the formula:

$$y(k) = 2 \cdot \sum_{n=0}^{N-1} x(n) \cos\left(\frac{\pi}{N}(n + n_0)(2k + 1)\right), \text{ for } 0 \le k < \frac{N}{2}.$$

The inverse MDCT is defined as

$$x(n) = \frac{2}{N} \cdot \sum_{k=0}^{\frac{N}{2}-1} y(k) \cos\left(\frac{\pi}{N}(n + n_0)(2k + 1)\right), \text{ for } 0 \le n < N.$$

The *pBuffer* argument provides the MDCT functions with the necessary work memory and helps to avoid memory allocation within the functions. The buffer may also increase the performance if the MDCT functions use the result of the previous operation stored in cache as an input array.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pMDCTSpec*, *pBuffer*, *pSrc*, and *pDst* (for ippsMDCTFwd_32f and ippsMDCTInv_32f) or *pSrcDst* (for MDCTFwd_32f_I and MDCTInv_32f_I) pointer is NULL.
ippStsContextMatchErr	Indicates an error when the specification structure *pMDCTSpec* is invalid.
ippStsMisalignedBuf	Indicates misaligned data. Supply aligned data for better performance.

Block Filtering Functions

Intel IPP functions described in this section implement the finite impulse response (FIR) block filter. You can use this group of functions to design transform domain adaptive filters. These filters preprocess the signal by decomposing the input vector into orthogonal components, which are subsequently used as inputs to a parallel bank of adaptive subfilters. You may use this approach for implementing frequency domain linear predictors in audio codecs, for example, CELP and AAC.

The filtering function receives a number of vectors (signals). Every call of a filtering function produces one filtered sample for each input signal. The library functions do not perform any particular adaptation method but you can specify the filter taps at each call of a filtering function.

To use the FIR block filter functions, follow these general steps:

1. Call `FIRBlockInitAlloc` to initialize the state structure of a block filter.
2. Call `FIRBlockOne` to filter a vector of samples through a block filter.
3. Call `FIRBlockFree` to free dynamic memory associated with the FIR block filter.

FIRBlockInitAlloc

Initializes FIR block filter state.

```
IppStatus ippsFIRBlockInitAlloc_32f(IppsFIRBlockState_32f** pState, int order,
    int len);
```

Arguments

pState	Pointer to the FIR block filter state structure to be created.
order	Number of elements in the array containing the tap values.
len	Number of input signals.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockInitAlloc` creates and initializes a FIR block filter state.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are null.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

ippStsFIRLenErr	Indicates an error when *order* or *len* is less than or equal to 0.

FIRBlockFree

Closes FIR block filter state.

```
IppStatus ippsFIRBlockFree_32f(IppsFIRBlockState_32f* pState);
```

Arguments

pState	Pointer to the FIR block filter state structure to be closed.

Discussion

This function is declared in the ippac.h header file. The function ippsFIRBlockFree closes the FIR block filter state by freeing all memory associated with the filter state created by the function ippsFIRBlockInitAlloc.

Call ippsFIRBlockFree after filtering is completed.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are null.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.

FIRBlockOne

Filters vector of samples through FIR block filter.

```
IppStatus ippsFIRBlockOne_32f(Ipp32f* pSrc, Ipp32f* pDst,
    IppsFIRBlockState_32f* pState, Ipp32f *pTaps);
```

Arguments

pSrc	Pointer to the input vector of samples to be filtered.
pDst	Pointer to the vector of filtered output samples.
pState	Pointer to the FIR filter state structure.
pTaps	Pointer to the vector of filter taps.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockOne` filters a vector of samples *pSrc* of the length *len* through a filter and stores the result in *pDst*.

The filter taps are specified in the vector *pTaps* of the length *order*. The values of *len* and *order* parameters are specified in the `FIRBlockInitAlloc` call.

In the following definition of the FIR filter, the sample of the input vector *i* to be filtered with the delay *k* is denoted x^i_{n-k}, and the taps are denoted h_k. The output value y_i is defined by the following formula:

$$y^i_n = \sum_{k=0}^{order-1} h_k x^i_{n-k}, \quad 0 \le i < len .$$

Before calling the function `ippsFIRBlockOne`, initialize the filter state by calling the function `ippsFIRBlockInitAlloc`. Specify the taps values in the argument *pTaps*.

Example 10-1 Single-Rate Filtering with the ippsFIRBlockOne Function

```
IppStatus fir(void)
{
#undef NUMITERS
#define NUMITERS 20
#undef BLOCKSIZE
#define BLOCKSIZE 20
    int n;
    int i;
    IppStatus status;
    IppsFIRBlockState_32f *fctx;
    Ipp32f x [BLOCKSIZE], y [BLOCKSIZE];
        const float taps[] = {
```

Example 10-1 Single-Rate Filtering with the ippsFIRBlockOne Function

```
        0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
            0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0
    };
    ippsFIRBlockInitAlloc_32f( &fctx, 11, BLOCKSIZE );
    for (n =0;n<NUMITERS;++n)
    {
        for (i = 0; i< BLOCKSIZE; i++) x[i]=(Ipp32f)sin(IPP_2PI *
            n *0.2 + i);
        status = ippsFIRBlockOne_32f( x, y, fctx, (Ipp32f*)taps );
        for (i = 0; i< BLOCKSIZE; i++)
            printf("%f", y[i]);
    }
    ippsFIRBlockFree_32f(fctx);
    return status;
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *`pSrc`*, *`pDst`*, *`pState`*, or *`pTaps`* pointer is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.
`ippStsMisalignedBuf`	Indicates misaligned arrays. Supply aligned data for better performance.
`ippStsFIRLenErr`	Indicates an error when *`tapsLen`* is less than or equal to 0.

Frequency Domain Prediction Functions

MPEG-2, 4 AAC encoder uses prediction in frequency domain (FDP) to decrease redundancy in the audio signal and ensure more effective coding. For each spectral line, input signals are filtered through the second order adaptive FIR filter called a predictor. Then, instead of processing the signal, the difference between the original signal and the filter output (that is, the prediction error), is passed for further processing. The decoder derives the original signal from the prediction error using a symmetrical block.

You should regularly reset the predictors to their initial state to reduce accumulated calculation error. You may also need to reset the predictors in special cases discussed and described in the ISO-144963 standard. You can reset predictors for the entire spectrum, several scale factor bands, or a selected group of spectral lines.

For more information on the algorithm of filter coefficient adaptation and FDP usage see ISO-144963, clause 6.5.3.2.

To use the FDP prediction tool functions described in this section, follow these steps:

1. Call the function `FDPInitAlloc` to allocate memory and initialize predictor state.
2. Call the function `FDPFwd` for each frame to calculate prediction error or call the function `FDPInv` to retrieve the original signal.
3. Call the functions `ResetFDP`, `ResetFDP_SFB`, or `ResetFDPGroup` to reset predictors in certain spectral lines at any time after creating the state.
4. Call the function `FDPFree` to free the memory allocated by `ippsFDPInitAlloc`.

FDPInitAlloc

Initializes predictor state.

```
IppStatus ippsFDPInitAlloc_32f(IppsFDPState_32f **pFDPState, int len);
```

Arguments

pFDPState	Pointer to the FDP state structure to be created.
len	Number of spectral lines to be processed.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsFDPInitAlloc` creates and initializes the FDP state.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to function is `NULL`.
`ippStsSizeErr`	Indicates an error when the length is less than or equal to 0.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

FDPFree

Closes FDP state.

```
IppStatus ippsFDPFree_32f(IppsFDPState_32f *pFDPState);
```

Arguments

`pFDPState`	Pointer to the FDP state structure to be closed.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsFDPFree` closes the FDP state by freeing all memory associated with the FDP state structure created by the function `FIRBlockInitAlloc`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

ResetFDP

Resets predictors for all spectral lines.

```
IppStatus ippsResetFDP_32f(IppsFDPState_32f *pFDPState);
```

Arguments

pFDPState	Pointer to the predictor specific state structure.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsResetFDP` resets predictors for all spectral lines.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

ResetFDP_SFB

Resets predictor-specific information in some scale factor bands.

```
IppStatus ippsResetFDP_SFB_32f (IppsFDPState_32f* pFDPState, const int*
    pBandsOffset, int bands_number, const Ipp8u *reset_flag);
```

Arguments

pFDPState	Pointer to the predictor specific state structure.
pBandsOffset	Pointer to band offset vector.
bands_number	Number of bands.

reset_flag	Array of flags showing whether predictors for spectral lines in a certain scale factor band need to be reset.

Discussion

This function is declared in the ippac.h header file. The function ippsResetFDP_32f resets predictors for all spectral lines in each scale factor band *i*, for which *reset_flag*[*i*] is not equal to 0.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one or more pointers passed to the function is NULL.
ippStsSizeErr	Indicates an error when *bands_number* is less than or equal to 0.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.

ResetFDPGroup

Resets predictors for group of spectral lines.

```
IppStatus ippsResetFDPGroup_32f (IppsFDPState_32f* pFDPState, int start, int
    step);
```

Arguments

pFDPState	Pointer to the predictor specific state structure.
start	Offset of the first spectral line in the group.
step	The distance between two neighbor spectral lines in the group.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsResetFDPGroup` resets predictors for each *`step`*-th spectral line beginning from the start up to the end of spectrum.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.
`ippStsSizeErr`	Indicates an error when *`reset_group_number`* or *`step`* is less than or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.

FDPFwd

Performs frequency domain prediction procedure and calculates prediction error.

```
IppStatus ippsFDPFwd_32f(IppsFDPState_32f* pFDPState, Ipp32f* pSrc,
    Ipp32f* pDst);
```

Arguments

`pFDPState`	Pointer to the predictor specific state structure.
`pSrc`	Pointer to the input data array.
`pDst`	Pointer to the data array to be filled with prediction errors.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsFDPFwd` applies frequency domain prediction procedure to the input signal *`pSrc`* and stores prediction errors in the *`pDst`* vector.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state structure is invalid.
`ippStsMisalignedBuf`	Indicates misaligned arrays. Supply aligned data for better performance.

FDPInv

Retrieves input signal from prediction error, using frequency domain prediction procedure.

```
IppStatus ippsFDPInv_32f(IppsFDPState_32f* pFDPState, Ipp32f* pSrcDst, const
    int* pBandsOffset, int bands_number, const Ipp8u* prediction_used);
```

Arguments

`pFDPState`	Pointer to the predictor specific state structure.
`pSrcDst`	Pointer to the input and output data array for the in-place operation.
`pBandsOffset`	Pointer to the band offset vector.
`bands_number`	Number of scale factor bands.
`prediction_used`	Array of flags showing whether prediction will be used in certain scale factor band.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsFDPInv` applies the procedure of frequency domain prediction to specific bands of the input spectral vector *`pSrcDst`*. Positions of bands are defined by the parameters *`bands_number`* and *`pBandsOffset`*.

For each scale factor band *i*, if *prediction_used*[*i*] is not equal to 0, all values of *pSrcDst* within the band are treated as a prediction error and the original signal is restored. If *prediction_used*[*i*] is equal to 0, all values of *pSrcDst* within the band are treated as a signal and will be passed without any changes.

Regardless of the *prediction_used* flag, the coefficients of each predictor are updated.

NOTE. *The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of pSrcDst vector is stored in the pBandsOffset*[*bands_number*] *element.*

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one or more pointers passed to the function is NULL.
ippStsSizeErr	Indicates an error when *bands_number* is less than or equal to 0.
ippStsContextMatchErr	Indicates an error when the state structure is invalid.
ippStsMisalignedBuf	Indicates misaligned arrays. Supply aligned data for better performance.

Huffman Algorithm Functions

There are several methods of compressing audio data.

Huffman decoding is a data size reduction method that uses statistical modeling to define which codes occur more frequently than others to build tables for subsequent encoding and decoding operations. Audio data in the bit stream is encoded with Variable Length Code (VLC) tables so that the shortest codes correspond to the most

frequent values and the longer codes correspond to the less frequent values. Every standard that uses Huffman decoding has corresponding tables listing possible codes and their values.

Example 10-2 Variable Length Code (VLC) Table Format

```
static Ipp32s Table[]=
{
max_bits,                The maximum length of code
total_subt,              The total number of all subtables
sub_sz1,                 The sizes of all subtables. Their sum must be
                         equal to the maximum length of code.
sub_sz2,
…,

sub_szTotal,
N1,                      The number of 1-bit codes
code1, value1,           The 1-bit codes and values. The number of
                         pairs must be equal to N1.
code2, value2,
...
codeN1, valueN1,
N2,                      The number of 2-bit codes
code1, value1,           The 2-bit codes and values. The number of
                         pairs must be equal to N2.
code2, value2,
...
codeN2, valueN2,
....
Nm,                      The number of maximum length codes.
code1, value1,           The m-bit codes and values. The number of
                         pairs must be equal to Nm.


code2, value2,
...
codeNm, valueNm,
-1                       The significant value to indicate the end of
                         table
};
```

Huffman algorithm is widely used, for example, in MPEG-1 layer 3 and AAC for coding MP3 samples and AAC, and coding AAC scale factors.

GetSizeHDT

Calculates size of the Huffman Decode table.

```
ippsGetSizeHDT_32s (const Ipp32s* pInputTable, Ipp32s* pBuffer, Ipp32s
    buffSize, Ipp32s *pSize);
```

Arguments

pInputTable	Pointer to the specified Huffman table, input parameter.
pBuffer	Pointer to a temporary buffer for calculation.
buffSize	Temporary buffer size.
pSize	Huffman decode table size, output parameter.

Discussion

This function is declared in the `ippac.h` header file. The function calculates size of Huffman decode table necessary for the call of the function `ippsBuildHDT`.

The parameter *buffSize* should be not less than $2^{maxlen\_of\_code} \cdot sizeof(int)$, where *maxlen_of_code* is the maximal length of the code in the specified Huffman table.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.

NOTE. *If the parameter buffSize is less than* $2^{maxlen\_of\_code} \cdot sizeof(int)$*, the function may operate incorrectly. No error code is returned in this case.*

BuildHDT

Builds Huffman decode table.

```
ippsBuildHDT_32s (const Ipp32s* pInputTable, Ipp32s* pDecodeTable, Ipp32s
    size);
```

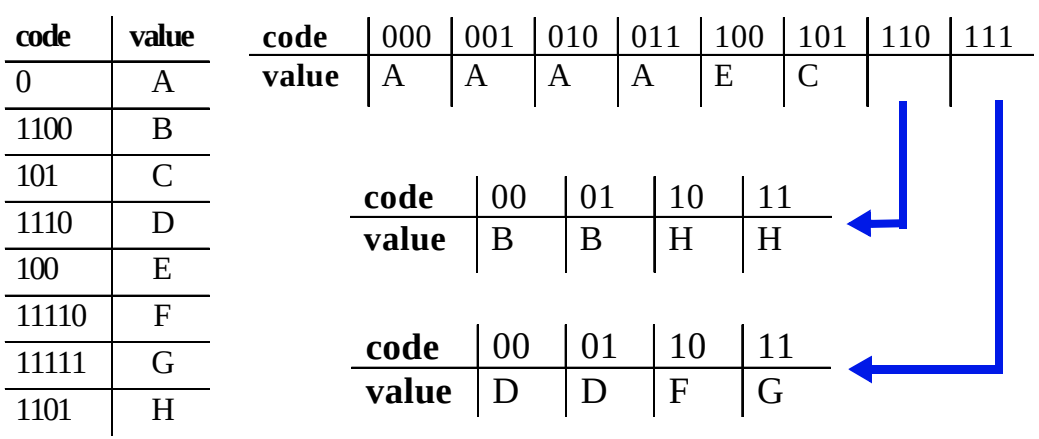
Arguments

pInputTable	Pointer to the specified Huffman table, input parameter.
pDecodeTable	Pointer to the memory where to build the resulting Huffman table.
size	Table size.

Discussion

This function is declared in the `ippac.h` header file. The function builds Huffman decode table necessary for the call of the function `ippsGetVLC`. You must allocate memory for the decode table and call the function `ippsGetHDT` to calculate the size of the table beforehand.

Figure 10-1 VLC Table and Subtables



Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.

DecodeVLC

Decodes value from Huffman table.

```
ippsDecodeVLC_32s (Ipp32s** pBitStream, Ipp32u* offset, Ipp32s* pDecodeTable,
    Ipp32s* pData);
ippsDecodeVLC_Block_32s,(Ipp32s **pBitStream, Ipp32u* offset, Ipp32s
    *pDecodeTable, Ipp32u length, Ipp16s *pData);
ippsDecodeVLC_MP3ESCBlock_32s,(Ipp32s **pBitStream, Ipp32u* offset, Ipp32s
    *pDecodeTable, Ipp32u length, Ipp32u linbits, Ipp16s *pData);
ippsDecodeVLC_AACESCBlock_32s,(Ipp32s **pBitStream, Ipp32u* offset, Ipp32s
    *pDecodeTable, Ipp32u length, Ipp16s *pData);
```

Arguments

pBitStream	Pointer to the bitstream.
offset	Pointer to offset between the bit that *pBitStream* points to and the start of the code.
pDecodeTable	Pointer to the Huffman table used for decoding.
length	Number of codes to be decoded.
linbits	Length of escape code.
pData	Pointer to the variable where the decoded value should be stored.

Discussion

This function is declared in the `ippac.h` header file. The function parses the bitstream and decodes variable length code using the table built by the function `ippsBuldHDT`, and resets the pointers to new positions. The pointer *`pBitStream`* points to the 32-bit value and the bit offset may vary from one to 32. After processing, the pointers are changed and their new values are returned.

`ippsDecodeVLC_32s.` The function `ippsDecodeVLC_32s` returns the table value corresponding to the found code.

`ippsDecodeVLC_Block_32s`
`ippsDecodeVLC_AACESCBlock_32s`
`ippsDecodeVLC_MP3Block_32s`

The functions `ippsDecodeVLC_Block_32s`, `ippsDecodeVLC_AACESCBlock_32s`, and `ippsDecodeVLC_MP3Block_32s` decode the block and fill the specified array, assigning two values to each code. The two values are obtained from the value corresponding to the code by extracting the value *`X`* from the second byte and the value *`Y`* from the first byte of the value corresponding to the code. The function subsequently processes the sign bits and the escape sequences, if necessary, and then stores the obtained values in the array.

See Example 10-2 for the VLC table structure and Figure 10-1 for an example of VLC table with its subtables.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one or more pointers passed to the function is `NULL`.
`ippStsVLCInputDataErr`	Indicates an error when incorrect input is used for encode/decode functions. For decode functions it can indicate that bitstream contain code that is not specified inside the used table.
`ippStsVLCAACEscCodeLengthErr`	Indicates an error when bitstream contains AAC-Esc code with the length more than 21.

GetSizeHET

Calculates size required for input Huffman table.

```
IppStatus ippsGetSizeHET_16s(const Ipp16s* pInputTable,Ipp32s * pSize);
```

Arguments

pInputTable	Input Huffman table in the specified format.
pSize	Address of the internal table size value in bytes.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsGetSizeHET` calculates the size necessary for the input Huffman table in an internal format. The size is calculated in bytes.

Example 10-3 shows the specified table format.

Example 10-3 Huffman Table in User Format

```
short huf_tabX[] = {
value1 , // max value in table
value2, //  length of  ESC-code
/* x, y, length of code in bit + sign bits, code */
0, 0,  2, 0x3,
0, 1,  2, 0x2,
……..
2, 2, 6, 0x0,
-1 /* end of table */
};
```

Return Value

`ippStsNoErr`	Indicates no error.

NOTE. *The column in the table that specifies the length of the code should also include the bits to store the sign information.*

BuildHET

Builds Huffman table.

```
IppStatus ippsBuildHET_16s (const Ipp16s* pInputTable, Ipp16s *
    pInternalTable);
```

Arguments

pInputTable	Input Huffman table in the specified format.
pInternalTable	Output Huffman table in the internal format.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsBuildHET_16s` builds a Huffman table in the internal format. You should allocate for the table sufficient memory in bytes, which is the size returned by the function `ippsGetSizeHET` and get the allocated memory as the *pInternalTable* parameter.

You should free the allocated memory when the Huffman encoding operations are completed.

Return Value

`ippStsNoErr`	Indicates no error.

HuffmanCountBits

Calculates size required for encoding quantized values.

```
IppStatus ippsHuffmanCountBits_16s(const Ipp16s* pInputData, Ipp32s length,
    const Ipp16s* pInternalTable, Ipp16s* pCountBits);
```

Arguments

pInputData	Pointer to the input array of quantized spectral values.
length	Input array size to be encoded with the specified Huffman table.
pInternalTable	Pointer to the Huffman table in the internal format.
pCountBits	Pointer to value of the required size in bits.

Discussion

This function is declared in the `ippac.h` header file. The function `ippsHuffmanCountBits` calculates the size in bits necessary to encode quantized values in the inner iteration loop of the encoding operation.

The column in the table that specifies the length of the code should also include the bits to store the sign information (see Example 10-3).

The input array value may not be greater the maximal value specified in the Huffman table. You can use the function `ippsThreshold_GT` to make sure that the input data meet the maximal value requirement.

The function adds the length of the escape codes to the total code length when escape tables are received as input data.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are null.

EncodeVLC

Encodes block of spectral values by using the specified Huffman table.

```
IppStatus ippsEncodeVLC_Block_16s(Ipp16s* pInputData, Ipp32s len, const
    Ipp16s*  pInternalTable, Ipp32s **pBitStream, Ipp32u* offset);
IppStatus ippsEncodeVLC_MP3ESCBlock_16s(Ipp16s* pInputData, Ipp32s len, const
    Ipp16s*  pInternalTable, Ipp32s **pBitStream, Ipp32u* offset);
```

Arguments

pInputData	Input array of quantized spectral values.
len	Input array values to be encoded with the input Huffman table.
pInternalTable	Huffman table in the internal format.
pBitStream	Pointer to the current double word in buffer.
offset	Pointer to the number of unread bits in the current double word.

Discussion

This function is declared in the `ippac.h` header file. The function encodes a block of spectral values by using the specified Huffman table.

The function `ippsEncodeVLC_Block_16s` is used for non-escape table encoding.

The function `ippsEncodeVLC_ESCBlock_16s` is used for escape table encoding.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are null.

`ippStsVLCInputDataErr`	Indicates an error when incorrect input is used for encode/decode functions. For decode functions it can indicate that bitstream contain code that is not specified inside the used table.

GetSizeHET_VLC

Calculates the size necessary for the input Huffman table in an internal format.

```
IppStatus ippsGetSizeHET_VLC_32s(const Ipp32s* pInputTable, Ipp32s * pSize);
```

Arguments

`pInputTable`	Input Huffman table in the specified format.
`pSize`	Address of the internal table size value in bytes.

Discussion

This function is declared in the `ippac.h` header file. The function calculates the size necessary for the input Huffman table in an internal format. The size is calculated in bytes.

Example 10-4 Huffman Table in User Format

```
Ipp32s    huff_table[] =
{
    value0, // Number of parameters in header.
    value1, // n-tuple size, can be 1, 2, 4 (required parameter)
    value2, // Unsigned table : 1 – yes, 0 – no (required parameter)
    value3, // Number of entries in table (required parameter)

    value4, // Esc–table type (optional parameter)
    value5, // Esc–code (optional parameter)
    value6, // Esc–code length (optional parameter)
       /*   x,   y,   w,   z, length, VLC-value  */
-1,  -1,   1,   0,        8,     0x00e8,
…
}
```

Example 10-5 Esc-table type

```
    ippVLCNonEscAlg = 0,
    ippVLCMp3EscAlg = 1,
    ippVLCAACEscAlg = 2
```

In case Esc-algorithm is used, the corresponding Huffman table should contain at least one-tuple with Esc-code inside. For this reason each of Esc parameters, namely Esc-table type, Esc-code, and Esc-code length are listed separately in the table.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the pointers to data arrays are null.
`ippStsVLCUsrTblHeaderErr`	Indicates an error when the user table has an invalid header.
`ippStsVLCUsrTblUnsupportedFmtErr`	Indicates an error when *n*-tuple size specified inside the user table header is incorrect.

ippStsVLCUsrTblEscAlgTypeErr	Indicates an error when Esc-code building algorithm specified inside the user table is not supported.
ippStsVLCUsrTblCodeLengthErr	Indicates an error when the user table contains code with unsupported length, for example, the code length is more than 32.

BuildHET_VLC

Builds a Huffman table in the internal format.

```
IppStatus ippsBuildHET_VLC_32s(Ipp32s* pInputTable, Ipp32s* pInternalTable);
```

Arguments

pInputTable	Input Huffman table in the specified format.
pInternalTable	Output Huffman table in the internal format.

Discussion:

This function is declared in the ippac.h header file. The function builds a Huffman table in the internal format.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are null.
ippStsVLCUsrTblHeaderErr	Indicates an error when the user table has an invalid header.
ippStsVLCUsrTblUnsupportedFmtErr	Indicates an error when *n*-tuple size specified inside the user table header is incorrect.

`ippStsVLCUsrTblEscAlgTypeErr`	Indicates an error when Esc-code building algorithm specified inside the user table is not supported.
`ippStsVLCUsrTblCodeLengthErr`	Indicates an error when the user table contains code with unsupported length, for example the code length is more than 32.

CountBits

Calculates the size in bits necessary to encode the input array.

```
IppStatus ippsCountBits_1tuple_VLC_16s(Ipp16s* pInputData, Ipp32s length,
    const Ipp32s * pInternalTable, Ipp16s* pCountBits);
IppStatus ippsCountBits_2tuple_VLC_16s(Ipp16s* pInputData, Ipp32s length,
    const Ipp32s * pInternalTable, Ipp16s* pCountBits);
IppStatus ippsCountBits_4tuple_VLC_16s(Ipp16s* pInputData, Ipp32s length,
    const Ipp32s * pInternalTable, Ipp16s* pCountBits);
```

Arguments

pInputData	Pointer to the input array.
length	Input array size to be encoded.
pInternalTable	Pointer to the Huffman table in internal format.
pCountBits	Pointer to the value of the required size in bits.

Discussion:

This function is declared in the `ippac.h` header file. The function calculates the size in bits necessary to encode the input array.

`ippsCountBits_2tuple_VLC_16s`. The function `ippsCountBits_2tuple_VLC_16s` is used for 2-tuple array.

ippsCountBits_4tuple_VLC_16s. The function ippsCountBits_4tuple_VLC_16s is used for 4-tuple array.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are null.
ippStsVLCInternalTblErr	Indicates an error when the internal table used for encoding is corrupt or not supported. For example, when a 2-tuple table is used for 4-tuple functions.

EncodeBlock

Encodes input array.

```
IppStatus ippsEncodeBlock_1tuple_VLC_16s(Ipp16s* pInputData, Ipp32s length,
    const Ipp32s * pInternalTable, Ipp32s** ppBitstream, Ipp32u* pOffset );
IppStatus ippsEncodeBlock_2tuple_VLC_16s(Ipp16s* pInputData, Ipp32s length,
    const Ipp32s * pInternalTable, Ipp32s** ppBitstream, Ipp32u* pOffset );
IppStatus ippsEncodeBlock_4tuple_VLC_16s(Ipp16s* pInputData, Ipp32s length,
    const Ipp32s * pInternalTable, Ipp32s** ppBitstream, Ipp32u* pOffset );
```

Arguments

pInputData	Pointer to the input array.
length	Input array size to be encoded.
pInternalTable	Pointer to the Huffman table in internal format.
ppBitstream	Pointer to the current double word in buffer.
pOffset	Pointer to the number of unread bits in the current double word.

Discussion

This function is declared in the ippac.h header file.

ippsEncodeBlock_1tuple_VLC_16s. The function ippsEncodeBlock_1tuple_VLC_16s encodes the input array by using the specified Huffman table.

ippsEncodeBlock_2tuple_VLC_16s. The function ippsEncodeBlock_2tuple_VLC_16s is used for 2-tuple non-escape table encoding.

ippsEncodeBlock_4tuple_VLC_16s. The function ippsEncodeBlock_4tuple_VLC_16s is used for 4-tuple non-esc table encoding.

Example 10-6 Possible way to use Huffman Coding functions described above

```
On initialization step:
    ippsGetSizeHET_VLC_32s(…);
    /// memory allocation
    ippsBuildHET_VLC_32s(…);
…
On counting bits step:

    Do
    {
        bit_number = ippsCountBits_XXX_VLC_16s(…)
        …
} While (bit_number > allowed_bit_number);

On bit stream step :

        ippsEncode_XXX_VLC_16s(…);
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the pointers to data arrays are null.
ippStsVLCInternalTblErr	Indicates an error when the internal table used for encoding is corrupted or unsupported, for example 2-tuple table is used for 4-tuple functions.
ippStsVLCInputDataErr	Indicates an error when incorrect input is used for encode/decode functions. For decoding functions it can indicate that the bitstream contains code unspecified in the table.

Vector Quantization Functions

This sections describes functions used for vector quantization operations.

CdbkInitAlloc

Initializes the codebook structure.

```
IppStatus ippsCdbkInitAlloc_VQ_32f(IppsCdbkState_VQ_32f** pCdbk, const Ipp32f*
    pSrc, int step, int height, Ipp_Cdbk_VQ_Hint hint);
```

Arguments

pCdbk	Pointer to the codebook structure to be created.
pSrc	Pointer to *Cdbk* table of the size *step * height* containing *height* quantization vectors of the length *step*.
step	Step to the next line in the table *pSrc*, quantization vector length.
height	Table height, number of quantization vectors.
hint	Reserved parameter.

Discussion

This function is declared in the `ippac.h` header file. This function initializes the structure that contains the codebook and additional information needed for the search operation. This structure is used during the vector quantization operation by the `PreSelect_VQ` , `MainSelect_VQ`, `IndexSelect_VQ`, and `VectorReconstruction_VQ` functions.

To free the memory allocated by this function, use the function `CdbkFree`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCdbk* or *pSrc* pointer is `NULL`.

CdbkFree

Closes the `IppsCdbkState_VQ_32f` structure created in `CdbkInitAlloc`.

```
IppStatus ippsCdbkFree_VQ_32f(IppsCdbkState_VQ_32f* pCdbk);
```

Arguments

pCdbk	Pointer to specified `IppsCdbkState_VQ_32f` structure.

Discussion

This function is declared in the `ippac.h` header file. The function closes the `IppsCdbkState_VQ_32f` structure created in `CdbkInitAlloc`.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pCdbk* or *pSrc* pointer is `NULL`.

PreSelect_VQ

Selects candidates for the nearest code vector of codebooks.

```
IppStatus ippsPreSelect_VQ_32f(const Ipp32f* pSrc, const Ipp32f* pWeight, int
    nDiv, const Ipp32s* pLengths, Ipp32s* pIndx, Ipps32s* pSign,
    IppsCdbkState_32f* pCdbk, int nCand, int* polbits);
```

Arguments

pSrc	Source vector to be quantized.
pWeight	Pointer to the vector of weights.
nDiv	Number of fragmentations of the *src* and *weights* vectors.

pLengths	Pointer to an array of lengths of fragmentations.
pIndx	Pointer to the output vector of indexes of the *nCand* minimum candidates.
pSign	Pointer to the output vector of signs of *nCand* minimum candidates. The value of 1 indicates that the minimal distortion appears when the norm is negative. The value of 0 indicates that the minimal distortion appears when the norm is positive.
pCdbk	Pointer to the specified IppsCdbkState_VQ_32f structure.
nCand	Number of output candidates.
polbits	Pointer to the *polbits* flag vector.

Discussion

This function is declared in the ippac.h header file. The function computes indexes and finds *nCand* vectors with the closest values from the codebook.

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

if the *polbits*[*idiv*] is greater than MAXBIT_SHAPE, the following distortion are also calculated

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

for *idiv* = 0 to *nDiv* - 1, *icb* is the number of the codebook line.

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

  $$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCdbk* initialized in the function CdbkInitAlloc.

The function then selects *nCand* candidates of *dist* and $dist_n$ with the minimum distortion measure.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pSrc* pointer is NULL.

MainSelect_VQ

Finds optimal indexes with minimal distortion.

```
IppStatus ippsMainSelect_VQ_32f(const Ipp32f* pSrc, const Ipp32f* pWeight,
    const Ipp32s* pLengths, int nDiv, int nCand, Ipp32s** pIndexCand,
    Ipps32s** pSignCand, Ipp32s** pIndx, Ipp32s** pSign, IppsCdbkState_32f**
    pCdbks, int nCdbks);
```

Arguments

pSrc	Source vector to be quantized.
pWeight	Pointer to the vector of lengths.
pLengths	Pointer to an array of lengths of fragmentations.
nDiv	Number of fragmentations of the *src* and *weights* vectors.
nCand	Number of input candidates.
pIndexCand	Pointer to the input vector of indexes of *nCand* minimum candidates.

pSignCand	Pointer to the input vector of signs of *nCand* minimum candidates.
pIndx	Pointer to the output vector of indexes.
pSign	Pointer to the output vector of signs.
pCdbks	Pointer to the specified IppsCdbkState_VQ_32f structure.
nCdbks	Number of codebooks.

Discussion

This function is declared in the ippac.h header file. The function restores vectors for all possible combinations of indexes across the specified number of codebooks then computes the distortion against the source vector. The function then returns the combination that provides for the minimal distortion.

The following formula serves for computing of the distortion for the given fragmentation *idiv* with the specified quantization vectors *icb*[*i*], where *i* is within the range of [0,*nCdbks*-1].

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp] - pSrcDiv[ismp])^2$$

where

$$rec[ismp] = \frac{\sum_{i=0}^{nCdbks-1} pSignCand[idiv*nCand+icb[i]]*ppTable[pIndexCand[idiv*nCand+icb[i]]][ismp]}{nCdbks}$$

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

  $$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCdbk* initialized in the function CdbkInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pSrc* pointer is NULL.

IndexSelect_VQ

Finds optimal vector set for specified number of codebooks.

```
IppStatus ippsIndexSelect_VQ_32f(const Ipp32f* pSrc, const Ipp32f* pWeight,
    int nDiv, const Ipp32s* pLengths,  int nCand, int** polbits, Ipp32s**
    pIndx, Ipp32s** pSign, IppsCdbkState_VQ_32f** pCdbks, int nCdbks);
```

Arguments

pSrc	Source vector to be quantized.
pWeight	Pointer to the vector of lengths.
nDiv	Number of fragmentations of the *src* and *weights* vectors.
pLengths	Pointer to an array of lengths of fragmentations.
nCand	Number of input candidates.
polbits	Indicates whether one or two norms should be used to compute the optimal vector set for the specified number of codebooks.
pIndx	Pointer to the output vector of indexes.

pSign	Pointer to the output vector of signs. The value of 1 indicates that the minimal distortion appears when the norm is negative. The value of 0 indicates that the minimal distortion appears when the norm is positive.
pCdbks	Pointer to the specified `IppsCdbkState_VQ_32f` structure.
nCdbks	Number of codebooks.

Discussion

This function is declared in the `ippac.h` header file. The function computes *nCand* vectors for each codebook by the following formula

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

if *polbits*[*idiv*] is 1, the following distortion is also calculated

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

for `idiv=0` to *nDiv* `-1`, *icb* – number of codebook line.

The function subsequently restores vectors for all possible combinations of indexes across the specified number of codebooks then computes the distortion against the source vector. The function then returns the combination that provides for the minimal distortion. The following formula serves for computing of the distortion for the given fragmentation *idiv* with the specified quantization vectors *icb*[*i*], where *i* is within the range of `[0,`*nCdbks*`-1]`.

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp] - pSrcDiv[ismp])^2$$

$$rec[ismp] = \frac{\sum_{i=0}^{nCdbks-1} pSignCand[idiv*nCand+icb[i]]^{*} ppTable[pIndexCand[idiv*nCand+icb[i]]][ismp]}{nCdbks}$$

for idiv=0 to *nDiv* -1, *icb* – number of codebook line.

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

  $$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

  $$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCdbk* initialized in the function CdbkInitAlloc.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pSrc* pointer is NULL.

NOTE. *You can use this function instead of calling the functions* PreSelect *and* MainSelect *to save time and memory cost.*

VectorReconstruction_VQ

Reconstructs vectors from indexes.

```
IppStatus ippsVectorReconstruction_VQ_32f(const Ipp32s** pIndx, const Ipp32s**
    pSign, const Ipp32s* pLength, int nDiv, IppsCdbkState_VQ_32f** pCdbk, int
    nCdbks, Ipp32f* pDst);
```

Arguments

pIndx	Pointer to array of input vectors of indexes for each codebook.
pSign	Pointer to array of input vectors of signs for each codebook containing either 1 or -1.
pLength	Pointer to array of lengths of partitions of output vector.
nDiv	Number of partitions of output vector.
pCdbk	Pointer to array of pointers to specified `IppsCdbkState_VQ_32f` structures.
pDst	Pointer to the reconstructed vector of spectrum values.
nCdbks	Number of codebooks.

Discussion

This function is declared in the `ippac.h` header file. The function reconstructs a vector divided on several partitions. A partition is defined as set of indexes of vectors from the specified number of codebooks, that is, one index for one partition from each codebook. To reconstruct the values of the output vector, the function calculates the arithmetic mean of vector values from the specified number of codebooks. The arrays *pIndx* and *pSign* contain the number of vectors from the codebook and the sign.

$$pDSt\left[\sum_{i=0}^{idiv-1} pLength[i]+j\right] = \frac{1}{nCdbk}\sum_{k=0}^{nCdbk-1} pSign[k][i]\cdot pCdbk[k]\rightarrow table[pIndx[k][i]][j]$$

for *i*=0 to *nDiv* –1 and for *j* = 0 to *pLength*[i]-1.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pCdbk* or *pSrc* pointer is NULL.

NOTE. *Length of pDst array should be equal to*

$$\sum_{k=0}^{nCdbk-1} pLengths[k]$$

Companding Functions

The functions described in this section perform an operation of data compression by using a logarithmic encoder-decoder, referred to as companding. Companding allows you to maintain a constant percentage error by logarithmically spacing the quantization levels [Rab78].

The Intel IPP companding functions perform the following conversion operations of signal samples:

- From 8-bit μ-law encoded format to PCM-linear or vice-versa.
- From 8-bit A-law encoded format to PCM-linear or vice-versa.
- From 8-bit μ-law encoded format to A-law encoded or vice-versa.

Samples encoded in μ-law or A-law format are non-uniformly quantized. The quantization functions used by these formats are designed to reduce the dependency of signal-to-noise ratio on the magnitude of the encoded signal. This is achieved by quantization (companding) at a finer resolution near zero, and at a coarse resolution at larger positive or negative levels. The output values are normalized to be in the range `[-1; +1]`.

These functions perform the μ-law and A-law companding in compliance with the CCITT G.711 specification. For the conversion rules and more details, refer to [CCITT].

Example 10-7 shows how to use companding functions.

MuLawToLin

Decodes samples from 8-bit μ-law encoded format to linear samples.

```
IppStatus ippsMuLawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMuLawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector, which stores 8-bit μ-law encoded signal samples to be decoded.
pDst	Pointer to the destination vector, which stores the linear sample results.
len	Number of samples in the vector.

Discussion

The function ippsMuLawToLin is declared in the ipps.h file. This function decodes the 8-bit μ-law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*.

The formula for μ-law companding is as follows:

$$|C_{\mu}(x)| = \frac{\ln(1 + 255 \cdot |x|)}{\ln(256)} \cdot 128, \quad -1 \le x \le 1$$

where x is the linear signal sample and $C_{\mu}(x)$ is the μ-law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

Application Notes

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of μ-law format is usually performed using look-up Tables 2a/G.711 and 2b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Return Value

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

LinToMuLaw

Encodes the linear samples using 8-bit μ-law format and stores them in a vector.

```
IppStatus ippsLinToMuLaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToMuLaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

Arguments

pSrc	Pointer to the vector that holds the signal samples (normalized to be less than 1.0) to be encoded.
pDst	Pointer to the vector that holds the output of the function ippsLinToMuLaw.
len	Number of samples in the vector.

Discussion

The function ippsLinToMuLaw is declared in the ipps.h file. This function encodes the PCM-linear samples in the vector *pSrc* using 8-bit μ-law format and stores them in the vector *pDst*.

Example 10-7 shows how to use the function ippsLinToMuLaw_32f8u.

Return Value

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

ALawToLin

Decodes the 8-bit A-law encoded samples to linear samples.

```
IppStatus ippsALawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsALawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

Arguments

pSrc	Pointer to the vector that holds the signal samples to be converted.
pDst	Pointer to the vector that holds the output of the function ippsALawToLin.
len	Number of samples in the vector.

Discussion

The function ippsALawToLin is declared in the ipps.h file. This function decodes the 8-bit A-law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*.

The formula for A-law companding is as follows:

$$|C_A(x)| = \begin{cases} \frac{87.56|x|}{1+\ln 87.56} \cdot 128, & 0 \le |x| \le \frac{1}{87.56} \\ \frac{1+\ln(87.56|x|)}{1+\ln 87.56} \cdot 128, & \frac{1}{87.56} < |x| \le 1 \end{cases},$$

where x is the linear signal sample and $C_A(x)$ is the A-law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

Application Notes

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of A-law format is usually performed using look-up Tables 1a/G.711 and 1b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Example 10-7 shows how to use the function `ippsALawToLin_8u32f`.

Return Value

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

LinToALaw

Encodes the linear samples using 8-bit A-law format and stores them in an array.

```
IppStatus ippsLinToALaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToALaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

Arguments

`pSrc`	Pointer to the vector that holds the signal samples to be encoded.
`pDst`	Pointer to the vector that holds the output of the function `ippsLinToALaw`.
`len`	Number of samples in the vector.

Discussion

The function `ippsLinToALaw` is declared in the `ipps.h` file. This function encodes the PCM-linear samples in the vector *`pSrc`* using 8-bit A-law format and stores them in the vector *`pDst`*.

Return Value

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

MuLawToALaw

Converts samples from 8-bit μ-law encoded format to 8-bit A-law encoded format.

```
IppStatus ippsMuLawToALaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector, which stores 8-bit μ-law encoded signal samples.
pDst	Pointer to the destination vector, which stores the 8-bit A-law encoded samples.
len	Number of samples in the vector.

Discussion

The function ippsMuLawToALaw is declared in the ipps.h file. This function converts signal samples from 8-bit μ-law encoded format in the vector *pSrc* to 8-bit A-law encoded format and stores them in the vector *pDst*.

Application Notes

The conversion of μ-law format to A-law format is usually performed using look-up Table 3/G.711shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Example 10-7 shows how to use the function ippsMuLawToALaw_8u.

Return Value

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

ALawToMuLaw

Converts samples from 8-bit A-law encoded format to 8-bit μ-law encoded format.

```
IppStatus ippsALawToMuLaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector, which stores 8-bit A-law encoded signal samples.
pDst	Pointer to the destination vector, which stores the 8-bit μ-law encoded samples.
len	Number of samples in the vector.

Discussion

The function ippsMuLawToALaw is declared in the ipps.h file. This function converts signal samples from 8-bit A-law encoded format in the vector *pSrc* to 8-bit μ-law format and stores them in the vector *pDst*.

Application Notes

The conversion of A-law format to μ-law format is usually performed using look-up Table 4/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

Return Value

ippStsNoErr	Indicates no error.

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

Example 10-7 Using Companding Functions

```
void compand( void ) {
     Ipp32f x[4] = { 0.1f, 0.2f, 0.3f, 0.4f };
     Ipp8u m[4], a[4];
     ippsLinToMuLaw_32f8u( x, m, 4 );
     ippsMuLawToALaw_8u( m, a, 4 );
     ippsALawToLin_8u32f( a, x, 4 );
     // now x must be close to original
     printf_32f("x =", x, 4, ippStsNoErr);
}
Output:
    x =  0.099609 0.207031 0.304688 0.398438
```

MP3 Audio Coding Functions

The ISO/IEC 11172-3 MPEG-1, Layer III also referred to as “MP3”) audio coding algorithms are widely used to compress stereophonic and dual-channel music signals. Well-suited for both transmission and storage applications, the MP3 algorithm delivers high-fidelity audio playback quality with bitrates as low as one-tenth of the original. As a result, the MP3 algorithm has become the *de facto* standard compression methodology for portable and handheld storage media, as well as for transmission of high-fidelity compressed audio over the Internet. MP3 encoder/decoder is widely used in music storage and audio recording.

Macros and Constants

The MP3 macro and constant definitions are listed in Table 10-2.

Table 10-2 MP3 Macro and Constant Definitions

Global Macro Name	Definition	Notes
IPP_MP3_GRANULE_LEN	576	The number of samples in one granule.
IPP_MP3_V_BUF_LEN	512	V data buffers length (32-bit words).
IPP_MP3_SF_BUF_LEN	40	Scale factor buffer length (8-bit words).
IPP_MP3_SFB_TABLE_LONG_LEN	138	Scale factor band table for long block length (16-bit words).
IPP_MP3_SFB_TABLE_SHORT_LEN	84	Scale factor band table for short block length (16-bit words).

Data Structures

The MP3 coding API includes several data structures.

The structure IppMP3FrameHeader contains the complete set of header information associated with one frame.

The structure IppMP3SideInfo contains the complete set of side information associated with one granule of one channel.

The structure IppMP3PsychoacousticModelTwoAnalysis contains the outputs generated by the Intel IPP implementation of ISO/IEC 11172-3 psychoacoustic analysis model 2, including estimates of the masked thresholds and perceptual entropy associated with the current frame. Masked thresholds are represented in terms of Mask-to-Signal Ratios (MSRs).

The structure IppMP3PsychoacousticModelTwoState contains the state information associated with the Intel IPP implementation of ISO/IEC 11172-3 psychoacoustic analysis model 2 to facilitate coherent block processing.

The structure IppMP3BitReservoir contains the state information associated with the quantization bit reservoir.

Frame Header

```
typedef struct {
    int id;            /* ID 1: MPEG-1, 0: MPEG-2 */
    int layer;         /* layer index 0x3: Layer I
                       //             0x2: Layer II
                       //             0x1: Layer III */
    int protectionBit; /* CRC flag 0: CRC on, 1: CRC off */
    int bitRate;       /* bit rate index */
    int samplingFreq;  /* sampling frequency index */
    int paddingBit;    /* padding flag 0: no padding, 1 padding */
    int privateBit;    /* private_bit, not used */
    int mode;          /* mono/stereo selection */
    int modeExt;       /* extension to mode */
    int copyright;     /* copyright or not, 0: no, 1: yes */
    int originalCopy;  /* original or copied, 0: copy, 1: original*/
    int emphasis;      /* flag indicating the type of de-emphasis */
    int CRCWord;       /* CRC-check word */

} IppMP3FrameHeader;
```

Side Information

```
typedef struct {
    int  part23Len;    /* number of main_data bits */
    int  bigVals;      /* half the number of Huffman code words whose
                          maximum amplitudes may be greater than 1 */
    int  globGain;     /* quantizes step size information */
    int  sfCompress;   /* number of bits used for scale factors */
    int  winSwitch;    /* window switch flag */
    int  blockType;    /* block type flag */
    int  mixedBlock;   /* flag 0: non mixed block, 1: mixed block */
    int pTableSelect[3];  /* Huffman table index for the 3 rectangle in
                             <big_values> field */
```

```
    int  pSubBlkGain[3]; /* gain offset from the global gain for one
                            subblock */
    int  reg0Cnt;      /* the number of scale factor bands in the
                          first region of <big_values> less one */
    int  reg1Cnt;      /* the number of scale factor bands in the
                          second region of <big_values> less one */
    int  preFlag;      /* flag indicating high frequency boost */
    int  sfScale;      /* scale factor scaling */
    int  cntlTabSel;   /* Huffman table index for the <count1> field
                          of quadruples */
} IppMP3SideInfo;
```

MP3 Psychoacoustic Model Two Analysis

```
typedef struct{
Ipp32s pMSR[36];/* MSRs for one granule/channel.
       For long blocks, elements 0-20 represent the thresholds
       associated with the 21 SFBs. For short blocks, elements
       0,3,6,..,33,
       elements 1,4,...,34, and elements 2,5,...,35, respectively,
       represent the thresholds associated with the 12 SFBs for each
       of the 3 consecutive short blocks in one granule/channel. That
       is, the block thresholds are interleaved such that the
       thresholds are grouped by SFB.*/
Ipp32s PE;     /* Estimated perceptual entropy, one granule/channel */
} IppMP3PsychoacousticModelTwoAnalysis;
```

Psychoacoustic Model Two State

```
typedef struct {
    Ipp64s pPrevMaskedThesholdLong[2][63]; /* long block masked
        threshold
        history buffer; Contains masked threshold estimates for the
        threshold
        calculation partitions associated with the two most recent long
        blocks */
```

```
Ipp64s pPrevMaskedThesholdShort[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;
```

MP3 Bit Reservoir

```
typedef struct {
    int BitsRemaining; /* bits currently remaining in the reservoir */
    int MaxBits;       /* maximum possible reservoir size, in bits,
        determined as follows:  min(7680-avg_frame_len, 2^9*8),
        where: avg_frame_len is the average frame length (in bits),
        including padding bits and excluding side information bits
} IppMP3BitReservoir;
```

MP3 Codec Enumerated Types

The Intel® IPP MP3 encoder and decoder APIs define enumerated data types that facilitate the synchronization and data transfer between the encoder and decoder components. As shown in Table 10-3, the MP3 codec API includes several enumerated types that provide semantic interpretations for frequently used constants.

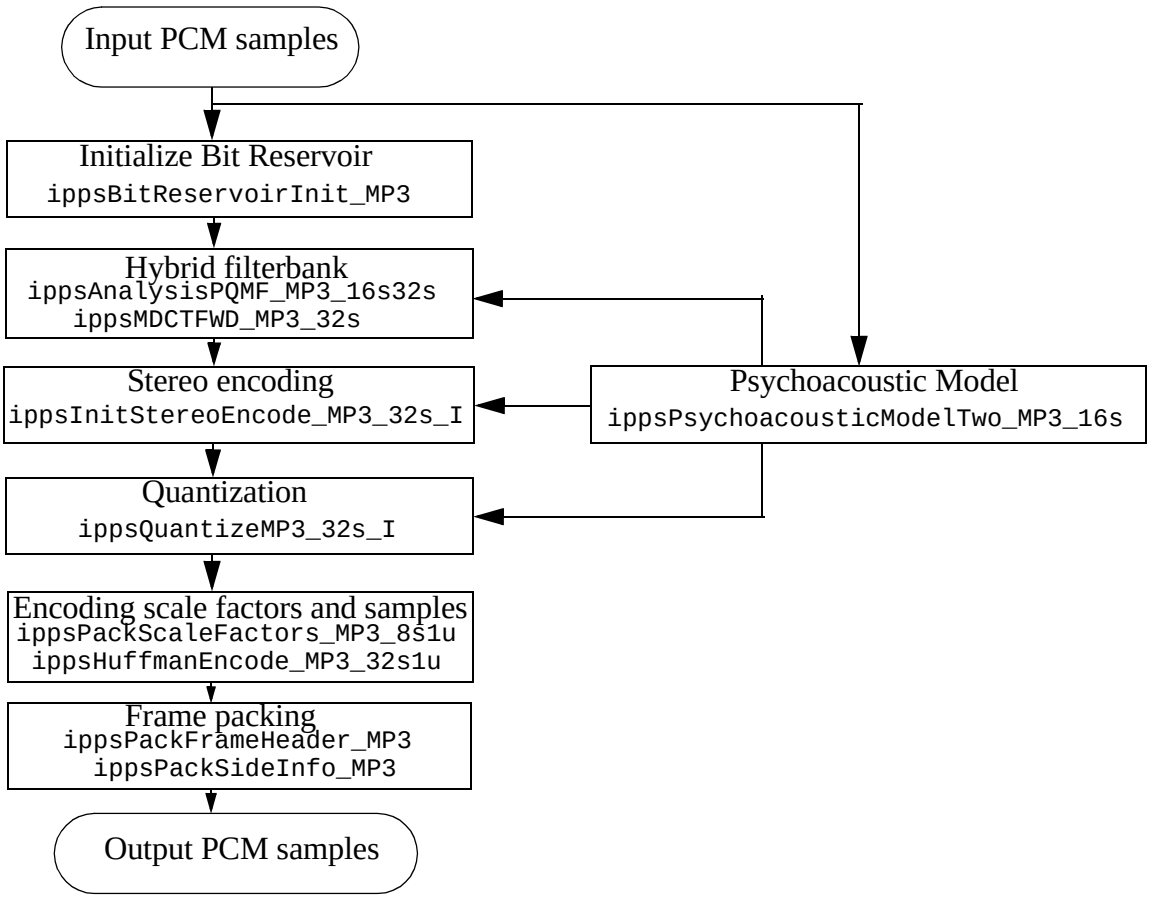
Table 10-3 MP3 Enumerated Data Types

Enumerated Type Name	Symbolic Values	Constant Value
IppMP3BitRate	ippMP3BitRateFree	0
	ippMP3BitRate32	1
	ippMP3BitRate40	2
	ippMP3BitRate48	3
	ippMP3BitRate56	4
	ippMP3BitRate64	5
	ippMP3BitRate80	6
	ippMP3BitRate96	7
	ippMP3BitRate112	8
	ippMP3BitRate128	9
	ippMP3BitRate160	10
	ippMP3BitRate192	11
	ippMP3BitRate224	12
	ippMP3BitRate256	13
	ippMP3BitRate320	14
IppMP3SampleRate	ippMP3SampleRate32000	2
	ippMP3SampleRate44100	0
	ippMP3SampleRate48000	1
IppMP3PcmMode	ippMP3NonInterleavedPCM	1
	ippMP3InterleavedPCM	2
IppMP3Emphasis	IppMP3EmphasisNone	0
	IppMP3Emphasis5015	1
	IppMP3EmphasisReserved	2
	IppMP3CCITTJ17	3

MP3 Audio Encoder

The MP3 encoder Application Programming Interface (API) provides a variety of capabilities, including bitstream packing functions and MP3 core encoding functions, See [ISO-11172]. This chapter provides a reference guide to the Intel® Integrated Performance Primitives (Intel® IPP) MP3 audio encoder API. As shown in Figure 10-2, this API includes several functions as well as predefined macros and constants.

Figure 10-2 Intel® IPP MP3 Encoder API Flowchart



AnalysisPQMF_MP3

Implements stage 1 of MP3 hybrid analysis filterbank.

```
IppStatus ippsAnalysisPQMF_MP3_16s32s (const Ipp16s *pSrcPcm, Ipp32s *pDstS,
    int pcmMode);
```

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 • most recent 480 (512-32) samples should be contained in the vector *pSrcPcm*[*pcmMode**i], where *i* = 0,1,..,479 • samples associated with the current granule should be contained in the vector *pSrcPcm*[*pcmMode***j*], where *j* = 480,481,..,1055.
pcmMode	PCM mode flag. Communicates to PQMF filterbank the type of input PCM vector organization to expect: • *pcmMode* = 1 denotes non-interleaved PCM input samples • *pcmMode* = 2 denotes interleaved PCM input samples.
pDstS	Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples under the following index: *pDstXs*[32**i* + *sb*], where *i* = 0,1,…,17 is time series index *sb* = 0,1,…,31 is the subband index.

Discussion

The function is declared in the ipps.h file. This function implements the first stage of MP3 hybrid analysis filterbank. The function applies the critically sampled block PQMF analysis bank characterized by the 512-sample prototype window to a PCM input audio vector.

Call the ippsAnalysisPQMF_MP3_16s32s function 18 times per granule on each channel, that is, 36 times per channel on each frame.

NOTE. *All coefficients are represented using the Q7.24 format*

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcPcm* or *pDstXs* is NULL.
ippStsErr	Indicates an error when *pcmMode* exceeds [1,2].

MDCTFwd_MP3

Implements stage 2 of the MP3 hybrid analysis filterbank.

```
IppStatus ippsMDCTFwd_MP3_32s (const Ipp32s *pSrc, Ipp32s *pDst, int
    blockType, int mixedBlock, IppMP3FrameHeader *pFrameHeader, Ipp32s
    *pOverlapBuf);
```

Arguments

pSrc	Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples that are indexed as follows: *pDstS*[32**i*+*sb*], where *i* = 0,1,…,17 is time series index, *sb* = 0,1,…,31 is the subband index. All coefficients are represented using the Q7.24 format.
blockType	Block type indicator: • 0 stands for normal block • 1 stands for start block

- 2 stands for short block
- 3 stands for stop block.

mixedBlock Mixed block indicator.

- 0 stands for not mixed
- 1 stands for mixed.

pFrameHeader Pointer to the IppMP3FrameHeader structure that contains the header associated with the current frame. Only MPEG-1 (*id* = 1) is supported.

pOverlapBuf Pointer to the MDCT overlap buffer that contains a copy of the most recent 576-element block of PQMF bank outputs. Prior to processing a new audio stream with the analysis filterbank, all elements of the buffer *pOverlapBuf* should be initialized to the constant value 0.

pDst Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank.

Discussion

The function is declared in the ipps.h file. This function implements the second stage of MP3 hybrid analysis filterbank by performing the following operations:

1. **Forward MDCT.** An appropriately arranged set of 12-point and/or 36-point forward Modified Discrete Cosine Transforms (MDCTs) is applied to the 18-sample spectral coefficient blocks generated on each of the 32 PQMF subbands during the first stage analysis.
2. **Aliasing reduction butterflies.** The butterflies specified in ISO/IEC 11172-3 are applied to the MDCT outputs in order to mitigate the aliasing artifacts introduced by cascading two critically sampled analysis filterbanks. Each of these introduces some non-negligible amount of interband aliasing.

The function ippsMDCTFwd_MP3_32s updates the 576-element MDCT overlap buffer *pMDCTOverlap*[], the contents of which must be preserved between calls to facilitate coherent block processing. The function must be applied once per granule on each channel (that is, applied twice per channel on each frame).

NOTE. *Input coefficients are represented in the Q7.24 format. Output coefficients are represented in the Q5.26 format.*

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the pointers *`pSrcXs`*, *`pDstXr`*, *`pFrameHeader`*, or *`pOverlapBuf`* is `NULL`.

PsychoacousticModelTwo_MP3

Implements ISO/IEC 11172-3 psychoacoustic model recommendation 2 to estimate masked threshold and perceptual entropy associated with a block of PCM audio input.

```
IppStatus ippsPsychoacousticModelTwo_MP3_16s (const Ipp16s *pSrcPcm,
    IppMP3PsychoacousticModelTwoAnalysis *pDstPsyInfo, int *pDstIsSfbBound,
    IppMP3SideInfo *pDstSideInfo, IppMP3FrameHeader *pFrameHeader,
    IppMP3PsychoacousticModelTwoState *pFramePsyState, Ipp32s *pWorkBuffer,
    int pcmMode);
```

Arguments

`pSrcPcm`	Pointer to the start of the buffer containing the input PCM audio vector, the samples of which should conform to the following format specification: • 16-bits per sample, signed, little-endian, Q15.

- *pSrcPcm* buffer should contain 1152 samples, that is, two granules of 576 samples each, if the parameter *pFrameHeader* -> *mode* has the value 1 (mono), or 2304 samples, that is, two granules of 576 samples each, if the parameter *pFrameHeader* -> *mode* has the value of 2 (stereo, dual mono).
- In the stereophonic case, the PCM samples associated with the left and right channels should be organized according to the *pcmMode* flag. Failure to satisfy any of the above PCM format and/or buffer requirements results in undefined model outputs.

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.

pFramePsyState Pointer to the first element in a set of IppMP3PsychoacousticModelTwoState structures that contain the psychoacoustic model state information associated with 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. Communicates the psychoacoustic model which type of PCM vector organization to expect:

- *pcmMode* = 1 denotes non-interleaved PCM input samples, that is, *pSrcPcm*[0..1151] contains the input samples associated with the left channel, and *pSrcPcm*[1152..2303] contains the input samples associated with the right channel.
- *pcmMode* = 2 denotes interleaved PCM input samples, that is, *pSrcPcm*[2**i*] and *pSrcPcm*[2**i*+1] contain the samples associated with the left and right channels, respectively, where *i* = 0,1,…,1151.

You can also use appropriately typecast elements `ippMP3NonInterleavedPCM` and `ippMP3InterleavedPCM` of the enumerated type `IppMP3PcmMode` as an alternative to the constants 1 and 2 for *pcmMode*.

pWorkBuffer Pointer to the workspace buffer internally used by the psychoacoustic model for storage of intermediate results and other temporary data. The buffer length must be at least 25,200 bytes, that is, 6300 elements of type `Ipp32s`.

pDstPsychoInfo Pointer to the first element in a set of `PsychoacousticModelTwoAnalysis` structures. Each set member contains the MSR and PE estimates for one granule. The number of elements in the set is equal to the number of channels, with the outputs arranged as follows: `(Analysis[0] = granule 1, channel 1), (…Analysis[1] = granule 1, channel 2), (…Analysis[2] = granule 2, channel 1), (…Analysis[3] = granule 2, channel 2).`

pDstIsSfbBound If intensity coding has been enabled, *pDstIsSfbBound* points to the list of SFB lower bounds above which all spectral coefficients should be processed by the joint stereo intensity coding module. Since the intensity coding SFB lower bound is block-specific, the number of valid elements pointed to by *pDstIsSfbBound* varies depending upon the individual block types associated with each granule. In particular, the list of SFB bounds is indexed as follows:

- *pIsSfbBound*`[3*`*gr*`]` for long block granules
- *pIsSfbBound*`[3*`*gr* `+` *w*`]` for short block granules,

where *gr* is the granule index (0 indicates granule 1 and 1 indicates granule 2), and *w* is the block index (0 indicates block 1, 1 indicates block 2, 2 indicates block 3).

For example, given short-block analysis in granule 1 followed by long block analysis in granule 2, the list of SFB bounds would be generated in the following order:
pIsSfbBound`[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}.`

Only one SFB lower bound decision is generated for long block granules, whereas three are generated for short block granules. If both MS and intensity coding are enabled, then the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding has been enabled, then the SFB bound represents the lowest non-MS SFB.

pDstSideInfo Pointer to the updated set of `IppMP3SideInfo` structures associated with all granules and channels. The model updates the following fields in all set elements: *blockType*, *winSwitch*, and *mixedBlock*. The number of elements in the set is equal to 2 times the number of channels. Ordering of the set elements is the same as *pDstPsychoInfo*.

pFramePsyState Pointer to the first element in a set of `IppMP3PsychoacousticModelTwoState` structures that contains the updated psychoacoustic model state information associated with both the current frame and next frame. The number of elements in the set is equal to the number of channels contained in the input audio. That is, a separate analysis is carried for each channel.

Prior to encoding a new audio stream, all elements of the psychoacoustic model state structure *pPsychoacousticModelState* should be initialized to contain the value 0.

In the signal processing domain, this could be accomplished using the function `ippsZero_16s` as follows:

```
ippsZero_16s ((Ipp16s *)
pPsychoacousticModelState,sizeof(IppMP3PsychoacousticMo
delTwoState)/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.

Discussion

The function is declared in the `ipps.h` file. This function implements the ISO/IEC 11172-3 psychoacoustic model recommendation 2 to estimate the masked threshold and perceptual entropy associated with a block of PCM audio input. Quantization process uses model outputs to estimate a perceptually optimal bit allocation for the spectral coefficients generated by the analysis filterbank. The psychoacoustic model also controls stereophonic MS/intensity mode selection and processing as well as analysis filterbank block size switching. Given one frame of PCM input audio of 1152 samples per channel, that is, two granules of 576 samples each, the psychoacoustic model generates the following outputs:

1. **Estimated SFB (scale factor band) Mask-to-Signal ratios (MSRs)**. The model generates a vector of estimated MSRs for the 21 SFBs in long block mode and 12 SFBs for each of three consecutive blocks in short block mode.

   The MSR is derived from the masked threshold, which quantifies the simultaneous masking power associated with one granule/channel (576 samples) of input audio. Given the properties of the audio stimulus presented to the listener, this threshold essentially quantifies the granule-instantaneous modified threshold of hearing. Ideally, the threshold estimate should provide a frequency-dependent intensity (dB SPL) profile beneath which an average listener cannot perceive quantization noise or, for that matter, any other spectral energy.

   To estimate the masked threshold from a block of input audio, the function `ippsPsych_MP3_16s` implements the procedure recommended in Annex D.2 of ISO/IEC 11172-3.

   First, the output of a classical FFT-based spectral analysis is grouped into threshold calculation partitions that are organized to achieve analysis with sub-critical bandwidth resolution. On each threshold calculation partition, the model employs a weighted estimate of tone-like or noise-like signal behavior determined by an assessment of a spectral unpredictability across time to estimate masking power in each partition.

   Second, a spreading function is applied to model the spectral selectivity of the

auditory system.

Finally, the estimated threshold is compared against the absolute threshold of hearing in quiet and the maximum of the two is assigned to the threshold calculation partition. Ultimately, in order to match its output to the bit allocation scheme of the quantization module, the model converts from the threshold calculation partition scale to a scale factor band (SFB) scale. One set of 21 SFB thresholds is generated for long blocks (576 samples), or three consecutive blocks of 12 SFB thresholds are generated for short blocks (192 samples).

To facilitate efficient quantization, the SFB thresholds are inverted and normalized by the signal energy and returned in a vector of SFB Mask-to-Signal ratios (MSRs). The estimated MSRs are returned in the `PsychoacousticModelTwoAnalysis` structure.

2. **Estimated perceptual entropy**. The model generates a perceptual entropy (PE) estimate for each granule. The PE quantifies the minimum number of bits required to represent the PCM samples of the granule with “perceptual transparency”. That is, without audible loss of quality for an average listener in comparison to the original, uncoded version.

   The estimated PE is derived from the masked threshold, in combination with classical assumptions about the minimum number of bits required to achieve a particular signal-to-noise ratio (SNR) target in each SFB, incremental per bit SNR improvement = +6 dB, where the minimum required SNR and hence minimum required number of bits for each SFB is derived from the signal-to-mask ratio (SMR).

   Perceptual entropy is used to control analysis filterbank block size switching, since sudden large PE increases are often associated with transient audio events that are prone to pre-echo distortion. The PE estimate is returned in the `PsychoacousticModelTwoAnalysis` structure.

3. **Analysis filterbank block size decision**. Using perceptual entropy and other indicators, the model determines whether or not the current granule is susceptible to pre-echo distortion. You should prefer using the short block

mode when pre-echoes are likely and use long blocks in all other cases.

In order to ensure selection of the appropriate block type, the decision incorporates single block look ahead switching logic. For example, if the current block type is long and the next block type is short, the current block type is changed from block type “long/normal” to block type “long/start” in order to guarantee seamless block processing upon mode switch. Similarly, if the current block type has been designated as “long/stop” and the next block type is determined to be “short”, the block switching logic changes the current block from “long/stop” back to “short” in order to avoid unnecessary mode switching. The block type decision is returned in the frame/granule `IppMP3SideInfo` structure.

4. **Joint stereophonic processing mode decision**. For 2-channel audio sources, the model evaluates interchannel correlations and other indicators in order to generate joint stereo LR/MS and/or intensity processing mode decisions. The joint stereo mode decision is returned in the `modeExt` field of the `IppMP3FrameHeader` structure.
5. **Intensity stereo coding SFB bound decision**. If intensity coding has been activated (see the preceding item joint stereophonic processing mode decision), the psychoacoustic model determines an appropriate lower SFB bound above which all spectral coefficients should be encoded using intensity mode stereophonic processing.

The psychoacoustic model performs analysis on a frame basis (1152 samples per channel), including two granules and up to two channels for either stereophonic or dual mono inputs. Valid lengths for both input and output vectors depend upon which mono or stereo channel modes have been enabled.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *`pSrcPcm`*, *`pDstPsyInfo`*, *`pDstSideInfo`*, *`pDstIsSfbBound`*, *`pFrameHeader`*, *`pDstPsyState`*, or *`pWorkBuffer`* is `NULL`.

JointStereoEncode_MP3

Transforms independent left and right channel spectral coefficient vectors into combined mid/side and/or intensity mode coefficient vectors suitable for quantization.

```
IppStatus ippsJointStereoEncode_MP3_32s_I (Ipp32s *pSrcDstXrL, Ipp32s
    *pSrcDstXrR, Ipp8s *pDstScaleFactorR, IppMP3FrameHeader *pFrameHeader,
    IppMP3SideInfo *pSideInfo, int *pIsSfbBound);
```

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 the function

entry, the `blockType` side information fields for both channels should reflect the analysis modes (short or long block) selected by the psychoacoustic model on each channel. All other fields in the *`pSideInfo`*`[0]` and *`pSideInfo`*`[1]` structures are ignored.

`pIsSfbBound` Pointer to the list of intensity coding SFB lower bounds for both channels of the current granule above which all L/R channel spectral coefficients are combined into an intensity-coded representation. The number of elements depends on the block type associated with the current granule. For short blocks, the SFB bounds are represented in the following order: *`pIsSfbBound`*`[0]` describes block 1, *`pIsSfbBound`*`[1]` describes block 2, and *`pIsSfbBound`*`[2]` describes block 3.
For long blocks, only a single SFB lower bound decision is required and is represented in *`pIsSfbBound`*`[0]`. If both MS and intensity coding are enabled, the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding is enabled, the SFB bound represents the lowest non-MS SFB.

`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 scale factors associated with one granule of the right/S channel. If intensity coding has been enabled by the psychoacoustic model above a certain SFB lower bound, as indicated by the frame header and the vector pointed to by `pIsSfbBound`, the function `StereoEncode_MP3_32s_I` updates with the appropriate scalefactors those elements of *`pDstScaleFactorR`*`[]` that are associated with intensity coded scale factor bands. Other

SFB entries in the scale factor vector remain unmodified. The length of the vector referenced by *pDstScaleFactorR* varies as a function of block size. The vector contains 21 elements for long block granules, or 36 elements for short block granules.

Discussion

The function is declared in the `ipps.h` file. This function transforms the independent left and right channel spectral coefficient vectors into combined mid/side (MS) and/or intensity (IS) mode coefficient vectors suitable for quantization. If MS coding is enabled (*pFrameHeader* -> *modeExt* & 0x10 == 1), the left and right channels are converted to Mid and Side channels as follows:

$$M = \frac{L+R}{\sqrt{2}} \quad S = \frac{L-R}{\sqrt{2}}$$

This function is called on dual granule basis. You should call it for every granule/2 channels.

If intensity coding is enabled (*pFrameHeader* -> *modeExt* & 0x01 = 1), the left channel carries the intensity data for SFBs above the SFB intensity lower bound and the right channel above the SFB lower bound is cleared. All coefficients are set to 0.

$$L = L + R, R = 0$$

In order to facilitate energy-proportional recovery of the left and right spectral coefficients at the decoder, an intensity energy scale factor, *is_pos*, is transmitted in place of the right channel scale factor since the right channel spectral coefficients above the SFB bound are eliminated. The energy normalization constant is derived from the L/R SFB energy ratios and then transformed to improve its quantization properties. That is,

$$is\_pos = n \operatorname{int}\left(\frac{12}{\pi}\arctan\left(\sqrt{\frac{L\_energy}{R\_energy}}\right)\right)$$

where *L_energy* and *R_energy* are, respectively, the SFB energies associated with the spectral coefficients of the left and right channels. At the decoder, the *is_pos* scale factor is used to apportion jointly coded signal energy between the left and right channels in a manner consistent with the distribution of signal energy prior to joint coding. The *is_pos* intensity scalefactors are returned in the vector *pDstScalefactorR*.

On each granule, joint stereo coding is applied once per channel pair (576 samples per granule on each channel). The function `JointStereoEncode_MP3_32s_I` must therefore be called once per granule, or twice per frame.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *pSrcDstXrL*, *pSrcDstXrR*, *pDstScaleFactorR*, *pFrameHeader*, *pSideInfo*, or *pIsSfbBound* is `NULL`.
`ippStsMP3SideInfoErr`	Indicates an error if `pSideInfo[0].blockType!= pSideInfo[1].blockType` when IS or MS is used.

Quantize_MP3

Quantizes spectral coefficients generated by analysis filterbank.

```
IppStatus ippsQuantize_MP3_32s_I (Ipp32s *pSrcDstXrIx, Ipp8s *pDstScalefactor,
    int *pDstScfsi, int *pDstCount1Len, int *pDstHufSize, IppMP3FrameHeader
    *pFrameHeader, IppMP3SideInfo *pSideInfo,
    IppMP3PsychoacousticModelTwoAnalysis *pPsychoInfo,
    IppMP3PsychoacousticModelTwoState *pFramePsyState, IppMP3BitReservoir
    *pResv, int meanBits, int *pIsSfbBound, Ipp32s *pWorkBuffer);
```

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
- *pSrcDstXrIx*[*gr**576 + *i*] for monaural, that is, input sources,

where

- *i* = 0,1,…,575 is the spectral coefficient index
- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

Depending on which type of joint coding has been applied, if any, the coefficients for each channel could be associated with L/R, M/S, and/or intensity representations of the input audio. All coefficients should be represented using the Q5.26 format.

pFrameHeader Pointer to the `IppMP3FrameHeader` structure that contains the header information associated with the current frame. Upon the function entry, the structure fields `samplingFreq`, `id`, `mode`, and `modeExt` should contain, respectively, the sample rate associated with the current input audio, the algorithm `id`, that is, MPEG-1 or MPEG-2, and the joint stereo coding commands generated by the psychoacoustic model. All other **pFrameHeader* fields are ignored. Only MPEG-1 (*id* = 1) is supported.

pSideInfo Pointer to the set of `IppMP3SideInfo` structures associated with all granules and channels. The set should contain 2**nchan*, elements and should be indexed as follows: *pSideInfo*[*gr***nchan* + *ch*], where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.

- *nchan* is the number of channels.
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

Upon the function entry, in all set elements the structure fields *blockType*, *mixedBlock*, and *winSwitch* should contain, respectively, the block type indicator (*start*, *short*, or *stop*), filter bank mixed block analysis mode specifier, and window switching flags (*normal* or *blockType*) associated with the current input audio. All other **pSideInfo* fields are ignored upon the function entry and updated upon function exit, as described below under the description of output arguments.

pPsychoInfo Pointer to the first element in a set of `PsychoacousticModelTwoAnalysis` structures associated with the current frame. Each set member contains the MSR and PE estimates for one channel of one granule. The set should contain `2*`*nchan*, elements and is indexed as: `pPsychoaInfo[`*gr*`*`*nchan*`+ ch]`, where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2
- *nchan* is the number of channels
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

pFramePsyState Pointer to the first element in a set of `IppMP3PsychoacousticModelTwoState` structures that contains the psychoacoustic model state information associated with both the current frame and next frame. The number of elements in the set is equal to the number of channels contained in the input audio. That is, a separate analysis is carried for each channel. The quantizer uses the frame type look ahead information *nextBlockType* to manage the bit reservoir. All other structure elements are ignored by the quantizer.

pResv Pointer to the `IppMP3BitReservoir` structure that contains the bit reservoir state information. Upon the function entry, all structure fields should contain valid data.

meanBits The number of bits allocated on an average basis for each frame of spectral coefficients and scalefactors given the target bit rate (kilobits per second) specified in the frame header. This number excludes the bits allocated for the frame header and side information. The quantizer uses *meanBits* as a target allocation for the current frame. Given perceptual bit allocation requirements greater than this target, the quantizer makes use of the surplus bits held in the bit reservoir to satisfy frame-instantaneous demands. Similarly, given perceptual bit allocation requirements below this target, the quantizer will store surplus bits in the bit reservoir for use by future frames.

pIsSfbBound Pointer to the list of SFB lower bounds above which all L/R channel spectral coefficients have been combined into an intensity-coded representation. The number of valid elements pointed to by pIsSfbBound depends upon the block types associated with the granules of the current frame.

In particular, the list of SFB bounds pointed to by pIsSfbBound is indexed as follows: *pIsSfbBound*[3**gr*] for long block granules, and *pIsSfbBound*[3**gr* + *w*] for short block granules, where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2
- *w* is the block index. 0 stands for block 1, 1 stands for block 2, 2 stands for block 3.

For example, given short-block analysis in granule 1 followed by long block analysis in granule 2, the list of SFB bounds would be expected in the following order:
pIsSfbBound[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}.

If a granule is configured for long block analysis, then only a single SFB lower bound decision is expected, whereas three are expected for short block granules. If both MS and intensity coding have been enabled, then the SFB intensity coding lower bound simultaneously

represents the upper bound SFB for MS coding. If only MS coding has been enabled, then the SFB bound represents the lowest non-MS SFB.

pWorkBuffer Pointer to the workspace buffer internally used by the quantizer for storage of intermediate results and other temporary data. The buffer length should be at least 2880 bytes, that is, 720 words of 32-bits each.

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 single channel input sources, where

- *i*=0,1,…,575 is the spectral coefficient index.
- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

pDstScaleFactor Pointer to the output set of scalefactors generated during the quantization process. These scalefactors determine the quantizer granularity. Scale factor vector lengths depend on the block mode associated with each granule. The order of the elements is:

1. (granule 1, channel 1)
2. (granule 1, channel 2)
3. (granule 2, channel 1)
4. (granule 2, channel 2).

Given this general organization, the side information for each granule/channel in conjunction with the flags contained in the vector *pDstScfsi* can be used to determine the precise scale factor vector indices and lengths.

pDstScfsi Pointer to the output vector of scale factor selection information. This vector contains a set of binary flags that indicate whether or not scalefactors are shared across granules of a frame within predefined

scale factor selection groups. For example, bands 0,1,2,3,4,5 form one group; bands 6,7,8,9,10 form a second group, as defined in ISO/IEC 11172-3. The vector is indexed as follows: *pDstScfsi*[*ch*][*scfsi_band*], where

- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.
- *scfsi_band* is the scale factor selection group number. Group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, 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 of a higher frequency than the *bigvals* region, the *count1* parameter indicates the size of the region in which spectral samples can be combined into quadruples for which all elements are of magnitude less than or equal to 1. The vector contains 2*nchan, elements and is indexed as follows: *pDstCount1Len*[*gr*nchan + ch*], where

- *gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- *nchan* is the number of channels.
- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

pDstHufSize Pointer to an output vector of Huffman coding bit allocation specifiers. For each granule/channel, the specifiers indicate the total number of Huffman bits required to represent the quantized spectral coefficients in the *bigvals* and *count1* regions.

Whenever necessary, each *HufSize* bit count is augmented to include the number of bits required to manage the bit reservoir. For frames in which the reservoir has reached the maximum capacity, the quantizer expends the surplus bits by padding with additional bits the Huffman representation of the spectral samples. The *HufSize* result returned by the quantizer reflects these padding requirements. That is, *HufSize*[i] is equal to the sum of the number of bits required for

Huffman symbols and the number of padding bits. The vector contains `2*nchan`, elements and is indexed as follows: `pDstHufSize[gr*nchan+ch]`, where

- `gr` is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- `nchan` is the number of channels.
- `ch` is the channel index. 0 stands for channel 1, 1 stands for channel 2.

`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`.

The set contains `2*nchan`, elements and is indexed as follows: `pSideInfo[gr*nchan+ch]`, where

- `gr` is the granule index. 0 stands for granule 1, 1 stands for granule 2.
- `nchan` is the number of channels.
- `ch` is the channel index. 0 stands for channel 1, 1 stands for channel 2.

`pResv` 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.

Discussion

The function is declared in the `ipps.h` file. This function quantizes the spectral coefficients generated by the analysis filterbank such that the resulting distortion, that is, quantization noise, is shaped to match a profile derived from the masked thresholds estimated by the psychoacoustic model.

Along with meeting perceptual distortion criteria, the quantizer simultaneously adjusts the overall bit allocation to achieve a fixed bit rate target. In accordance with the ISO/IEC 11172-3 recommendation, a bit reservoir is maintained in order to meet

instantaneous peak rate demands without violating on an average basis the fixed rate constraint. Surplus bits are deposited in the reservoir during frames with lower than average perceptual bit rate requirements. Supplementary bits are withdrawn from the reservoir to satisfy frames with higher-than-average perceptual bit allocation requirements.

The quantizer manages the reservoir and overall bit allocation such that a constant average rate constraint is satisfied. The quantizer operates on a complete frame of data, that is, two granules and either one or two channels, and it should therefore be called once per frame.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrcDstXrIx`*, *`pDstScaleFactor`*, *`pDstScfsi`*, *`pDstCount1Len`*, *`pDstHufSize`*, *`pFrameHeader`*, *`pSideInfo`*, *`pPsychInfo`*, *`pFramePsyState`*, *`pResv`*, **`pIsSfbBound`**, or *`pWorkBuffer`* is `NULL`.
`ippStsMP3SideInfo`	Indicates an error when *`pSideInfo->winSwitch`* and *`pSideInfo->mixedBlock`* are both defined.

PackScalefactors_MP3

Applies noiseless coding to scalefactors and packs output into bitstream buffer.

```
IppStatus ippsPackScalefactors_MP3_8s1u (const Ipp8s *pSrcScalefactor, Ipp8u
    **ppBitStream, int *pOffset, IppMP3FrameHeader *pFrameHeader,
    IppMP3SideInfo *pSideInfo, int *pScfsi, int granule, int channel);
```

Arguments

`pSrcScaleFactor` Pointer to a vector of scalefactors generated during the quantization process for one channel of one granule. Scale factor vector lengths depend on the block mode. Short block granule scale factor vectors contain 36 elements, or 12 elements for each subblock. Long block

granule scale factor vectors contain 21 elements.

Thus short block scale factor vectors are indexed as follows: *pSrcScaleFactor*[*sb**12+*sfb*], where

- *sb* is the subblock index. 0 stands for subblock 1, 1 stands for subblock 2, 2 stands for subblock 3.
- *sfb* is the scale factor band index (0-11).

Long block scale factor vectors are indexed as follows: *pSrcScaleFactor*[*sfb*], where *sfb* is the scale factor band index (0-20).

The associated side information for an individual granule/channel can be used to select the appropriate indexing scheme.

ppBitStream Pointer to the encoded bitstream buffer. The *ppBitStream* parameter is a double pointer to the first byte in the bitstream buffer intended to receive the Huffman-encoded scale factor bits generated by the function EncodeScaleFactors_MP3_8s1u. The scale factor 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 Bitstream 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 the function entry, the structure fields *id* and *modeExt* should contain, respectively, the algorithm *id* (MPEG-1 or MPEG-2) and the joint stereo coding commands generated by the psychoacoustic model. All other **pFrameHeader* fields are ignored. Only MPEG-1 (*id* = 1) is supported.

pSideInfo Pointer to the IppMP3SideInfo structure for the current granule and channel. Upon function entry, the structure fields *blockType*, *mixedBlock*, and *sfCompress* should contain, respectively, the block

type indicator *start*, *short*, or *stop*, filter bank mixed block analysis mode specifier, and scale factor bit allocation. All other **pSideInfo* fields are ignored by the scale factor encoder.

pScfsi Pointer to the scale factor selection information table that contains the set of binary flags that indicate whether scalefactors are shared across granules of a frame within the predefined scale factor selection groups.

For example, bands 0,1,2,3,4,5 form one group and bands 6,7,8,9,10 form a second group (as defined in ISO/IEC 11172-3). The vector is indexed as follows: *pScfsi*[*ch*][*scfsi_band*], where

- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.
- *scfsi_band* is the scale factor selection group number. Group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, and group 3 includes SFBs 16-20.

granule Index of the current granule. 0 stands for granule 1, 1 stands for granule 2.

channel Index of the current channel. 0 stands for channel 1, 1 stands for channel 2.

ppBitStream Updated bitstream byte pointer. This parameter points to the first available bitstream buffer byte immediately following the bits generated by the scale factor Huffman encoder and sequentially written into the stream buffer. The scale factor bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3.

pOffset Updated bitstream bit pointer. The *pOffset* parameter indexes the next available bit in the next available byte referenced by the updated bitstream buffer byte pointer **ppBitStream*. This parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit.

Discussion

The function is declared in the `ipps.h` file. This function applies noiseless coding to the scalefactors and then packs the output into the bitstream buffer. This function operates on one channel of one granule at a time. Therefore, you should call this function once for each channel of each granule.

The resulting bitstream is fully compliant with the syntax specified in ISO/IEC 11172-3.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrcScaleFactor`*, *`ppBitStream`*, *`ppBitStream`*, *`pOffset`*, *`pFrameHeader`*, *`pSideInfo`*, or *`pScfsi`* is `NULL`.
`ippStsMP3SideInfoErr`	Indicates an error when *`pFrameHeader->id`* `== IPP_MP3_ID_MPEG1` and *`pSideInfo->sfCompress`* exceeds `[0..15]`; *`pFrameHeader->id`* `== IPP_MP3_ID_MPEG2` and *`pSideInfo->sfCompress`* exceeds `[0..511]`.
`ippStsMP3FrameHeaderErr`	Indicates an error when *`pFrameHeader->id`* `== IPP_MP3_ID_MPEG2` and *`pFrameHeader->modeExt`* exceeds `[0..3]`.

HuffmanEncode_MP3

Applies lossless Huffman encoding to quantized samples and packs output into bitstream buffer.

```
IppStatus ippsHuffmanEncode_MP3_32s1u (Ipp32s *pSrcIx, Ipp8u **ppBitStream,
    int *pOffset, IppMP3FrameHeader *pFrameHeader, IppMP3SideInfo *pSideInfo,
    int count1Len, int hufSize);
```

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	Bitstream byte pointer. The *ppBitStream* parameter is a double pointer to the first byte in the bitstream 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	Bitstream 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 big value spectral region. The Huffman encoder ignores all other frame header fields.Only MPEG-1 (*id* = 1) is supported.
pSideInfo	Pointer to the `IppMP3SideInfo` structure for the current granule and channel. The structure elements `bigVals`, `pTableSelect[0]-[2]`, `reg0Cnt`, and `reg1Cnt` are used to control coding of spectral coefficients in the big value region. The structure element `cnt1TabSel` is used to select the appropriate Huffman table for the (-1,0,+1)-valued 4-tuples in the *count1* region. For detailed descriptions of all side information elements, see the structure definition header file.
count1Len	The `count1` region length specifier. Indicates the number of spectral samples for the current granule/channel above the big value region that can be combined into 4-tuples in which all elements are of magnitude less than or equal to 1.

hufSize	Huffman coding bit allocation specifier. Indicates the total number of bits that are required to represent the Huffman-encoded quantized spectral coefficients for the current granule/channel in both the *bigvals* and *count1* regions. Whenever necessary, this bit count should be augmented to include the number of bits required to manage the bit reservoir. For frames in which the reservoir has reached maximum capacity, the surplus bits are expended by padding with additional bits the Huffman representation of the spectral samples. The *HufSize* result returned by the function Quantize_MP3_32s_I reflects these padding requirements. That is, *HufSize*[*i*] is equal to the total of the number of bits required for Huffman symbols and the number of padding bits.
ppBitStream	Updated bitstream byte pointer. The parameter **ppBitStream* points to the first available bitstream buffer byte immediately following the bits generated by the spectral coefficient Huffman encoder and sequentially written into the stream buffer. The Huffman symbol bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3.
pOffset	Updated bitstream bit pointer. The *pOffset* parameter indexes the next available bit in the next available byte referenced by the updated bitstream buffer byte pointer **ppBitStream*. This parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit.

Discussion

The function is declared in the ipps.h file. This function applies lossless Huffman encoding to the quantized samples and packs the output into the bitstream buffer.

The function encodes one granule at a time, and therefore must be called once for each granule of each channel.

The resulting bitstream is fully compliant with ISO/IEC 11172-3.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *ppBitStream*, *pOffset*, *pSrcIx*, *pSideInfo*, *ppBitStream*, or *pFrameHeader* is NULL.
ippStsBadArgErr	Indicates an error when *pOffset* exceeds [0,7].
ippStsMP3SideInfoErr	Indicates an error when *pSideInfo*->bigVals*2>IPP_MP3_GRANULE_LEN (pSideInfo->reg0Cnt + pSideInfo->reg1Cnt + 2) >= 23, *pSideInfo->cnt1TabSel* exceeds [0,1], *pSideInfo->pTableSelect*[i] exceeds [0..31].
ippStsMP3FrameHeader	Indicates an error when *pFrameHeader*->id != 1 *pFrameHeader*->layer != 1 *pFrameHeader*->samplingFreq exceeds [0..2].

PackFrameHeader_MP3

Packs the content of the frame header into the bitstream.

```
IppStatus ippsPackFrameHeader_MP3 (IppMP3FrameHeader *pSrcFrameHeader, Ipp8u
    **ppBitStream);
```

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 bitstream buffer. The *ppBitStream* parameter is a double pointer to the first byte in the bitstream 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*.
ppBitStream	Updated bitstream byte pointer. The parameter **ppBitStream* points to the first available bitstream buffer byte immediately following the packed frame header bits. The frame header bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3.

Discussion

The function is declared in the `ippac.h` file. This function packs the content of the frame header into the bitstream.

The resulting bitstream is fully compliant with the syntax specified in ISO 11172-3.

The function should be called once per frame.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrcFrameHeader*, *ppBitStream*, or *ppBitStream* is `NULL`.

PackSideInfo_MP3

Packs the side information into the bitstream buffer.

```
IppStatus ippsPackSideInfo_MP3 (IppMP3SideInfo *pSrcSideInfo, Ipp8u
    **ppBitStream, int mainDataBegin, int privateBits, int *pSrcScfsi,
    IppMP3FrameHeader *pFrameHeader);
```

Arguments

pSrcSideInfo Pointer to the IppMP3SideInfo structures. This should contain twice the channel number of elements. The order is the following:

- granule 1, channel 1
- granule 1, channel 2
- granule 2, channel 1
- granule 2, channel 2.

All fields of all set elements should contain valid data upon the function entry.

mainDataBegin Negative bitstream offset, in bytes. The parameter value is typically the number of bytes remaining in the bit reservoir before the start of quantization for the current frame. When computing *mainDataBegin*, you should exclude the header and side information bytes.

The side information formatter packs the 9-bit value of *mainDataBegin* into the main_data_begin field of the output bitstream.

privateBits Depending on the number of channels, the function extracts the appropriate number of least significant bits from the parameter *privateBits* and packs them into the private_bits field of the output bitstream. The ISO/IEC 11172-3 bitstream syntax reserves a channel-dependent number of application-specific (private) bits in the layer III bitstream audio data section immediately following the parameter main_data_begin. See ISO/IEC 11172-3:1993. For dual- and single-channel streams, respectively, three and five bits are reserved.

pSrcScfsi Pointer to the scale factor selection information table. This vector contains a set of binary flags that indicate whether scalefactors are shared across granules of a frame within predefined scale factor selection groups. For example, bands 0,1,2,3,4,5 form one group and bands 6,7,8,9,10 form the second group, as defined in ISO/IEC 11172-3 [2].

The vector is indexed as *pDstScfsi*[*ch*][*scfsi_band*], where

- *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.
- *scfsi_band* is the scale factor selection group number. Group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, and group 3 includes SFBs 16-20.

pFrameHeader Pointer to the IppMP3FrameHeader structure. Only MPEG-1 (*id* = 1) is supported. Upon the function entry, the structure fields id, mode, and layer should contain, respectively, the algorithm id (MPEG-1 or MPEG-2), the mono or stereo mode, and the MPEG layer specifier.

All other **pFrameHeader* fields are ignored.

ppBitStream Pointer to the encoded bitstream buffer. The parameter is a double pointer to the first byte in the bitstream 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*.

ppBitStream Updated bitstream byte pointer. The parameter **ppBitStream* points to the first available bitstream buffer byte immediately following the packed side information bits. The frame header bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3:1993.

Discussion

The function is declared in the ippac.h file. This function packs the side information into the bitstream buffer.

The resulting bitstream is fully compliant with the syntax specified in ISO 11172-3.

This function should be called once per frame.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcSideInfo*, *ppBitStream*, *ppBitStream*, *pSrcScfsi*, or *pFrameHeader* is NULL.

`ippStsMP3FrameHeaderErr`	Indicates an error when *pFrameHeader*`->`*id* exceeds `[0,1]` *pFrameHeader*`->layer != 1` *pFrameHeader*`->mode` exceeds `[0..3]`.

BitReservoirInit_MP3

Initializes all elements of the bit reservoir state structure.

```
IppStatus ippsBitReservoirInit_MP3 (IppMP3BitReservoir *pDstBitResv,
    IppMP3FrameHeader *pFrameHeader);
```

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 the function `BitReservoirInit_MP3` since both are used to generate the bit reservoir initialization parameters. All other frame header parameters are ignored by the bit reservoir initialization function. Only MPEG-1 (*id* `= 1`) is supported.
pDstBitResv	Pointer to the initialized `IppMP3BitReservoir` state structure. The structure element `BitsRemaining` is initialized as 0. The structure element `MaxBits` is initialized to reflect the maximum number of bits that can be contained in the reservoir at the start of any given frame. The appropriate value of `MaxBits` is directly determined by the selected algorithm (MPEG-1 or MPEG-2) and the stream bit rate indicated by the rate index parameter *pFrameHeader*`.bitRate`.

Discussion

The function is declared in the `ippac.h` file. This function initializes all elements of the bit reservoir state structure based on the coding algorithm (MPEG-1 or MPEG-2) and the average per frame bit allocation specified in the frame header.

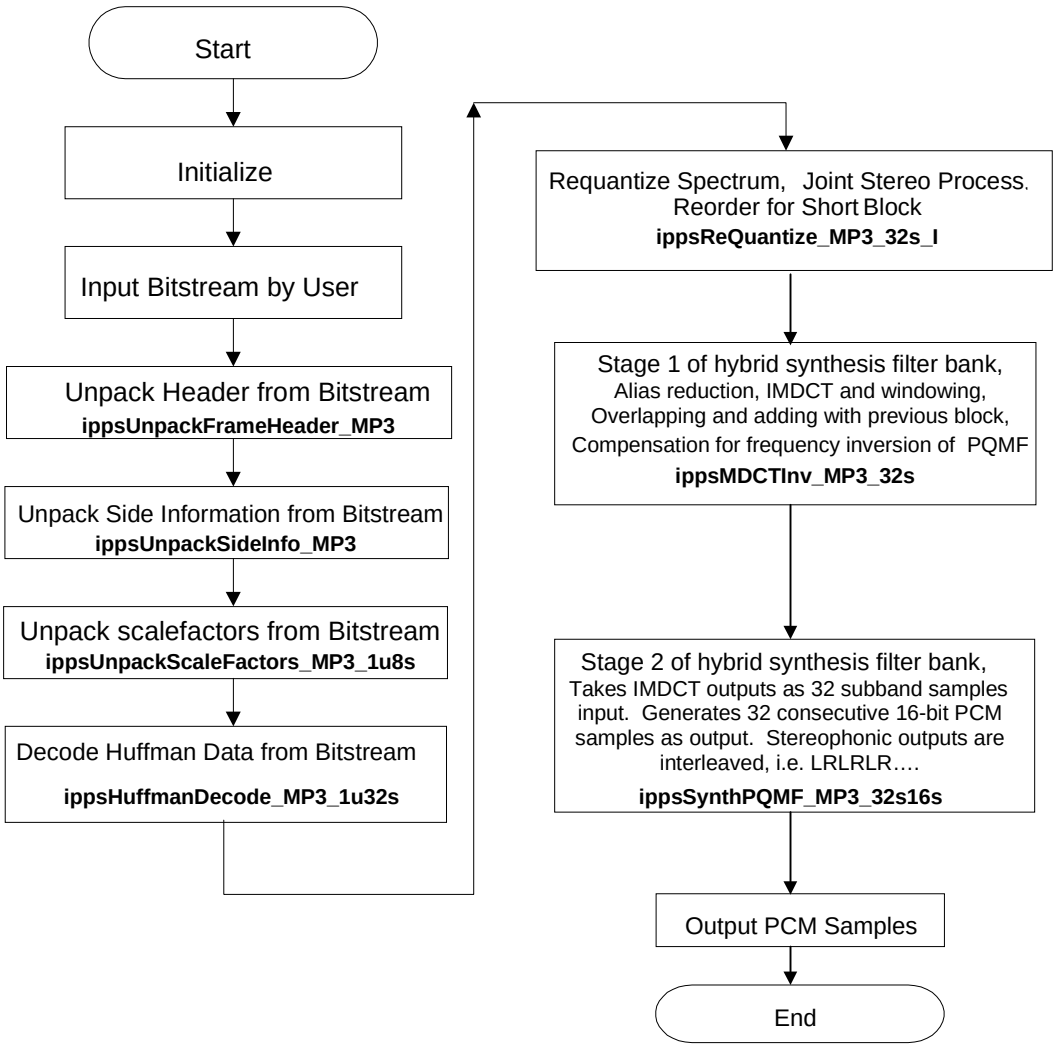
Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pDstBitResv`* or *`pFrameHeader`* is `NULL`.
`ippStsMP3FrameHeaderErr`	Indicates an error when *`pFrameHeader->id`* `!= _IPP_MP3_ID_MPEG1`.

MP3 Audio Decoder

This section describes Intel IPP functions for assembling audio decoders compliant with the layer III portions of the ISO/IEC 11172-3 MPEG-1 and ISO/IEC 13818-3 MPEG-2 audio specifications (see [ISO]) and provides a reference guide to the Intel IPP MP3 Application Programming Interface (API). As shown in Figure 10-3, the MP3 API consists of seven functions, two data structures, and several pre-defined constants and macros.

Figure 10-3 MP3 Decoder API



UnpackFrameHeader_MP3

Unpacks audio frame header.

```
IppStatus ippsUnpackFrameHeader_MP3(Ipp8u** ppBitStream,
    IppMP3FrameHeader* pFrameHeader);
```

Arguments

ppBitStream	Pointer to the pointer to the first byte of the MP3 frame header (**ppBitstream* will be updated in the function).
pFrameHeader	Pointer to the MP3 frame header structure.

Discussion

The function ippsUnpackFrameHeader_MP3 is declared in the ippac.h file. This function unpacks the audio frame header. If cyclic redundancy check (CRC) is enabled, this function also unpacks the CRC word.

Before calling ippsUnpackFrameHeader_MP3, you should locate the bit stream synchronization word and ensure that **ppBitStream* points to the first byte of the 32-bit frame header.

If CRC is enabled, the assumption is that the 16-bit CRC word is adjacent to the 32-bit frame header, as defined in the MP3 standard. Before returning to the caller, the function updates the pointer **ppBitStream* so that it references the next byte after the frame header or the CRC word.

The first byte of the 16-bit CRC word is stored in *pFrameHeader*->CRCWord(15:8), and the second byte is stored in *pFrameHeader*->CRCWord(7:0).

The function does not detect corrupted frame headers.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *ppBitStreamp*, *FrameHeader*, or *ppBitStream* is NULL.

UnpackSideInfo_MP3

Unpacks side information from input bitstream for use during decoding of associated frame.

```
IppStatus ippsUnpackSideInfo_MP3(Ipp8u** ppBitStream,
    IppMP3SideInfo* pDstSideInfo, int* pDstMainDataBegin,
    int* pDstPrivateBits, int* pDstScfsi,  IppMP3FrameHeader* pFrameHeader);
```

Arguments

ppBitStream	Pointer to the pointer to the first byte of the side information associated with the current frame in the bit stream buffer. The function updates this parameter.
pFrameHeader	Pointer to the structure that contains the unpacked MP3 frame header. The header structure provides format information about the input bitstream. Both single- and dual-channel MPEG-1 and MPEG-2 modes are supported.
pDstSideInfo	Pointer to the MP3 side information structure. The structure contains side information that applies to all granules and all channels for the current frame. One or more of the structures are placed contiguously in the buffer pointed to by *pDstSideInfo* in the following order: `{granule 0 (channel 0, channel 1), granule 1 (channel 0, channel 1)}`.
pDstMainDataBegin	Pointer to the `main_data_begin` field.
pDstPrivateBits	Pointer to the `private bits` field.
pDstScfsi	Pointer to the scale factor selection information associated with the current frame. The data is organized contiguously in the buffer pointed to by *pDstScfsi* in the following order: `{channel 0 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3), channel 1 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3)}`.

Discussion

The function ippsUnpackSideInfo_MP3 is declared in the ippac.h file. This function unpacks the side information from the input bitstream. Before calling the function ippsUnpackSideInfo_MP3, make sure that the pointer *ppBitStream points to the first byte of the bit stream that contains the side information associated with the current frame. Before returning to the caller, the function updates the pointer *ppBitStream so that it references the next byte after the side information.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pDstSideInfo*, *pDstMainDataBegin*, *pDstPrivateBits*, *pDstScfsi*, *pFrameHeader*, or *ppBitStream* is NULL.
ippStsMP3FrameHeaderErr	Indicates an error when some elements in the MP3 frame header structure are invalid: *pFrameHeader*->id != 0 *pFrameHeader*->id != 1 *pFrameHeader*->layer != 1 *pFrameHeader*->mode < 0 *pFrameHeader*->mode > 3.
ippStsMP3SideInfoErr	Indicates an error when the value of *block_type* is zero when *window_switching_flag* is set.

UnpackScaleFactors_MP3

Unpacks scalefactors.

```
IppStatus ippsUnpackScaleFactors_MP3_1u8s(Ipp8u** ppBitStream,
    int* pOffset, Ipp8s* pDstScaleFactor, IppMP3SideInfo* pSideInfo,
    int* pScfsi, IppMP3FrameHeader* pFrameHeader, int granule,
    int channel);
```

Arguments

ppBitStream	Pointer to the pointer to the first bitstream buffer byte associated with the scale factors for the current frame, granule, and channel. The function updates this parameter.
pOffSet	Pointer to the next bit in the byte referenced by **ppBitStream*. Valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit. The function updates this parameter.
pSideInfo	Pointer to the MP3 side information structure associated with the current granule and channel.
pScfsi	Pointer to scale factor selection information for the current channel.
channel	Channel index. Can take the values of either 0 or 1.
granule	Granule index. Can take the values of either 0 or 1.
pFrameHeader	Pointer to MP3 frame header structure for the current frame.
pDstScaleFactor	Pointer to the scale factor vector for long and/or short blocks.

Discussion

The function ippsUnpackScaleFactors_MP3 is declared in the ippac.h file. This function unpacks short and/or long block scalefactors for one granule of one channel and places the results in the vector *pDstScaleFactor*. Before returning to the caller, the function updates **ppBitStream* and **pOffset* so that they point to the next available bit in the input bitstream.

NOTE. *MPEG-2 intensity mode: if the intensity position is equal to the maximum value, or illegal position, the illegal position sets to negative. Consequently, in the requantization module, negative positions indicate illegal positions. Scalefactors that are not treated as intensity positions must be set to positive before using them.*

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`ppBitStream`*, *`pOffset`*, *`pDstScaleFactor`*, *`pSideInfo`*, *`pScfsi`*, *`ppBitStream`*, or *`pFrameHeader`* is `NULL`.
`ippStsBadArgErr`	Indicates an error when • *`pOffset`* `> 7` • *`pOffset`* `< 0` • granule and/or channel exceeds `[0, 1]`.
`ippStsMP3FrameHeaderErr`	Indicates an error when • *`pFrameHeader->id`* exceeds `[0, 1]` • *`pFrameHeader->modeExt`* exceeds `[0..3]`.
`ippStsMP3SideInfoErr`	Indicates an error when • *`pSideInfo->blockType`* exceeds `[0, 3]` • *`pSideInfo->mixedBlock`* exceeds `[0, 1]` • *`pSideInfo->sfCompress`* exceeds `[0, 15]` or *`pScfsi`*`[0..3]` exceeds `[0, 1]` when *`pFrameHeader`* indicates the bitstream is MPEG-1 • *`pSideInfo->sfCompress`* exceeds `[0, 511]` when *`pFrameHeader`* indicates the bitstream is MPEG-2.

HuffmanDecode_MP3
HuffmanDecodeSfb_MP3
HuffmanDecodeSfbMbp_MP3

Decodes Huffman data.

```
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);
```

Arguments

ppBitStream	Pointer to the pointer to the first bit stream byte that contains the Huffman code words associated with the current granule and channel. The function updates this parameter.
pOffset	Pointer to the starting bit position in the bit stream byte pointed to by **ppBitStream*. The parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit, and 7 corresponds to the least significant bit. The function updates this parameter.
pSideInfo	Pointer to MP3 structure containing side information associated with the current granule and channel.
pFrameHeader	Pointer to MP3 structure containing the header associated with the current frame.

pDstIs	Pointer to the vector of decoded Huffman symbols used to compute the quantized values of the 576 spectral coefficients associated with the current granule and channel.
pDstNonZeroBound	Pointer to the spectral region above which all coefficients are set to zero.
hufSize	The number of Huffman code bits associated with the current granule and channel.
pSfbTableLong	Pointer to the scale factor bands table for a long block.
pSfbTableShort	Pointer to scale factor band table for short block. You can use the default table from MPEG-1, MPEG-2 standards.
pMbpTable	Pointer to the mixed block partition table.

Discussion

The functions `ippsHuffmanDecode_MP3`, `ippsHuffmanDecodeSfb_MP3`, and `ippsHuffmanDecodeSfbMbp_MP3_1u32s` are declared in the `ippac.h` file. These functions decode Huffman symbols for the 576 spectral coefficients associated with one granule of one channel. Before returning to the caller, the function updates **ppBitStream* so that it points to the byte in the bitstream that contains the first new bit following the decoded block of Huffman codes.

The function `ippsHuffmanDecodeSfb_MP3` makes the use of a predefined scale factor bands table possible when working with a long block. Alternatively, you can use the default table from MPEG-1, or MPEG-2 standards.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, *pDstIs*, *pDstNonZeroBound*, *pSideInfo*, *pFrameHeader*, or *ppBitStream* is `NULL` or when *pOffset* `< 0` or *pOffset* `> 7`.
`ippStsMP3FrameHeaderErr`	Indicates an error when some elements in the MP3 frame header structure are invalid.

ippStsMP3SideInfoErr	Indicates an error when some elements in the MP3 side information structure are invalid or when *hufSize* < 0 or *hufSize* > *pSideInfo->part23Len*.
ippStsErr	Indicates unknown error.

ReQuantize_MP3
ReQuantizeSfb_MP3

Requantizes the decoded Huffman symbols.

```
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);
```

Arguments

pSrcDstIsXr	Pointer to the vector of the decoded Huffman symbols. For stereo and dual channel modes, right channel data begins at the address &(*pSrcDstIsXr*[576]). The function updates this vector.
pNonZeroBound	Pointer to the spectral bound above which all coefficients are set to zero. For stereo and dual channel modes, the left channel bound is pNonZeroBound [0], and the right channel bound is pNonZeroBound [1].
pScaleFactor	Pointer to the scale factor buffer. For stereo and dual channel modes, the right channel scalefactors begin at & (*pScaleFactor* [IPP_MP3_SF_BUF_LEN]).
pSideInfo	Pointer to the side information for the current granule.

pFrameHeader	Pointer to the frame header for the current frame.
pBuffer	Pointer to the workspace buffer. The buffer length must be 576 samples.
pSfbTableLong	Pointer to the scale factor bands table for a long block.
pSfbTableShort	Pointer to the scale factor bands table for a short block.

Discussion

The functions ippsReQuantize_MP3 and ippsReQuantizeSfb_MP3 are declared in the ippac.h file. These functions requantize the decoded Huffman symbols. Spectral samples for the synthesis filter bank are derived from the decoded symbols using the requantization equations given in the ISO standard.

If necessary, you can apply stereophonic mid/side (M/S) and/or intensity decoding. The function returns requantized spectral samples in the vector *pSrcDstIsXr*.

For short blocks you can perform reordering operation. You should preallocate a workspace buffer pointed to by *pBuffer* prior to calling the requantization function. For short blocks, the value pointed to by *pNonZeroBound* is recalculated according to the reordered sequence.

You can use predefined scale factor bands table for a long or short block with the function ippsReQuantizeSfb_MP3. Alternatively, you can use the default table from MPEG-1, or MPEG-2 standards.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrcDstIsXr*, *pNonZeroBound*, *pScaleFactor*, *pSideInfo*, *pFrameHeader*, or *pBuffer* is NULL.
ippStsBadArgErr	Indicates an error when *pNonZeroBound* exceeds [0, 576].

ippStsMP3FrameHeaderErr	Indicates an error when *pFrameHeader->id* exceeds [0, 1], *pFrameHeader->samplingFreq* exceeds [0, 2], *pFrameHeader->mode* exceeds [0, 3], *pFrameHeader->modeExt* exceeds [0, 3].
ippStsMP3SideInfoErr	Indicates an error when the bitstream is in the stereo mode, but the block type of left is different from that of right, *pSideInfo*[ch].blockType exceeds [0, 3], *pSideInfo*[ch].mixedBlock exceeds [0, 1], *pSideInfo*[ch].globGain exceeds [0, 255], *pSideInfo*[ch].sfScale exceeds [0, 1], *pSideInfo*[ch].preFlag exceeds [0, 1], *pSideInfo*[ch].pSubBlkGain[w] exceeds [0, 7], where ch is within the range of 0 to 1, and *w* is within the range of 0 to 2.
ippStsErr	Indicates an unknown error.

MDCTInv_MP3

Performs the first stage of hybrid synthesis filter bank.

```
IppStatus ippsMDCTInv_MP3_32s(Ipp32s* pSrcXr, Ipp32s* pDstY,
    Ipp32s* pSrcDstOverlapAdd, int nonZeroBound,  int* pPrevNumOfImdct,
    int blockType, int mixedBlock);
```

Arguments

pSrcXr	Pointer to the vector of requantized spectral samples for the current channel and granule, represented in Q5.26 format.
pDstY	Pointer to the vector of IMDCT outputs in Q7.24 format for input to PQMF bank.

pSrcDstOverlapAdd	Pointer to the overlap-add buffer. Contains the overlapped portion of the previous granule IMDCT output in Q7.24 format. The function updates this buffer.
nonZeroBound	Limiting bound for spectral coefficients. All spectral coefficients exceeding this boundary become zero for the current granule and channel.
pPrevNumOfImdct	Pointer to the number of IMDCTs computed for the current channel of the previous granule. The function updates this parameter so that it references the number of IMDCTs for the current granule.
blockType	Block type indicator.
mixedBlock	Mixed block indicator.

Discussion

The function `ippsMDCTInv_MP3` is declared in the `ippac.h` file. This function performs the first stage of the hybrid synthesis filter bank. The following operations are performed:

- alias reduction
- inverse modified discrete cosine transform (IMDCT) according to block size specifiers and mixed block modes
- overlap add of IMDCT outputs
- frequency inversion prior to pseudo-quadrature mirror synthesis filter (PQMF) bank.

Since IMDCT is a lapped transform, you should preallocate a buffer referenced by *pSrcDstOverlapAdd* to maintain the IMDCT overlap-add state.

The buffer must contain 576 elements. Prior to the first call to the synthesis filter bank, all elements of the overlap-add buffer should be set to zero. In between all subsequent calls, the MP3 application must preserve the contents of the overlap-add buffer.

Upon entry to `ippsMDCTInv_MP3_32s`, the overlap-add buffer should contain the IMDCT output generated by operating on the previous granule. Upon exit from `ippsMDCTInv_MP3_32s`, the overlap-add buffer contains the overlapped portion of the output generated by operating on the current granule.

Upon return from the function, the IMDCT sub-band output samples are organized as follows:

pDstY[*j**32+subband], for *j* = 0 to 17, subband = 0 to 31.

Q5.26 designates that 5 bits before and 26 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q7.24 designates that 7 bits before and 24 bits after fixed point position are used to present a 32-bit value in the fixed point format.

NOTE. *Note that the pointers pSrcXr and pDstY must reference different buffers.*

Return Value

ippStsNoErr	Indicates no error.
ippStsBadArgErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsErr	Indicates an error when one or more of the following input data errors are detected: either *blockType* exceeds [0,3], or *mixedBlock* exceeds [0,1], or *nonZeroBound* exceeds [0,576], or **pPrevNumOfImdct* exceeds [0,32].

SynthPQMF_MP3

Performs the second stage of hybrid synthesis filter bank.

```
IppStatus ippsSynthPQMF_MP3_32s16s(Ipp32s* pSrcY, Ipp16s* pDstAudioOut,
    Ipp32s* pVBuffer, int* pVPosition, int mode);
```

Arguments

pSrcY	Pointer to the block of 32 IMDCT sub-band input samples in Q7.24 format.
pDstAudioOut	Pointer to the block of 32 reconstructed PCM output samples in 16-bit signed little-endian format. Left and right channels are interleaved according to the *mode* flag.
pVBuffer	Pointer to the input workspace buffer containing Q7.24 data. The function updates this parameter.
pVPosition	Pointer to the internal workspace index. The function updates this parameter.
mode	Flag that indicates whether or not PCM audio output channels should be interleaved. 1 indicates not interleaved, 2 indicates interleaved.

Discussion

The function `ippsSynthPQMF_MP3` is declared in the `ippac.h` file. This function performs the second stage of the hybrid synthesis filter bank, that is, a critically-sampled 32-channel PQMF synthesis bank that generates 32 time-domain output samples for each 32-sample input block of IMDCT outputs.

For each input block, the PQMF generates an output sequence of 16-bit signed little-endian PCM samples in the vector pointed to by *pDstAudioOut*.

If *mode* equals to 2, the left and right channel output samples are interleaved, that is, LRLRLR, so that the left channel data is organized as follows:

pDstAudioOut `[2*i]`, *i* = 0 to 31.

If *mode* equals 1, then the left and right channel outputs are not interleaved.

Since PQMF bank contains memory, the MP3 application must maintain two state variables in between calls to the function.

The application must preallocate a workspace buffer of size 512x (Number of Channels) for the PQMF computation. This buffer is referenced by the pointer *pVBuffer* and its elements should be initialized to zero prior to the first call. During subsequent calls, the *pVBuffer* input for the current call should contain the *pVbuffer* output generated by the previous call.

In addition to *pVBuffer*, the MP3 application must also initialize to zero and thereafter preserve the value of the state variable *pVPosition*. The MP3 application should modify the values contained in *pVBuffer* or *pVPosition* only during decoder reset, and the reset values should always be zero.

Return Value

ippStsNoErr	Indicates no error.
ippNullPtrErr	Indicates an error when least one of the pointers *pSrcY*, *pDstAudioOut*, *pVBuffer*, or *pVPosition* is NULL.
ippStsBadArgErr	Indicates an error when at least one of the specified pointers is NULL, or the value of *mode* is less than 1 or more than 2, or when the value of *pVPosition* exceeds [0, 15].
ippStsErr	Indicates an unknown error.

Advanced Audio Coding Functions

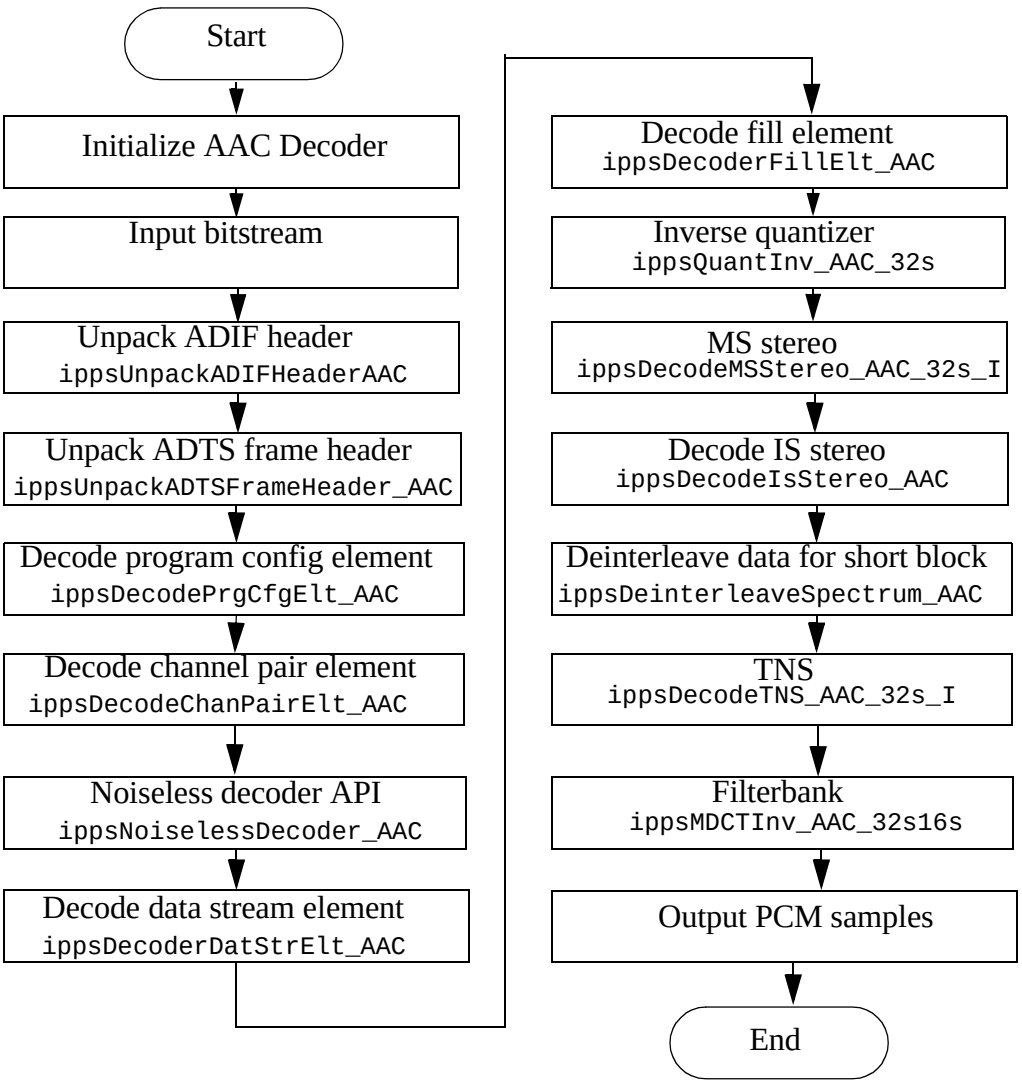
The ISO/IEC 13818-7 MPEG-2 AAC (Advanced Audio Coding) algorithm is an efficient coding method for surround signals like 5-channel signals (left, right, center, left surround, right surround).

MPEG-2 AAC supports up to 48 main audio channels with the sampling frequency between 8kHz and 96kHz. MPEG formal tests have shown that for 5-channel audio signals, AAC satisfies the ITU-R quality requirements and provides slightly better audio quality at 320 kilobits per second (kbps) than MPEG-2 BC (Backwards Compatibility) provides at 640 kbps.

Due to its high coding efficiency, AAC is a prime candidate for any digital broadcasting system and has been selected by the DRM (Digital Radio Mondiale) system. AAC also plays a major role for the delivery of high quality music via the Internet. Moreover, with some modifications, AAC is the only high quality audio coding scheme adopted within the MPEG-4 standard, the future “global multimedia language”.

The AAC decoder API provides a variety of AAC LC decoder functions, including bitstream unpacking and AAC core decoding functions. This provides customers great flexibility in configuring the decoder system. See ISO/IEC 13818-7:1997.

Figure 10-4 AAC Flowchart



Global Macros

Table 10-4 shows the definitions of global macros.

Table 10-4 Global Macro Definitions

Global Macro Name	Definition	Notes
IPP_AAC_FRAME_LEN	1024	size of data in one frame
IPP_AAC_SF_LEN	120	scale factor buffer length
IPP_AAC_GROUP_NUM_MAX	8	maximum group number for one frame
IPP_AAC_TNS_COEF_LEN	60	TNS coefficients buffer length
IPP_AAC_TNS_FILT_MAX	8	maximum TNS filter number for one frame
IPP_AAC_PRED_SFB_MAX	41	maximum prediction scale factor bands number for one frame
IPP_AAC_ELT_NUM	16	maximum number of elements for one program
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

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 scale factor bands */
    int     sfGrouping;        /* scale factor grouping information */
    int     predDataPres;      /* prediction data present flag for one
                                  frame, 0: prediction off, 1: prediction
                                  on */
    int     predReset;         /* prediction reset flag, 0: reset off,
                                  1: reset on */
    int     predResetGroupNum; /* prediction reset group number */
    Ipp8u pPredUsed[IPP_AAC_PRED_SFB_MAX+3];/* prediction flag buffer
                                    for each scale factor band: 0:
                                    off, 1: on buffer
                                    length 44 bytes, 4-byte align*/
    /* decoded from the above info */
    int     numWinGrp;         /* group number */
    int     pWinGrpLen[IPP_AAC_GROUP_NUM_MAX];/* buffer for number of
                                          windows in each group */
} IppAACIcsInfo;
```

AAC Scalable Main Element Header

```
typedef struct{
    int windowSequence; //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 scale factor band */
} IppAACChanPairElt;
```

Channel Information

```
typedef struct {
    int     tag;
    int     id;             /* element id */
    int     samplingRateIndex; /* sampling rate index */
    int     predSfbMax;     /* maximum prediction scale factor bands */
    int     preWinShape;    /* previous block window shape */

    int     winLen;         /* 128: 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 scale factor
offset
                                      in each section */
    int pMaxSect[IPP_AAC_GROUP_NUM_MAX];  /* maximum section number
                                          for each group */
    int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX];/*TNS number filter
                                             number buffer*/
    /* TNS data */
    int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX];/* TNS coefficients
                                                  resolution flag */
    int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX];  /* TNS filter length */
    int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX];  /* TNS filter order */
    int pTnsDirection[IPP_AAC_TNS_FILT_MAX];  /* TNS filter direction
                                                 flag */
}IppAACChanInfo;
```

MPEG-2 AAC Primitives

In the following sections, the parameter *maxSfb* designates the number of scale factor bands transmitted per group and unpacked from the bitstream. The parameter *numSwb* stands for the number of scale factor window bands for the short block or the number of scale factor window bands for the long block. The values are computed according to the sampling rate and the block type.

See clause 8.3.1 of ISO/IEC 13818-7:1997.

UnpackADIFHeader_AAC

Gets the AAC ADIF format header.

```
IppStatus ippsUnpackADIFHeader_AAC (Ipp8u **ppBitStream, IppAACADIFHeader
    *pADIFHeader, IppAACPrgCfgElt *pPrgCfgElt, int prgCfgEltMax);
```

Arguments

ppBitStream	Double pointer to the current byte before the ADIF header.
prgCfgEltMax	Maximum program configure element number. Must be within the range of [1, 16].
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.

Discussion

This function is declared in the ippac.h file. The function gets the AAC ADIF format header, including program configuration elements from the input bitstream. See Table 6.2, and 6.21. of ISO/IEC 13818-7:1997.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pADIFHeader*, *pPrgCfgElt*, or **ppBitStream* is NULL.
ippStsAacPrgNumErr	Indicates an error when the decoded *pADIFHeader->numPrgCfgElt > prgCfgEltMax*, or *prgCfgEltMax* is outside the range of [1,IPP_AAC_MAX_ELT_NUM].

NOTE. `pADIFHeader->numPrgCfgElt` *is the number directly unpacked from bitstream plus 1.*

UnpackADTSFrameHeader_AAC

Gets ADTS frame header from the input bitstream.

```
IppStatus ippsUnpackADTSFrameHeader_AAC (Ipp8u **ppBitStream,
    IppAACADTSFrameHeader *pADTSFrameHeader);
```

Arguments

ppBitStream	Double pointer to the current byte.
ppBitStream	Double pointer to the current byte after unpacking the ADTS frame header.
pADTSFrameHeader	Pointer to the `IppAACADTSFrameHeader` structure.

Discussion

The function gets ADTS frame header from the input bitstream. If the CRC word is applied, the first byte of the 16-bit CRC word is stored in `pADTSFrameHeader->CRCWord[15:8]` and the second byte is stored in `pADTSFrameHeader->CRCWord[7:0]`. It does not check whether the header is corrupt.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pOffest*, *pPrgCfgElt*, or **ppBitStream* is `NULL`.

ippStsAacAdtsSyncWordErr	Indicates an error when the syncword error occurs.

DecodePrgCfgElt_AAC

Gets program configuration element from the input bitstream.

```
IppStatus ippsDecodePrgCfgElt_AAC (Ipp8u **ppBitStream, int *pOffset,
    IppAACPrgCfgElt *pPrgCfgElt);
```

Arguments

ppBitStream	Double pointer to the current byte.
pOffset	Pointer to the bit position in the byte pointed to by **ppBitStream*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
ppBitStream	Double pointer to the current byte after decoding the program configuration element.
pOffset	Pointer to the bit position in the byte pointed to by **ppBitStream*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
pPrgCfgElt	Pointer to IppAACPrgCfgElt structure.

Discussion

This function is declared in the ippac.h file. The function gets the program configuration element from the input bitstream. See clause 8.5 and Table 6.21 of ISO/IEC 13818-7.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, *pPrgCfgElt*, or **ppBitStream* is NULL.
ippStsAacBitOffsetErr	Indicates an error when *pOffset* is out of the range of [0,7].

DecodeChanPairElt_AAC

Gets channel_pair_element from the input bitstream.

```
IppStatus ippsDecodeChanPairElt_AAC (Ipp8u **ppBitStream, int *pOffset,
    IppAACIcsInfo *pIcsInfo, IppAACChanPairElt *pChanPairElt, int predSfbMax);
```

Arguments

ppBitStream	Double pointer to the current byte.
pOffset	Pointer to the bit position in the byte pointed to by **ppBitStream*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
predSfbMax	Maximum prediction scale factor bands. For LC profile, set *predSfbMax* = 0.
ppBitStream	Double pointer to the current byte after decoding the channel pair element.
pOffset	Pointer to the bit position in the byte pointed to by **ppBitStream*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
pIcsInfo	Pointer to IppAACIcsInfo structure. If pIcsInfo->predDataPres = 0, set *pIcsInfo->predReset* = 0.

	Only the first *pIcsInfo->numWinGrp* elements in *pIcsInfo->* *pWinGrpLen* are meaningful. You should not change some members of the structure, as shown in Table 10-5.
pChanPairElt	Pointer to IppAACChanPairElt structure. You should not change some members of the structure, as shown in Table 10-6.

Discussion

This function is declared in the ippac.h file. The function gets the channel pair element from the input bitstream. Individual channel stream is not included.

If *common_window* flag decoded from the input bitstream is 0, all members of *pIcsInfo* and *pChanPairElt* remain unchanged except for *pChanPairElt->commonWin*.

See clause 8.3 and Table 6.10, 6.11 of ISO/IEC 13818-7:1997.

Table 10-5 Unchanged Members of *pIcsInfo*

Unchanged Members	Conditions
sfGrouping	*pIcsInfo->winSequence* != 2
predResetGroupNum	*pIcsInfo->predDataPres* == 0 || *pIcsInfo->predReset* == 0
pPredUsed[sfb]	*pIcsInfo->predDataPres* == 0

Table 10-6 Unchanged Members of *pChanPairElt*

Members	Conditions
pMsUsed[sfb]	*pChanPairElt*->msMaskPres != 1

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, **ppBitStream*, *pIcsInfo*, or *pChanPairElt* is NULL.
ippStsAacBitOffsetErr	Indicates an error when *pOffset* is out of the range of [0,7].
ippStsAacMaxSfbErr	Indicates an error when *pIcsInfo->maxSfb* decoded from bitstream is greater than IPP_AAC_MAX_SFB, the maximum scale factor band for all sampling frequencies.

NoiselessDecoder_LC_AAC

Decodes all data for one channel.

```
IppStatus ippsNoiselessDecoder_LC_AAC (Ipp8u **ppBitStream, int *pOffset, int
    commonWin, IppAACChanInfo *pChanInfo, Ipp16s *pDstScalefactor, Ipp32s
    *pDstQuantizedSpectralCoef, Ipp8u *pDstSfbCb, Ipp8s *pDstTnsFiltCoef);
```

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.
ppBitStream	Double pointers to bitstream buffer.
pOffset	Pointer to the offset in one byte.
pChanInfo	Pointer to the channel information. IppAACChanInfo structure. Denotes *pIcsInfo* as *pChanInfo*->pIcsInfo as shown in Table 10-7.

pDstScalefactor	Pointer to the scale factor or intensity position buffer. Buffer length is more than or equal to 120. Only *maxSfb* elements are stored for each group. There is no space between sequence groups.
pDstQuantizedSpectralCoef	Pointer to the quantized spectral coefficients data. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length is more than or equal to 1024.
pDstSfbCb	Pointer to the scale factor band codebook. Buffer length must be more than or equal to 120. Store *maxSfb* elements for each group. There is no space between the sequence groups.
pDstTnsFiltCoef	Pointer to TNS coefficients. Buffer length must be more than or equal to 60. The store sequence is TNS order elements for each filter for each window. The elements are not changed if the corresponding TNS order is zero.

Discussion

This function is declared in the `ippac.h` file. The function decodes all data for one channel, including scale factor/intensity positions, spectral coefficients, TNS coefficients, and associated side information for LC profile.

You need to set *pChanInfo*`->pIcsInfo`, *pChanInfo->samplingRateIndex*, *pChanInfo*`->predSfbMax` to correct pointer/values before calling this function.

Table 10-7 Input/Output Members List of *pChanInfo*

Member	Output
Tag	Not used.
id	Not used.
preWinShape	Not used.
pChanPairElt	Not used.
samplingRateIndex	As input. Not changed.

Table 10-7 Input/Output Members List of *pChanInfo*

Member	Output
predSfbMax	As input. Must be 0. Not changed.
winLen	As output. Set to 128, if decoded, *pIcsInfo->winSequence* is short block. Otherwise set to 1024.
numWin	As output. Set to 8, if decoded pIcsInfo-*>winSequence* is short block. Otherwise set to 1.
numSwb	As output. Set to the maximum number of scale factor window bands in each group according to *samplingRateIndex* and *pIcsInfo->winSequence*. See Table 8.4-8.1 of ISO/IEC 13818-7:1997.
globGain *pulseDataPres* *tnsDataPres* *gainContrDataPres*	As output. Unpacked from bitstream.
pMaxSect	As output. Pointer to the maximum of sections number in each group. Only *pIcsInfo->numWinGrp* elements in the buffer are meaningful.
pSectCb	As output. Pointer to the section codebook. Only *pMaxSect*[g] elements are stored for each group. There is no space between the sequence groups.
pTnsRegionLen	As output. Pointer to the length of the region in units of scale factor bands to which one filter is applied in each window.
pTnsFiltOrder	As output. Pointer to the order of the temporal noise shaping filter applied to each window.
pTnsDirection	As output. Pointer to the token that indicates whether the filter is applied in the upward or downward direction. 0 stands for upward and 1 for downward.
pIcsInfo	As input if *commonWin* == 1. As output if *commonWin* == 0. If *pIcsInfo->predDataPres* == 0, set *pIcsInfo->predReset* = 0. Only the first *pIcsInfo->numWinGrp* elements in *pIcsInfo->pWinGrpLen* are meaningful. Under specific conditions, some members of the structure must remain unchanged. See Table 10-5.

Return Values

ippStsNoErr	Indicates no error.

`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *`ppBitStream`*, *`pOffset`*, *`pChanInfo`*, *`pDstScalefactor`*, *`pDstQuantizedSpectralCoef`*, *`pDstSfbCb`*, *`pDstTnsFiltCoef`*, *`pChanInfo->pIcsInfo`*, or *`*ppBitStream`* is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when *`*pOffset`* is out of range `[0,7]`.
`ippStsAacComWinErr`	Indicates an error when *`commonWin`* exceeds `[0,1]`.
`ippStsAacSmplRateIdxErr`	Indicates an error when *`pChanInfo->samplingRateIndex`* exceeds `[0,11]`.
`ippStsAacPredSfbErr`	Indicates an error when *`pChanInfo->predSfbMax`* is not equal to 0.
`ippStsAacMaxSfbErr`	Indicates an error when *`pChanInfo->pIcsInfo->maxSfb > pChanInfo->numSwb`*.
`ippStsAacSectCbErr`	Indicates an error when the codebook pointed to by *`pChanInfo->pSectCb`* is illegal or when *`*(pChanInfo->pSectCb)`*`==12, 13`.
	If the current channel is not the right channel of the channel pair element, *`*pSectCb`* `= 14, 15` is also illegal.
`ippStsAacPlsDataErr`	Indicates an error when the *`pChanInfo->pIcsInfo->winSequence`* indicates a short sequence and *`pChanInfo->pulsePres`* indicates pulse data present.
	The start scale factor band for pulse data `>=` *`pChanInfo->numSwb`*, or pulse data position *`offset >= winLen`*.
`ippStsAacGainCtrErr`	Indicates an error when *`pChanInfo->gainControlPres`* is decoded as 1, which means that gain control data is present. Gain control data is not currently supported.
`ippStsAacCoefValErr`	Indicates an error when the quantized coefficients value pointed to by *`pDstCoef`* exceeds range `[-8191, 8191]`.

DecodeDatStrElt_AAC

Gets data stream element from the input bitstream.

```
IppStatus ippsDecodeDatStrElt_AAC (Ipp8u **ppBitStream, int *pOffset,
    int *pDataTag, int *pDataCnt, Ipp8u * pDstDataElt);
```

Arguments

ppBitStream	Double pointer to the current byte.
pOffset	Pointer to the bit position in the byte pointed to by **ppBitStream*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
ppBitStream	Double pointer to the current byte after the decode data stream element.
pOffset	Pointer to the bit position in the byte pointed to by **ppBitStream*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
pDataTag	Pointer to *element_instance_tag*. See Table 6.20 of ISO/IEC 13818-7:1997.
pDataCn	Pointer to the value of data length in bytes.
pDstDataElt	Pointer to the data stream buffer that contains the data stream extracted from the input bitstream. There are 512 elements in the buffer pointed to by *pDstDataElt*.

Description

This function is declared in the `ippac.h` file. The function gets data stream element from the input bitstream.

See clause 8.6 and Table 6.20 of ISO/IEC 13818-7:1997.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *`ppBitStream`*, *`pOffset`*, `*`*`ppBitStream`*, *`pDataTag`*, *`pDataCnt`*, or *`pDstDataElt`* is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when `*`*`pOffset`* is out of range `[0,7]`.

DecodeFillElt_AAC

Gets the fill element from the input bitstream.

```
IppStatus ippsDecodeFillElt_AAC (Ipp8u **ppBitStream, int *pOffset,
    int *pFillCnt, Ipp8u * pDstFillElt);
```

Arguments

`ppBitStream`	Pointer to the pointer to the current byte.
`pOffset`	Pointer to the bit position in the byte pointed to by `*`*`ppBitStream`*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
`ppBitStream`	Pointer to the pointer to the current byte after the decode fill element.
`pOffset`	Pointer to the bit position in the byte pointed to by `*`*`ppBitStream`*. Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
`pFillCnt`	Pointer to the value of the length of total fill data in bytes.
`pDstFillElt`	Pointer to the fill data buffer whose length must be equal to or greater than 270.

Discussion

This function is declared in the `ippac.h` file. The function gets the fill element from the input bitstream.

See clause 8.7 and Table 6.22 of ISO/IEC 13818-7:1997.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *`ppBitStream`*, *`pOffset`*, `*`*`ppBitStream`*, *`pFillCnt`*, or *`pDstFillElt`* is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when `*`*`pOffset`* is out of the range of `[0,7]`.

QuantInv_AAC

Performs inverse quantization of Huffman symbols for current channel in-place.

```
IppStatus ippsQuantInv_AAC_32s_I (Ipp32s *pSrcDstSpectralCoef, const Ipp16s
    *pScalefactor, int numWinGrp, const int *pWinGrpLen, int maxSfb, const
    Ipp8u *pSfbCb, int samplingRateIndex, int winLen);
```

Arguments

`pSrcDstSpectralCoef`	On input, pointer to the input quantized coefficients. For short block the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. On output, pointer to the destination inverse quantized coefficient in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group.

	Buffer length must be more than or equal to 1024. The maximum error of output *pSrcDstSpectralCoef*[*i*] is listed in Table 10-8.
pScalefactor	Pointer to the scale factor buffer. Buffer length must be more than or equal to 120.
numWinGrp	Group number.
pWinGrpLen	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
maxSfb	Maximal scale factor bands number for the current block.
pSfbCb	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Only *maxSfb* elements for each group are meaningful. There are no spaces between the sequence groups.
samplingRateIndex	Sampling rate index. Valid within [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997.
winLen	Data number in one window.

Discussion

This function is declared in the ippac.h file. The function performs inverse quantization of Huffman symbols for the current channel as shown by the formula.

$$pSrcDst[i] = sign(pSrcDst[i]) * (pSrcDst[i])^{\frac{4}{3}} * 2^{\left(\frac{1}{2}(pScalefactor[sfb] - 100)\right)}$$

See clause 10 of ISO/IEC 13818-7:1997.

Table 10-8 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

Table 10-8 Computation Error List for *pSrcDstSpectralCoef*

Output		Conditions
7	129~8191	< 2 ^ 29

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *`pSrcDstSpectralCoef`*, *`pScalefactor`*, *`pSfbCb`*, *`pWinGrpLen`* is `NULL`.
`ippStsAacSmplRateIdxErr`	Indicates an error when *`pChanInfo->samplingRateIndex`* exceeds `[0, 11]`.
`ippStsAacMaxSfbErr`	Indicates an error when *`maxSfb`* exceeds `[0,IPP_AAC_MAX_SFB]`.
`ippStsAacWinGrpErr`	Indicates an error when *`numWinGrp`* exceeds `[0, 8]` for long window or is not equal to 1 for short window.
`ippStsAacWinLenErr`	Indicates an error when *`winLen`* is not equal to 128 or 1024;
`ippStsAacCoefValErr`	Indicates an error when the quantized coefficients value pointed to by *`pSrcDstSpectralCoef`* exceeds `[-8191,8191]`.

DecodeMsStereo_AAC

Processes mid-side (MS) stereo for pair channels in-place.

```
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);
```

Arguments

pSrcDstL	On input, pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. On output, pointer to left channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
pSrcDstR	On input, pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. On output, pointer to right channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
msMaskPres	MS stereo mask flag. • 0: MS off • 1: MS on • 2: MS all bands on.
pMsUsed	Pointer to the MS Stereo flag buffer. Buffer length must be more than or equal to 120.
pSfbCbPointer	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Store *maxSfb* elements for each group.There is no space between the sequence groups.
numWinGrp	Group number.
pWinGrpLen	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
maxSfb	Maximal scale factor bands number for the current block.

samplingRateIndex	Sampling rate index. Valid within [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997.
winLen	Data number in one window.
pSrcDstR	Pointer to right channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
pSfbCb	Pointer to the scale factor band codebook. If invert_intensity (group, sfb) = -1, and if *pSfbCb = INTERITY_HCB, let *pSfbCb = INTERITY_HCB2. If *pSfbCb = INTERITY_HCB2, let *pSfbCb = INTERITY_HCB. Buffer length must be more than or equal to 120. Store *maxSfb* elements for each group. There is no space between the sequence groups.

Discussion

This function is declared in the ippac.h file. The function performs mid-side (MS) stereo process for pair channels and at the same time runs the invert_intensity(group, sfb) function and stores the values in the *pSfbCb* buffer.

In the case when MS stereo flag is on, perform operation on *pSrcDstL*[i] and *pSrcDstR*[i] as described in the following formula:

$$\begin{aligned} pSrcDstL'[i] &= pSrcDstL[i] + pSrcDstR[i] \\ pSrcDstR'[i] &= pSrcDstL[i] - pSrcDstR[i]. \end{aligned}$$

See clause 12 of ISO/IEC 13818-7:1997.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the pointers *pSrcDstL*, *pSrcDstR*, *pMsUsed*, *pWinGrpLen*, *pSfbCb* is NULL.
ippStsAacMaxSfbErr	Indicates an error when the coefficient index calculated from *samplingFreqIndex* and *maxSfb* exceeds *winLen* in each window.
ippStsAacSamplRateIdxErr	Indicates an error when *pChanInfor->samplingRateIndex* exceeds [0,11].
ippStsAacWinGrpErr	Indicates an error when *numWinGrp* exceeds [0,8] for long window or is not equal to 1 for short window.
ippStsAacWinLenErr	Indicates an error when *winLen* is not equal 128 or 1024.
ippStsStereoMaskErr	Indicates an error when the stereo mask flag is not equal 1 or 2.

DecodeIsStereo_AAC

Processes intensity stereo for pair channels.

```
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);
```

Arguments

pSrcL	Pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
pScalefactor	Pointer to the scale factor buffer. Buffer length must be more than or equal to 120.

pSfbCb	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Store *maxSfb* elements for each group.There is no space between the sequence groups. Respective values of *pSfbCb*[sfb] equal to 1, -1, or 0 indicate the intensity stereo mode, that is, direct, inverse, or none.
numWinGrp	Group number.
pWinGrpLen	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
maxSfbMax	Maximal scale factor bands number for the current block.
samplingRateIndex	Sampling rate index. Valid within [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997.
winLen	Data number in one window.
pDstR	Pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.

Discussion

This function is declared in the ippac.h file. The function processes intensity stereo for pair channels.

You should perform operation on *pSrcL*[*i*], *pDstR*[*i*] according to the following formula.

$$pDstR[i] = pSrc[i] * is\_intensity(g, sfb) * 2^{\left(-\frac{1}{4} pScalefactor[sfb]\right)}$$

NOTE. `invert_intensity(g, sfb)` *is not used in the formula, because it is already decoded and stored in* `pSfbCb[sfb]` *in the MS stereo process primitive. Refer to clause 12 of* ISO/IEC 13818*-7:1997.*

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *`pSrcDstL`*, *`pSrcDstR`*, *`pScaleFactor`*, *`pWinGrpLen`*, or *`pSfbCb`* is `NULL`.
`ippStsAacMaxSfbErr`	Indicates an error when the coefficient index calculated from *`samplingFreqIndex`* and *`maxSfb`* exceeds *`winLen`* in each window.
`ippStsAacSamplRateIdxErr`	Indicates an error when *`pChanInfor->samplingRateIndex`* exceeds `[0,11]`.
`ippStsAacWinGrpErr`	Indicates an error when *`numWinGrp`* exceeds `[0,8]` for long window or is not equal to 1 for short window.
`ippStsAacWinLenErr`	Indicates an error when *`winLen`* is not equal to 128 or 1024.

DeinterleaveSpectrum_AAC

Deinterleaves the coefficients for short block.

```
IppStatus ippsDeinterleaveSpectrum_AAC_32s (const Ipp32s *pSrc, Ipp32s *pDst,
    int numWinGrp, const int *pWinGrpLen, int maxSfb, int samplingRateIndex,
    int winLen);
```

Arguments

pSrc	Pointer to source coefficients buffer. The coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
numWinGrp	Group number.
pWinGrpLen	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
maxSfbMax	Maximal scale factor bands number for the current block.
samplingRateIndex	Sampling rate index. Valid in [0, 11]. See Table 6.5 of ISO/IEC 13818-7:1997.
winLen	Data number in one window.
pDst	Pointer to the output of coefficients. Data sequence is ordered in *pDst*[*w**128+*sfb***sfbWidth*[*sfb*]+*i*], where *w* is window index, *sfb* is scale factor band index, *sfbWidth* is the scale factor band width table, *i* is the index within scale factor band. Buffer length must be more than or equal to 1024.

Discussion

This function is declared in the ippac.h file. The function deinterleaves the coefficients for short block.

See clause 8.3.5 of ISO/IEC 13818-7:1997.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *pSrc*, *pDst*, or *pWinGrpLen* is NULL.
ippStsAacMaxSfbErr	Indicates an error when the coefficient index calculated from *samplingFreqIndex* and *maxSfb* exceeds *winLen* in each window.

ippStsAacSamplRateIdxErr	Indicates an error when *pChanInfor->samplingRateIndex* exceeds [0,11].
ippStsAacWinGrpErr	Indicates an error when *numWinGrp* exceeds [0,8].
ippStsAacWinLenErr	Indicates an error when *winLen* is not equal to 128.

DecodeTNS_AAC

Decodes for Temporal Noise Shaping in-place.

```
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);
```

Arguments

pSrcDstSpectralCoefs	On input, pointer to the input spectral coefficients to be filtered by the all-pole filters in Q13.18 format. There are 1024 elements in the buffer pointed to by *pSrcDstSpectralCoefs*. On output, pointer to the output spectral coefficients after filtering by the all-pole filters in Q13.18 format. See Table 10-9 for the computation error compared with the double precision data.
pTnsNumFilt	Pointer to the number of noise shaping filters used for each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsNumFilt* which are arranged as follows: *pTnsNumFilt*[*w*]: the number of noise shaping filters used for window *w*, *w* = 0 to *numWin*-1.

pTnsRegionLen	Pointer to the length of the region in units of scale factor bands to which one filter is applied in each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsRegionLen*, which are arranged as follows: *pTnsRegionLen*[*i*]: the length of the region to which filter `filt` is applied in window $w, i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt, w = 0$ to *numWin*-1, *filt*=0 to *pTnsNumFilt*[*w*]-1.
pTnsFiltOrder	Pointer to the order of one noise shaping filter applied to each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsFiltOrder*, which are arranged as follows: *pTnsFiltOrder*[i]: the order of one noise shaping filter `filt`, which is applied to window w, i=$w, i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt, w = 0$ to *numWin*-1, *filt*=0 to *pTnsNumFilt*[w]-1.
pTnsFiltCoefRes	Pointer to the resolution of 3 bits or 4 bits of the transmitted filter coefficients for each window of the current frame. There are 8 elements in the buffer pointed to by *pTnsFiltCoefRes*, which are arranged as follows: *pTnsFiltCoefRes*[w]: the resolution of the transmitted filter coefficients for window *w*, *w* = 0 to *numWin*-1.
pTnsFiltCoef	Pointer to the coefficients of one noise shaping filter applied to each window of the current frame. There are 60 elements in the buffer pointed to by *pTnsFiltCoef*, which are arranged as follows:

pTnsFiltCoef[*i*], *pTnsFiltCoef*[*i*+1], …, *pTnsFiltCoef*[i+order-1]: the coefficients of one noise shaping filter *filt*, which is applied to window w.

The order is the same as that of the noise shaping filter *filt* as applied to window w, w = 0 to numWin-1, filt=0 to *pTnsNumFilt*[w]-1.

For example, *pTnsFiltCoef*[0], *pTnsFiltCoef*[1], …, *pTnsFiltCoef*[order0-1] are the coefficients of the noise shaping filter 0, which is applied to window 0, if present.

If so, *pTnsFiltCoef*[*order0*], *pTnsFiltCoef*[*order0*+1], …, *pTnsFiltCoef*[order0+order1-1] are the coefficients of the noise shaping filter 1 applied to window 0, if present, and so on.

order0 is the same as that of the noise shaping filter 0 applied to window 0, and order1 is the order of the noise shaping filter 1 applied to window 0.

After window 0 is processed, process window 1, then window 2 until all *numWin* windows are processed.

pTnsDirection Pointer to the token that indicates whether the filter is applied in the upward or downward direction.

0 stands for upward and 1 for downward.

There are 8 elements in the buffer pointed to by *pTnsDirection* which are arranged as follows:

pTnsDirection[i]: the token indicating whether the filter filt is applied in upward or downward direction to window

w, i=*w*, $i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt,\ w = 0$

to *numWin*-1, *filt*=0 to *pTnsNumFilt*[*w*]-1.

maxSfb Number of scale factor bands transmitted per window group of the current frame.

profile	Profile index from Table 7.1 in ISO/IEC 13818-7:1997.
samplingRateIndex	Index indicating the sampling rate of the current frame.
winLen	Data number in one window.

Discussion

This function is declared in the ippac.h file. The function performs decoding process for Temporal Noise Shaping (TNS) that controls the temporal shape of the quantization noise within each window of the transform.

The TNS decoding process proceeds separately for each window of the current frame by applying the all-pole filtering to selected regions of the spectral coefficients.

Table 10-9 Computation Error List for *pSrcDstSpectralCoefs*

MAX(error(*pSrcDstSpectralCoefs*[*i*]))	Condition
4095	8 == numWin
32767	1 == numWin

numWin is the number of windows in a window sequence of the current frame. *numWin* is equal to 8 if window sequence is EIGHT_SHORT_SEQUENCE, or to 1 for other window sequences.

numSwb is the total number of scale factor window bands for the actual window type, that is, long or short window of the current frame.

NOTE. *This function supports LC profile only.*

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the pointers *pSrcDstSpectralCoefs*, *pTnsNumFilt*, *pTnsRegionLen*, *pTnsFiltOrder*, *pTnsFiltCoefRes*, *pTnsFiltCoef*, or *pTnsDirection* is NULL.
IppStsTnsProfielErr	Indicates an error when *profile* != 1.
ippStsAacTnsNumFiltErr	Indicates an error when a data error occurs. • for a short window sequence *pTnsNumFilt*[*w*] exceeds [0,1] • for long window sequence, *pTnsNumFilt*[*w*] exceeds [0,3].
ippStsAacTnsLenErr	Indicates an error when **pTnsRegionLen* exceeds [0, *numSwb*].
ippStsAacTnsOrderErr	Indicates an error when a data error occurs. • for short window sequence, **pTnsFiltOrder* exceeds [0,7] • for long window sequence, **pTnsFiltOrder* exceeds [0,12].
ippStsAacTnsCoefResErr	Indicates an error when *pTnsFiltCoefRes*[w] exceeds [3, 4].
ippStsAacTnsCoefErr	Indicates an error when **pTnsFiltCoef* exceeds [-8, 7].
ippStsAacTnsDirectErr	Indicates an error when **pTnsDirection* exceeds [0, 1].

NOTE. *numWin is the number of windows in a window sequence of the current frame. numWin is 8 if window sequence is* EIGHT_SHORT_SEQUENCE*, or 1 for other window sequences.*

NOTE. *numSwb is the total number of scale factor window bands for the actual window type, that is, long or short window, of the current frame.*

MDCTInv_AAC

Maps time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples.

```
IppStatus ippsMDCTInv_AAC_32s16s (Ipp32s *pSrcSpectralCoefs, Ipp16s
    *pDstPcmAudioOut, Ipp32s *pSrcDstOverlapAddBuf, int winSequence, int
    winShape, int prevWinShape, int pcmMode);
```

Arguments

pSrcSpectralCoefs	Pointer to the input time-frequency domain samples in Q13.18 format. There are 1024 elements in the buffer pointed to by *pSrcSpectralCoefs*.
pSrcDstOverlapAddBuf	Pointer to the overlap-add buffer that contains the second half of the previous block windowed sequence in Q13.18. There are 1024 elements in this buffer.
winSequence	Flag that indicates which window sequence is used for current block.

winShape	Flag that indicates which window function is selected for current block.
prevWinShape	Flag that indicates which window function is selected for previous block.
pcmMode	Flag that indicates whether the PCM audio output is interleaved, that is has the pattern LRLRLR… or not. 1 stands for not interleaved. 2 stands for interleaved
pDstPcmAudioOut	Pointer to the output 1024 reconstructed 16-bit signed little-endian PCM samples in Q15, interleaved, if needed. The maximum computation error for *pDstPcmAudioOut* is less than 1 for each vector element. The total quadratic error for the vector is less than 96.
pSrcDstOverlapAddBuf	Pointer to the overlap-add buffer which contains the second half of the current block windowed sequence in Q13.18. The maximum computation error for *pDstPcmAudioOut* is less than 4 for each vector element. The total quadratic error for the vector is less than 1536.

Discussion

This function is declared in the `ippac.h` file. This function maps the time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples as output for each channel.

This module consists of

- IMDCT transform
- windowing
- overlap-add operation.

In order to adapt the time/frequency resolution of the filterbank to the characteristics of the input signal, a block switching tool is also adopted. For each channel, 1024 time-frequency domain samples are transformed into the time domain via the IMDCT.

After applying the windowing operation, the first half of the windowed sequence is added to the second half of the previous block windowed sequence to reconstruct 1024 output samples for each channel. Output can be interleaved according to *pcmMode*.

If *pcmMode* equals to 2, the output is in the sequence *pDstPcmAudioOut*[2*i], i=0 to 1023, that is, 1024 output samples are stored in the sequence: *pDstPcmAudioOut*[0], *pDstPcmAudioOut*[2], *pDstPcmAudioOut*[4],..., *pDstPcmAudioOut*[2046].

If *pcmMode* equals 1, the output is in the sequence *pDstPcmAudioOut*[i], i=0 to 1023.

You should also preallocate an input-output buffer pointed to by *pSrcDstOverlapAddBuf* for the overlap-add operation.

Reset this buffer to zero before the first call and then use the output of the current call as the input of the next call for the same channel.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *pSrcSpectralCoefs*, *pSrcDstOverlapAddBuf* and *pDstPcmAudioOut* is NULL.
ippStsAacWinSeqErr	Indicates an error when *winSequence* exceeds [0,3].
ippStsAacWinShapeErr	Indicates an error when *winShape* or *prevWinShape* exceeds [0,1].
ippStsAacPcmModeErr	Indicates an error when *pcmMode* exceeds [1,2].

MPEG-4 AAC Primitives

This section introduces primitive functions for MPEG-4 AAC operations.

DecodeMainHeader_AAC

Gets main header information and main layer information from bit stream.

```
IppStatus ippsDecodeMainHeader_AAC(Ipp8u **ppBitStream, int *pOffset,
    IppAACMainHeader *pAACMainHeader, int channelNum, int monoStereoFlag);
```

Arguments

ppBitStream	Double pointer to bitstream buffer.
pOffset	Pointer to the offset in one byte.
channelNum	Number of channels.
monoStereoFlag	Current frame has mono and stereo layers.
ppBitStream	Double pointer to bitstream buffer after decode main element.
pOffset	Pointer to the offset in one byte after decode main element.
pAACMainHeader	Pointer to the main element header.

Discussion

This function is declared in the `ippac.h` file. The function gets main header information and main layer information from bit stream.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pAACMainHeader*, `*`*ppBitStream*, or *pOffset* is `NULL`.
`ippStsAacBitOffsetErr`	Indicates an error when *pOffset* exceeds `[0,7]`.
`ippStsAacChanErr`	Indicates an error when *channelNum* exceeds `[1,2]`.
`ippStsAacMonoStereoErr`	Indicates an error when *monoStereoFlag* exceeds `[0,1]`.

DecodeExtensionHeader_AAC

Gets extension header information and extension layer information from bit stream.

```
IppStatus ippsDecodeExtensionHeader_AAC(Ipp8u **ppBitStream, int *pOffset,
    IppAACExtHeader *pAACExtHeader, int monoStereoFlag, int thisLayerStereo,
    int monoLayerFlag, int preStereoMaxSfb, int hightstMonoMaxSfb, int
    winSequence);
```

Arguments

ppBitStream	Double pointer to the bitstream buffer.
pOffset	Pointer to the offset in one byte.
monoStereoFlag	Flag indicating that the current frame has mono and stereo layers.
thisLayerStereo	Flag indicating that the current layer is stereo.
monoLayerFlag	Flag indicating that the current frame has a mono layer.
preStereoMaxSfb	Previous stereo layer *maxSfb*.
hightstMonoMaxSfb	Last mono layer *maxSfb*.
winSequence	Window type, short or long.
pAACExtHeader	Pointer to the extension element header.

Discussion

This function is declared in the `ippac.h` file. The function gets extension header information and extension layer information from the bitstream.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsBadArgErr`	Indicates an error when at least one of the pointers *ppBitStream*, *pAACExtHeader*, or *pOffset* is `NULL`.

ippStsAacBitOffsetErr	Indicates an error when *pOffset is out of the range of [0,7].
ippStsAacStereoLayerErr	Indicates an error when *thisLayerStereo* exceeds [0,1].
ippStsAacMonoLayerErr	Indicates an error when *monoLayerFlag* exceeds [0,1].
ippStsAacMaxSfbErr	Indicates an error when *maxSfb* exceeds [0,IPP_AAC_MAX_SFB].
ippStsAacMonoStereoErr	Indicates an error when *monoStereoFlag* exceeds [0,1].
ippStsAacWinSeqErr	Indicates an error when *winSequence* exceeds [0,3].

DecodePNS_AAC

Implements perceptual noise substitution (PNS) coding within individual channel stream (ICS).

```
IppStatus ippsDecodePNS_AAC_32s(Ipp32s *pSrcDstSpec, int *pSrcDstLtpFlag,
    Ipp8u *pSfbCb, Ipp16s *pScaleFactor, int maxSfb,int numWinGrp, int
    *pWinGrpLen,int samplingFreqIndex, int winLen,int *pRandomSeed);
```

Arguments

pSrcDstSpec	Pointer to spectrum coefficients for perceptual noise substitution (PNS).
pSrcDstLtpFlag	Pointer to long term predict (LTP) flag.
pSfbCb	Pointer to the scale factor codebook.
pScaleFactor	Pointer to the scale factor value.
maxSfb	Number of scale factor bands used in this layer.
numWinGrp	Number of window groups.
pWinGrpLen	Pointer to the length of every window group.
samplingFreqIndex	Sampling frequency index.
winLen	Window length. 1024 for long windows, 128 for short windows.

pRandomSeed	Random seed for PNS.
pSrcDstSpec	Pointer to the output spectrum substituted by perceptual noise.

Discussion

This function is declared in the ippac.h file. The function implements perceptual noise substitution (PNS) coding within the individual channel stream (ICS). Certain sets of spectral coefficients are derived from random vectors rather than from Huffman-coded symbols and inverse quantization process.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *pSrcDstSpec*, *pSfbCb*, *pScaleFactor*, *pRandomSeed*, *pWinGrpLen* or *pSrcDstLtpFlag* is NULL.
ippStsAacMaxSfbErr	Indicates an error when *maxSfb* exceeds [0,IPP_AAC_MAX_SFB].
ippStsAacSmplRateIdxErr	Indicates an error when *pChanInfo->samplingRateIndex* exceeds [0,12].
ippStsAacWinLenErr	Indicates an error when *winLen* is not equal to 128 or 1024.

LongTermReconstruct_AAC

Uses Long Term Reconstruct (LTR) to reduce signal redundancy between successive coding frames.

```
IppStatus ippsLongTermReconstruct_AAC_32s(Ipp32s *pSrcEstSpec, Ipp32s
    *pSrcDstSpec, int *pLtpFlag, int winSequence, int samplingFreqIndex);
```

Arguments

pSrcDstSpec	Pointer to spectral coefficients for LTP.

pSrcEstSpec	Pointer to the frequency domain vector.
winSequence	Window type, long or short.
samplingFreqIndex	Sampling frequency index.
pLtpFlag	Pointer to the LTP flag.

Discussion

This function is declared in the ippac.h file. The function uses Long Term Reconstruct (LTP) to reduce signal redundancy between successive coding frames.

LTP is a forward adaptive predictor that is inherently less sensitive to the round-off numerical errors in the decoder or bi-errors in the transmitted spectral coefficients.

You should add the vector of decoded spectral coefficients and the corresponding frequency domain vector to get the vector of reconstructed spectral coefficients.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the pointers *pSrcDstSpec*, *pSrcEstSpec* and *pLtpFlag* is NULL.
ippStsAacSmplRateIdxErr	Indicates an error when *pChanInfo->samplingRateIndex* exceeds [0,12].
ippStsAacWinSeqErr	Indicates an error when *winSequence* exceeds [0,3].

MDCTFwd_AAC

Generates spectrum coefficient of PCM samples.

```
IppStatus ippsMDCTFwd_AAC_32s(Ipp32s *pSrc, Ipp32s *pDst, Ipp32s
    *pSrcDstOverlapAdd, int winSequence, int winShape, int preWinShape, Ipp32s
    *pWindowedBuf);
```

Arguments

pSrc	Pointer to temporal signals to do MDCT.

pSrcDstOverlapAdd	Pointer to overlap buffer. Not used for MPEG-4 AAC decoding.
winSequence	Window sequence indicating if the block is long or short.
winShape	Current window shape.
preWinShape	Previous window shape.
pWindowedBuf	Work buffer for MDCT. Should be at least 2048 words.
pSrcDstOverlapAdd	Pointer to overlap buffer. Not used for MPEG-4 AAC decoding.
pDst	Output of MDCT, the spectral coefficients of PCM samples.

Discussion

This function is declared in the `ippac.h` file. The function generates the spectrum coefficient of PCM samples in the MDCT Long Term Reconstruct (LTP) loop.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *pSrc*, *pDst*, *pWindowedBuf* or *pSrcDstOverlapAdd* is `NULL`.
`ippStsAacWinSeqErr`	Indicates an error when *winSequence* exceeds `[0,3]`.
`ippStsAacWinShapeErr`	Indicates an error when *preWinShape* exceeds `[0,1]`.

EncodeTNS_AAC

Performs reversion of TNS in the Long Term Reconstruct loop in-place.

```
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);
```

Arguments

pSrcDst	On input, pointer to the spectral coefficients for the TNS encoding operation. On output, pointer to the spectral coefficients after the TNS encoding operation.
pTnsNumFilt	Pointer to the number of TNS filters.
pTnsRegionLen	Pointer to the length of TNS filter.
pTnsFiltOrder	Pointer to the TNS filter order.
pTnsFiltCoefRes	Pointer to the TNS coefficient resolution flag.
pTnsFiltCoef	Pointer to the TNS filter coefficients.
pTnsDirection	Pointer to the TNS direction flag.
maxSfb	Maximum scale factor number.
profile	Audio profile.
samplingFreqIndex	Sampling frequency index.
winLen	Window length.

Discussion

This function is declared in the `ippac.h` file. The function performs in-place reversion of TNS in the LTP loop, or Analysis Temporal Noise Shaping.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *pSrcDstSpectralCoef*, *pTnsNumFilt*, *pTnsRegionLen*, *pTnsFiltOrder*, *pTnsFiltCoefRes*, `pTnsFiltCoef`, or *pTnsDirection* is `NULL`.
`ippStsTnsProfileErr`	Indicates an error when *profile* `!= 1`.
`ippStsAacTnsNumFiltErr`	Indicates an error when `*`*pTnsNumFilt* exceeds `[0, 1]` for the short window sequence or `[0, 3]` for the long window sequence.

ippStsAacTnsLenErr	Indicates an error when *pTnsRegionLen exceeds [0,numSwb].
ippStsAacTnsOrderErr	Indicates an error when *pTnsFiltOrder exceeds [0, 7] for the short window sequence or [0, 12] for the long window sequence.
ippStsAacTnsCoefResErr	Indicates an error when *pTnsFiltCoefRes exceeds [3, 4].
ippStsAacTnsCoefErr	Indicates an error when *pTnsFiltCoef exceeds [-8, 7].
ippStsAacTnsDirectErr	Indicates an error when *pTnsDirection exceeds [0,1].
ippStsAacSmplRateIdxErr	Indicates an error when samplingRateIndex exceeds [0, 12].
ippStsAacWinLenErr	Indicates an error when winLen is not equal to 128 or 1024.

LongTermPredict_AAC

Gets the predicted time domain signals in the Long Term Reconstruct (LTP) loop.

```
IppStatus ippsLongTermPredict_AAC_32s(Ipp32s *pSrcTimeSignal, Ipp32s
    *pDstEstTimeSignal, IppAACLtpInfo *pAACLtpInfo, int winSequence);
```

Arguments

pSrcTimeSignal	Pointer to the temporal signals to be predicted in the temporary domain.
pDstEstTimeSignal	Pointer to the output of samples after LTP.
pAACLtpInfo	Pointer to the LTP information.
winSequence	Window type, short or long.
pDstEstTimeSignal	Pointer to the prediction output in time domain.

Discussion

This function is declared in the `ippac.h` file. The function gets the predicted time domain signals in the LTP loop.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers is `NULL`.
`ippStsAacWinSeqErr`	Indicates an error when *`winSequence`* exceeds `[0,3]`.

NoiseLessDecode_AAC

Performs noiseless decoding.

```
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);
```

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 for scalable objects. When *`commonWin`* `== 0 &&` *`scaleFlag`*`==0`, you need to decode LTP information and save it in *`pAACMainHeader`*`->pLtpInfo[]`.
`pChanInfo`	Pointer to channel information structure.
`windowSequence`	Window type, short or long.
`maxSfb`	Number of scale factor bands.

commonWin	Indicates if the channel pair uses the same ICS information.
scaleFlag	Flag indicating whether the scalable type is used.
audioObjectType	Audio object type indicator: • 1 indicates the main type • 2 indicates the LC type • 6 indicates the scalable mode.
ppBitStream	Double pointer to the parsed bitstream.
pOffset	Pointer to the offset in one byte.
pChanInfo	Pointer to the channel information structure.
pDstScaleFactor	Pointer to the parsed scale factor.
pDstQuantizedSpectralCoef	Pointer to the quantized spectral coefficients after Huffman decoding.
pDstSfbCb	Pointer to the scale factor codebook index.
pDstTnsFiltCoef	Pointer to TNS filter coefficients. Not used for scalable objects.

Discussion

This function is declared in the `ippac.h` file. This is a general noiseless decoding module for MPEG-2 and MPEG-4 objects.

In case one scale factor band uses PNS in MPEG-4 AAC scalable object, *`pDstScaleFactor` contains the noise energy for this scale factor band and *`pDstSfbCb`*`[sfb]` should be `NOISE_HCB(13)`. The spectrum in this scale factor band is not necessarily Huffman-decoded, and the *`pDstQuantizedSpectralCoef`* of this scale factor band can be zero.

In AAC scalable object, *`pDstTnsFiltCoef`* and *`pAACMainHeader`* are not used.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, **ppBitStream*, *pAACMainHeader*, *pDstScaleFactor*, *pDstTnsFiltCoef*, *pDstQuantizedSpectralCoef*, *pChanInfo* or *pDstSfbCb* is NULL.
ippStsAacBitOffsetErr	Indicates an error when **pOffset* exceeds [0,7].
ippStsAacComWinErr	Indicates an error when *commonWin* exceeds [0,1].
ippStsAacMaxSfbErr	Indicates an error when *maxSfb* exceeds [0,IPP_AAC_MAX_SFB].
ippStsAacSmplRateIdxErr	Indicates an error when *pChanInfo->samplingRateIndex* exceeds [0,11].
ippStsAacCoefValErr	Indicates an error when the quantized coefficients value pointed to by *pDstCoef* exceeds the range of [-8191,8191].

LtpUpdate_AAC

Performs required buffer update in the Long Term Reconstruct (LTP) loop.

```
IppStatus ippsLtpUpdate_AAC_32s (Ipp32s *pSpecVal, Ipp32s *pLtpSaveBuf, int
    winSequence, int winShape, int preWinShape, Ipp32s *pWorkBuf);
```

Arguments

pSpecVal	Pointer to spectral value after TNS decoder in LTP loop.
pLtpSaveBuf	Pointer to the save buffer for LTP. Buffer length should be 3**frameLength*.
winSequence	Window type.

- 0 stands for long
- 1 stands for long start
- 2 stands for short
- 3 stands for long stop.

winShape	KBD or SIN window shape.
preWinShape	Previous window shape.
pWorkBuf	Work buffer for LTP update, length of *pWorkBuf* should be at least 2048*3 = 6144 words.
pLtpSaveBuf	Pointer to save buffer for LTP. Buffer length should be 3**frameLength*. The value is saved for next frame.

Discussion

This function is declared in the `ippac.h` file. The function performs required buffer update in the Long Term Reconstruct (LTP) loop. This operation includes IMDCT and updating the save buffer.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the pointers *pLtpSaveBuf*, *pWorkBuf*, or *pSpecVal* is `NULL`.
`ippStsAacWinSeqErr`	Indicates an error when *winSequence* exceeds `[0,3]`.
`ippStsAacWinShapeErr`	Indicates an error when *winShape* or *preWinShape* exceeds `[0,1]`.

String Functions 11

This chapter describes Intel® IPP functions that perform operations with strings. Intel IPP string functions do not consider zero as the end of the string, but require that the length of the string (number of elements) be specified explicitly.
Overlapping of the strings is not supported (for not in-place operations).
Intel IPP string functions operate with two data types, `Ipp8u` and `Ipp16u`.

The full list of functions in this group is given in Table 11-1.

Table 11-1 Intel IPP String Functions

Function Base Name	Operation
`Find`, `FindRev`	Looks for the first occurrence of the substring matching the specified string.
`FindC` `FindRevC`	Looks for the first occurrence of the specified element within the source string.
`FindCAny` `FindRevCAny`	Looks for the first occurrence of any element of the specified array within the source string.
`Insert`	Inserts a string into another string.
`Remove`	Removes a specified number of elements from the string.
`Compare`	Compares two strings of the fixed length.
`CompareIgnore`, `CompareIgnoreLatin`	Compares two strings of the fixed length ignoring case.
`Equal`	Compares two strings of the fixed length for equality.
`TrimC`	Deletes all occurrences of a specified symbol both in the beginning and in the end of the string.
`TrimCAny`	Deletes all occurrences of any of the specified symbols both in the beginning and in the end of the source string.
`TrimStartCAny` `TrimEndCAny`	Deletes all occurrences of any of the specified symbols either in the beginning or in the end of the source string, respectively.
`ReplaceC`	Replaces all occurrences of a specified element in the source string with another element.

Table 11-1 Intel IPP String Functions

Function Base Name	Operation
Uppercase, UppercaseLatin	Converts alphabetic characters of a string to all uppercase symbols.
Lowercase, LowercaseLatin	Converts alphabetic characters of a string to all lowercase symbols.
Hash	Calculates a hash value for the string.
Concat	Concatenates several strings together.
ConcatC	Concatenates several strings together and inserts symbol delimiters between them.
SplitC	Splits source string into separate parts.

Find, FindRev

Looks for the first occurrence of the substring matching the specified string.

```
IppStatus ippsFind_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind,
      int lenFind, int* pIndex);
IppStatus ippsFind_16u(const Ipp16u* pSrc, int len, const Ipp16u* pFind,
      int lenFind, int* pIndex);
IppStatus ippsFindRev_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind,
      int lenFind, int* pIndex);
IppStatus ippsFindRev_16u(const Ipp16u* pSrc, int len, const Ipp16u* pFind,
      int lenFind, int* pIndex);
```

Arguments

pSrc	Pointer to the source string.
len	Number of elements in the source string.
pFind	Pointer to the reference string.
lenFind	Number of elements in the reference string.

pIndex	Pointer to the result index.

Discussion

The functions ippsFind and ippsFindRev are declared in the ippch.h file. These functions search through the source string *pSrc* for a substring of elements that match the specified reference string *pFind*. Starting point of the first occurrence of the matching substring is stored in *pIndex*. If no matching substring is found, then *pIndex* is set to -1. The function ippsFindRev searches the source string in the reverse direction.
The search is case-sensitive.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* or *lenFind* is negative.

FindC
FindRevC

Looks for the first occurrence of the specified element within the source string.

```
IppStatus ippsFindC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind,
      int* pIndex);
IppStatus ippsFindC_16u(const Ipp16u* pSrc, int len, Ipp16u valFind,
      int* pIndex);
IppStatus ippsFindRevC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind,
      int* pIndex);
IppStatus ippsFindRevC_16u(const Ipp16u* pSrc, int len, Ipp16u valFind,
      int* pIndex);
```

Arguments

pSrc	Pointer to the source string.
len	Number of elements in the source string.
valFind	Value of the specified element.
pIndex	Pointer to the result index.

Discussion

The functions ippsFindC and ippsFindRevC are declared in the ippch.h file. These functions search through the source string *pSrc* for the first occurrence of the specified element with the value *valFind*. The position of this element is stored in *pIndex*. If no matching element is found, then *pIndex* is set to -1. The function ippsFindRev searches the source string in the reverse direction.
The search is case-sensitive.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

FindCAny, FindRevCAny

Looks for the first occurrence of any element of the specified array within the source string.

```
IppStatus ippsFindCAny_8u(const Ipp8u* pSrc, int len,
     const Ipp8u* pAnyOf, int lenAnyOf, int* pIndex);
IppStatus ippsFindCAny_16u(const Ipp16u* pSrc, int len,
     const Ipp16u* pAnyOf, int lenAnyOf, int* pIndex);
IppStatus ippsFindRevCAny_8u(const Ipp8u* pSrc, int len,
     const Ipp8u* pAnyOf, int lenAnyOf, int* pIndex);
```

```
IppStatus ippsFindRevCAny_16u(const Ipp16u* pSrc, int len,
     const Ipp16u* pAnyOf, int lenAnyOf, int* pIndex);
```

Arguments

pSrc	Pointer to the source string.
len	Number of elements in the source string.
pAnyOf	Pointer to the array containing reference elements.
lenAnyOf	Number of elements in the array.
pIndex	Pointer to the result index.

Discussion

The functions `ippsFindCAny` and `ippsFindRevCAny` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for the first occurrence of any reference element from the specified array *pAnyOf*. The position of this element is stored in *pIndex*. If no matching element is found, then *pIndex* is set to `-1`. The function `ippsFindRevCAny` searches the source string in the reverse direction. The search is case-sensitive.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len* or *lenAnyOf* is negative.

Insert

Inserts a string into another string.

```
IppStatus ippsInsert_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u*
     pInsert, int insertLen, Ipp8u* pDst, int startIndex);
```

```
IppStatus ippsInsert_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
     pInsert, int insertLen, Ipp16u* pDst, int startIndex);
IppStatus ippsInsert_8u_I(const Ipp8u* pInsert, int insertLen,
     Ipp8u* pSrcDst, int* pSrcDstLen, int startIndex);
IppStatus ippsInsert_16u_I(const Ipp16u* pInsert, int insertLen,
     Ipp16u* pSrcDst, int* pSrcDstLen, int startIndex);
```

Arguments

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pInsert	Pointer to the string to be inserted.
insertLen	Number of elements in the string to be inserted.
pDst	Pointer to the destination string.
pSrcDst	Pointer to the source and destination string for in-place operation.
pSrcDstLen	Pointer to the number of elements in the source and destination string for in-place operation.
startIndex	Index of the insertion point.

Discussion

The function ippsInsert is declared in the ippch.h file. This function inserts the string *pInsert* containing *insertLen* elements into a source string *pSrc* of length *srcLen*. Insertion position is specified by *startIndex*. The result is stored in the *pDst*.

The in-place flavors of ippsInsert insert the string *pInsert* in the source string *pSrcDst* and store the result in the destination string *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if one of the *srcLen*, *insertLen*, *pSrcSdtLen*, *startIndex* is negative, or *startIndex* is greater than *srcLen* or *pSrcDstLen*.

Remove

Removes a specified number of elements from the string.

```
IppStatus ippsRemove_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst,
     int startIndex, int len);
IppStatus ippsRemove_16u(const Ipp16u* pSrc, int srcLen, Ipp16u* pDst,
     int startIndex, int len);
IppStatus ippsRemove_8u_I(Ipp8u* pSrcDst, int* pSrcDstLen, int
     startIndex, int len);
IppStatus ippsRemove_16u_I(Ipp16u* pSrcDst, int* pSrcDstLen, int
     startIndex, int len);
```

Arguments

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pDst	Pointer to the destination string.
pSrcDst	Pointer to the source and destination string for in-place operation.
pSrcDstLen	Pointer to the number of elements in the source and destination string for in-place operation.
startIndex	Index of the starting point.
len	Number of elements to be removed.

Discussion

The function ippsRemove is declared in the ippch.h file. This function removes the *len* elements from a source string *pSrc* of length *srcLen*. Starting position is specified by *startIndex*. The result is stored in the *pDst*.

The in-place flavors of ippsRemove remove the *len* elements from the source string *pSrcDst* and store the result in the destination string *pSrcDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if one of the *srcLen*, *len*, *pSrcSdtLen*, *startIndex* is negative, or *(startIndex+len)* is greater than *srcLen* or *pSrcDstLen*.

Compare

Compares two strings of the fixed length.

```
IppStatus ippsCompare_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
     int len, int *pResult);
IppStatus ippsCompare_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
     int len, int *pResult);
```

Arguments

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Discussion

The function ippsCompare is declared in the ippch.h file. This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. The value *pResult* = *pSrc1*[i]-*pSrc2*[i] is computed successively for each i-th element, *i* =0,...*len*-1. When the first pair of non-matching elements occurs (that is, when *pResult* is not equal to zero), the function stops operation and returns the value *pResult*. The returned value is positive when *pSrc1*[i]>*pSrc2*[i] and negative when *pSrc1*[i]<*pSrc2*[i]. If the strings are equal, the function returns *pResult* = 0. The comparison is case-sensitive.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

CompareIgnoreCase, CompareIgnoreCaseLatin

Compares two strings of the fixed length ignoring case.

```
IppStatus ippsCompareIgnoreCase_16u(const Ipp16u* pSrc1,
       const Ipp16u* pSrc2, int len, int* pResult);
IppStatus ippsCompareIgnoreCaseLatin_8u(const Ipp8u* pSrc1,
       const Ipp8u* pSrc2, int len, int* pResult);
IppStatus ippsCompareIgnoreCaseLatin_16u(const Ipp16u* pSrc1,
       const Ipp16u* pSrc2, int len, int* pResult);
```

Arguments

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Discussion

The functions ippsCompareIgnoreCase and ippsCompareIgnoreCaseLatin are declared in the ippch.h file. These functions compare first *len* elements of two strings *pSrc1* and *pSrc2*. If all pairs of elements in the strings are equal, the function returns *pResult* = 0. If the pair of non-matching elements occurs in the i-th position, the function stops operation and returns *pResult*. The returned value is positive when *pSrc1*[i]>*pSrc2*[i] and negative when *pSrc1*[i]<*pSrc2*[i].
The comparison is case-insensitive.

The function ippsCompareIgnore operates with Unicode characters.
The function ippsCompareIgnoreLatin operates with ASCII characters.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Equal

Compares two string of the fixed length for equality.

```
IppStatus ippsEqual_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len,
      int* pResult);
IppStatus ippsEqual_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, int len,
      int* pResult);
```

Arguments

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Discussion

The function ippsEqual is declared in the ippch.h file. This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. Each element of the first string is compared with the corresponding element of the second string. When the first pair of non-matching elements is found, the function stops operation and stores 0 in *pResult*. If the strings are equal, the function stores 1 in *pResult*.
The comparison is case-sensitive.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

TrimC

Deletes all occurrences of a specified symbol in the beginning and in the end of the string.

```
IppStatus ippsTrimC_8u(const Ipp8u* pSrc, int srcLen, Ipp8u odd,
     Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimC_16u(const Ipp16u* pSrc, int srcLen, Ipp16u odd,
     Ipp16u* pDst, int* pDstLen);
IppStatus ippsTrimC_8u_I(Ipp8u* pSrcDst, int* pLen, Ipp8u odd);
IppStatus ippsTrimC_16u_I(Ipp16u* pSrcDst, int* pLen, Ipp16u odd);
```

Arguments

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pSrcDst	Pointer to the source and destination string for the in-place operation.
pDst	Pointer to the destination string.
pDstLen	Pointer to the computed number of elements in the destination string.
pLen	Pointer to the number of elements in the source and destination string for the in-place operation.
odd	Symbol to be deleted.

Discussion

The function ippsTrimC is declared in the ippch.h file. This function deletes all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrc* containing *srcLen* elements. The function stores the result string containing *pDstLen* elements in *pDst*.

The in-place flavors of ippsTrimC delete all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrcDst* containing *pLen* elements. These functions store the result string containing *pLen* elements in *pSrcDst*.

The operation is case-sensitive.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *srcLen* or *pLen* is negative.

TrimCAny
TrimStartCAny
TrimEndCAny

Deletes all occurrences of any of the specified symbols in the beginning and in the end of the source string.

```
IppStatus ippsTrimCAny_8u(const Ipp8u* pSrc, int srcLen,
     const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimCAny_16u(const Ipp16u* pSrc, int srcLen,
     const Ipp16u* pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);

IppStatus ippsTrimStartCAny_8u(const Ipp8u* pSrc, int srcLen,
     const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
```

```
IppStatus ippsTrimStartCAny_16u(const Ipp16u* pSrc, int srcLen,
     const Ipp16u* pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);

IppStatus ippsTrimEndCAny_8u(const Ipp8u* pSrc, int srcLen,
     const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimEndCAny_16u(const Ipp16u* pSrc, int srcLen,
     const Ipp16u* pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);
```

Arguments

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pTrim	Pointer to the array containing the specified elements.
trimLen	Number of elements in the array.
pDst	Pointer to the destination string.
pDstLen	Pointer to the computed number of elements in the destination string.

Discussion

The functions ippsTrimCAny, ippsTrimStartCAny, and ippsTrimEndCAny are declared in the ippch.h file.
The function ippsTrimCAny deletes all occurrences of any of the specified elements stored in the array *pTrim* if they are present either in the beginning or in the end of the source string *pSrc*, and stores the result string containing *pDstLen* elements in *pDst*.
The function stops operation when it finds the first non-matching element.
The functions ippsTrimStartCAny and ippsTrimEndCAny perform this operation only in the beginning or in the end of the source string *pSrc,* respectively.
The operation is case-sensitive.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *srcLen* or *trimLen* is negative.

ReplaceC

Replaces all occurrences of a specified element in the source string with another element.

```
IppStatus ippsReplaceC_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
      Ipp8u oldVal, Ipp8u newVal);
IppStatus ippsReplaceC_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len,
      Ipp16u oldVal, Ipp16u newVal);
```

Arguments

pSrc	Pointer to the source string.
len	Number of elements in the source string.
pDst	Pointer to the destination string.
oldVal	Element to be replaced.
newVal	Element that replaces *oldVal*.

Discussion

The function ippsReplaceC is declared in the ippch.h file. This function replaces all occurrences of a specified element *oldVal* in the source string *pSrc* with another specified element *newVal*, and stores the new string in the *pDst*.
The operation is case-sensitive.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Uppercase, UppercaseLatin

Converts alphabetic characters of a string to all uppercase symbols.

```
IppStatus ippsUppercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsUppercase_16u_I(Ipp16u* pSrcDst, int len);

IppStatus ippsUppercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsUppercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsUppercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsUppercaseLatin_16u_I(Ipp16u* pSrcDst, int len);
```

Arguments

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

Discussion

The functions `ippsUppercase` and `ippsUppercaseLatin` are declared in the `ippch.h` file. These functions convert each alphabetic character of the source string *pSrc* to upper case and stores the result in *pDst*.

The in-place flavors of these functions convert each alphabetic character of the source string *pSrcDst* to upper case and store the result in *pSrcDst*.

The function `ippsUppercase` operates with Unicode characters.
The function `ippsUppercaseLatin` operates with ASCII characters.

Return Value

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Lowercase, LowercaseLatin

Converts alphabetic characters of a string to all lowercase symbols.

```
IppStatus ippsLowercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsLowercase_16u_I(Ipp16u* pSrcDst, int len);

IppStatus ippsLowercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLowercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsLowercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsLowercaseLatin_16u_I(Ipp16u* pSrcDst, int len);
```

Arguments

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

Discussion

The functions ippsLowercase and ippsLowercaseLatin are declared in the ippch.h file. These functions convert each alphabetic character of the source string *pSrc* to lower case and store the result in *pDst*.

The in-place flavors of these functions convert each alphabetic character of the source string *pSrcDst* to lower case and store the result in *pSrcDst*.

The function ippsLowercase operates with Unicode characters.
The function ippsLowercaseLatin operates with ASCII characters.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Hash

Calculates the hash value for the string.

```
IppStatus ippsHash_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHash_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);
```

Arguments

pSrc	Pointer to the source string.
len	Number of elements in the string.
pHashVal	Pointer to the result value.

Discussion

The function ippsHash is declared in the ippch.h file. This function produces the hash value *pHasVal* for the specified string *pSrc*. The hash value is fairly unique to the given string. If hash values of two strings are different, the strings are different as well. If the hush values are the same, the strings are probably identical. The hash value is calculated in the following manner: initial value is set to 0, then the hash value is calculated successively for each i-th string element as *pHashVal*[i] = 2**pHashVal*[i-1]^*pSrc*[i], where ^ denotes a bitwise exclusive OR (XOR) operator. The hash value for the last element is the hash value for the whole string.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Concat

Concatenates several strings together.

```
IppStatus ippsConcat_8u_D2L(const Ipp8u* const pSrc[],
     const int* pSrcLen[], int numSrc, Ipp8u* pDst);
IppStatus ippsConcat_16u_D2L(const Ipp16u* const pSrc[],
     const int* pSrcLen[], int numSrc, Ipp16u* pDst);
IppStatus ippsConcat_8u(const Ipp8u* pSrc1, int len1, const Ipp8u* pSrc2,
     int len2, Ipp8u* pDst);
IppStatus ippsConcat_16u(const Ipp16u* pSrc1, int len1, const Ipp16u* pSrc2,
     int len2, Ipp16u* pDst);
```

Arguments

pSrc1	Pointer to the first source string.
len1	Number of elements in the first string.
pSrc2	Pointer to the second source string.
len2	Number of elements in the second string.
pSrc	Pointer to the array of source strings.
pSrcLen	Pointer to the array of lengths of the source strings.
numSrc	Number of source strings.
pDst	Pointer to the destination string.

Discussion

The function ippsConcat is declared in the ippch.h file. This function concatenates several strings together. Functions with D2L suffix operate with multiple *numSrc* strings *pSrc*[], while functions without this suffix operate with two strings *pSrc1* and *pSrc2* only. Resulting string is stored in the *pDst*. Necessary memory blocks should be allocated for the destination string before the function is called. Summary length of the strings to be concatenated can not be greater than IPP_MAX_32S.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len1* or *len2* is negative, or *srcLen*[i] is negative for i < *numSrc*.
ippStsSizeErr	Indicates an error condition if *numSrc* is equal to or less than 0.

ConcatC

Concatenates several strings together and inserts symbol delimiters between them.

```
IppStatus ippsConcatC_8u_D2L(const Ipp8u* const pSrc[],
     const int* pSrcLen[], int numSrc, Ipp8u delim, Ipp8u* pDst);
IppStatus ippsConcatC_16u_D2L(const Ipp16u* const pSrc[],
     const int* pSrcLen[], int numSrc, Ipp16u delim, Ipp16u* pDst);
```

Arguments

pSrc	Pointer to the array of source strings.
pSrcLen	Pointer to the array of lengths of the source strings.
numSrc	Number of source strings.
delim	Symbol delimiter.

pDst	Pointer to the destination string.

Discussion

The function ippsConcatC is declared in the ippch.h file. This function concatenates *numSrc* strings *pSrc* together and inserts a specified delimiter symbol *delim* between them in the resulting string *pDst*.

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *srcLen*[i] is negative for i < *numSrc*.
ippStsSizeErr	Indicates an error condition if *numSrc* is equal to or less than 0.

SplitC

Splits source string into separate parts.

```
IppStatus ippsSplitC_8u_D2L(const Ipp8u* pSrc, int srcLen, Ipp8u delim,
      Ipp8u* pDst[], int* pDstLen[], int* pNumDst);
IppStatus ippsSplitC_16u_D2L(const Ipp16u* pSrc, int srcLen, Ipp16u delim,
      Ipp16u* pDst[], int* pDstLen[], int* pNumDst);
```

Arguments

pSrc	Pointer to the source strings.
srcLen	Number of elements in the source string.
delim	Symbol delimiter.
pDst	Pointer to the array of the destination strings.
pDstLen	Pointer to array of the destination string lengths.
pNumDst	Number of destination strings.

Discussion

The function `ippsSplitC` is declared in the `ippch.h` file. This function breaks source string *`pSrc`* into *`pNumDst`* separate strings *`pDst`* using a specified symbol *`delim`* as a delimiter.
If `n` specified delimiters occur in the beginning or in the end of the source string, then `n` empty strings are appended to the array of destination strings.
If `n` specified delimiters occur in a certain position within the source string, then (`n-1`) empty strings will be inserted into the array of destination strings, where `n` is the number of delimiter occurrences in this position.

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *`srcLen`* is negative, or *`dstLen`* is negative for `i < pNumDst`.
`ippStsSizeErr`	Indicates an error condition if *`pNumDst`* is equal to or less than 0.
`ippStsOvermatchStrings`	Indicates a warning if number of output strings exceeds the initially specified number *`pNumDst`*; in this case odd strings are discarded.
`ippStsOverlongString`	Indicates a warning if in some output strings the number of elements exceeds the initially specified value *`pDstLen`*; in this case corresponding strings are truncated to initial lengths.

Fixed-Accuracy Arithmetic Functions

12

This chapter describes Intel® IPP fixed-accuracy transcendental mathematical functions of vector arguments. These functions take an input vector as argument, compute values of the respective elementary function element-wise, and return the results in an output vector.

Function specifications comply with the common API agreement of Intel IPP, but include some new features essential to scientific arithmetic functions. The main feature is a more elaborate specification of accuracy that differs from the common definition in adding several new levels of accuracy, besides original levels introduced by single precision and double precision data formats.

Fixed-accuracy vector functions implementation supports IEEE-754 standard in all flavors, which means that:

- All functions have a precisely determined and guaranteed level of accuracy for all argument values.
- All special value processing and exceptions handling requirements are met, which implies that when accuracy is below the standard level, the function meets IEEE-754 requirements in all other respects.

The choice of accuracy levels should be based on practical experience and identified application demands. Available options are specified in the function name suffix and include `A11`, `A21`, or `A24` for the single precision, and `A50` or `A53` for the double precision data format. Flavors `A11`, `A21`, and `A50` provide approximately 3, 6, and 15 exact decimal digits, respectively. For flavors `A24` and `A53`, the maximum guaranteed error is within 1 ulp and in most cases does not exceed 0.55 ulp.

Fixed-accuracy arithmetic functions subset of Intel IPP has the similar functionality as the respective part of Intel® Math Kernel Library (Intel®MKL) .

However, Intel IPP provides lower-level transcendental functions that have separate flavors for each mode of operations and data type and are better suitable for multimedia and signal processing in real time applications.
The full list of these functions is given in Table 12-1.

Table 12-1 Intel IPP Fixed-Accuracy Arithmetic Functions

Function Short Name	Data Types	Description
Power and Root Functions		
Inv	32f, 64f	Computes inverse value of each vector element.
Div	32f, 64f	Divides elements of one vector by corresponding elements of another vector.
Sqrt	32f, 64f	Computes square root of each vector element.
InvSqrt	32f, 64f	Computes inverse square root of each vector element.
Cbrt	32f, 64f	Computes cube root of each vector element.
InvCbrt	32f, 64f	Computes inverse cube root of each vector element.
Pow	32f, 64f	Raises each element of one vector to the power of corresponding element of another vector.
Powx	32f, 64f	Raises each element of a vector to a constant power.
Exponential and Logarithmic Functions		
Exp	32f, 64f	Raises *e* to the power of each vector element.
Ln	32f, 64f	Computes natural logarithm of each vector element.
Log10	32f, 64f	Computes common logarithm of each vector element.
Trigonometric Functions		
Cos	32f, 64f	Computes cosine of each vector element.
Sin	32f, 64f	Computes sine of each vector element.
SinCos	32f, 64f	Computes sine and cosine of each vector element.
Tan	32f, 64f	Computes tangent of each vector element.
Acos	32f, 64f	Computes inverse cosine of each vector element.
Asin	32f, 64f	Computes inverse sine of each vector element.
Atan	32f, 64f	Computes inverse tangent of each vector element.
Atan2	32f, 64f	Computes four-quadrant inverse tangent of elements of two vectors.
Hyperbolic Functions		
Cosh	32f, 64f	Computes hyperbolic cosine of each vector element.

Table 12-1 Intel IPP Fixed-Accuracy Arithmetic Functions (continued)

Function Short Name	Data Types	Description
Sinh	32f, 64f	Computes hyperbolic sine of each vector element.
Tanh	32f, 64f	Computes hyperbolic tangent of each vector element.
Acosh	32f, 64f	Computes inverse (nonnegative) hyperbolic cosine of each vector element.
Asinh	32f, 64f	Computes inverse hyperbolic sine of each vector element.
Atanh	32f, 64f	Computes inverse hyperbolic tangent of each vector element.
Special Functions		
Erf	32f, 64f	Computes the error function value.
Erfc	32f, 64f	Computes the complementary error function value

NOTE. *You should not confuse fixed-accuracy arithmetic functions described in this chapter with functions in chapter 5 that have similar functionality but follow different accuracy specifications.*

Intel IPP fixed-accuracy arithmetic functions may return status codes of normal execution (`IppStsNoErr`), error conditions `IppStsSizeErr`, `IppStsNullPtrErr` (see Table 2-2 in Chapter 2), and the following specific warnings: `IppStsDomain`, `IppStsSingularity`, `IppStsOverflow`, `IppStsUnderflow`. In case of warnings, the value returned is positive and the computation is continued.
Table 12-2 lists status codes and the corresponding messages for these warnings.

Table 12-2 Warning Status Codes for Fixed-Accuracy Arithmetic Functions

Status	Value	Message
`IppStsOverflow`	12	Overflow occurred in the operation.
`IppStsUnderflow`	17	Underflow occurred in the operation.
`IppStsSingularity`	18	Singularity occurred in the operation.
`IppStsDomain`	19	Argument is out of the function domain.

See Appendix A, "Handling of Special Cases" for more information on function operation in cases when their arguments take on specific values that are outside the range of function definition.

Power and Root Functions

Inv

Computes inverse value of each vector element.

```
IppStatus ippsInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsInv is declared in the ippvm.h file. This function computes the inverse value of each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor ippsInv_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsInv_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsInv_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsInv_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsInv_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{pSrc[n]}, 0 \le n < len .$$

Example 12-1 shows how to use the function ippsInv.

Example 12-1 Using ippsInv Function

```
IppStatus ippsInv_32f_A21_sample(void)
{
const Ipp32f x[4] = {-9.975, 1.272, -6.134, 6.175};
Ipp32f       y[4];

    IppStatus st = ippsInv_32f_A21( x, y, 4 );

    printf(" ippsInv_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInv_32f_A21:
 x = -9.975 1.272 -6.134 6.175
 y = -0.100 0.786 -0.163 0.162
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsSingularity	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Div

Divides each element of the first vector by corresponding element of the second vector.

```
IppStatus ippsDiv_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsDiv_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f*  pDst, int len);
IppStatus ippsDiv_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f*  pDst, int len);
IppStatus ippsDiv_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsDiv_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsDiv is declared in the ippvm.h file. This function divides each element of the vector *pSrc1* by the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsDiv_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsDiv_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsDiv_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsDiv_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsDiv_64f_A53 guarantees 53 correctly rounded bits of significand, including the impede bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as $pDst[n] = \frac{pSrc1[n]}{pSrc2[n]}, 0 \le n < len$.

Example 12-2 shows how to use the function ippsDiv.

Example 12-2 Using ippsDiv Function

```
IppStatus ippsDiv_32f_A21_sample(void)
{
const Ipp32f x1[4] = {599.088, 735.034, 572.448, 151.640};
const Ipp32f x2[4] = {385.297, 609.005, 361.403, 225.182};
Ipp32f       y[4];

    IppStatus st = ippsDiv_32f_A21( x1, x2, y, 4 );

    printf(" ippsDiv_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}

Output results:

 ippsDiv_32f_A21:
 x1 = 599.088 735.034 572.448 151.640
 x2 = 385.297 609.005 361.403 225.182
 y  = 1.555 1.207 1.584 0.673
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsSingularity	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc2* is equal to 0.

Sqrt

Computes square root of each vector element.

```
IppStatus ippsSqrt_32f_A11 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_32f_A21 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_32f_A24 ( const Ipp32f*  pSrc, Ipp32f*  pDst, int len );
IppStatus ippsSqrt_64f_A50 ( const Ipp64f*  pSrc, Ipp64f*  pDst, int len );
IppStatus ippsSqrt_64f_A53 ( const Ipp64f*  pSrc, Ipp64f*  pDst, int len );
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsSqrt is declared in the ippvm.h file. This function computes square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsSqrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsSqrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsSqrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsSqrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsSqrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the impede bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sqrt{pSrc[n]}, 0 \le n < len$.

Example 12-3 shows how to use the function `ippsSqrt`.

Example 12-3 Using `ippsSqrt` Function

```
IppStatus ippsSqrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {5850.093, 4798.730, 3502.915, 8959.624};
Ipp32f       y[4];

    IppStatus st = ippsSqrt_32f_A21( x, y, 4 );

    printf(" ippsSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSqrt_32f_A21:
 x = 5850.093 4798.730 3502.915 8959.624
 y = 76.486 69.273 59.185 94.655
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`IppStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *`pSrc`* is less than 0.

InvSqrt

Computes inverse square root of each vector element.

```
IppStatus ippsInvSqrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvSqrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

`pSrc`	Pointer to the source vector.
`pDst`	Pointer to the destination vector.
`len`	Number of elements in the vectors.

Discussion

The function `ippsInvSqrt` is declared in the `ippvm.h` file. This function computes inverse square root of each element of *`pSrc`* and stores the result in the corresponding element of *`pDst`*.

For single precision data:

function flavor `ippsInvSqrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInvSqrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInvSqrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInvSqrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInvSqrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{\sqrt{pSrc[n]}}, 0 \le n < len.$$

Example 12-4 shows how to use the function `ippsInvSqrt`.

Example 12-4 Using `ippsInvSqrt` Function

```
 IppStatus ippsInvSqrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {7105.043, 5135.398, 3040.018, 149.944};
Ipp32f       y[4];

    IppStatus st = ippsInvSqrt_32f_A21( x, y, 4 );

    printf(" ippsInvSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInvSqrt_32f_A21:
 x = 7105.043 5135.398 3040.018 149.944
 y = 0.012 0.014 0.018 0.082
```

Example 12-4 Using `ippsInvSqrt` Function (continued)

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`IppStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *`pSrc`* is less than 0.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *`pSrc`* is equal to 0.

Cbrt

Computes cube root of each vector element.

```
IppStatus ippsCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

`pSrc`	Pointer to the source vector.
`pDst`	Pointer to the destination vector.
`len`	Number of elements in the vectors.

Discussion

The function `ippsCbrt` is declared in the `ippvm.h` file. This function computes cube root of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sqrt[3]{pSrc[n]}, 0 \le n < len$.

Example 12-5 shows how to use the function `ippsCbrt`.

Example 12-5 Using `ippsCbrt` Function

```
IppStatus ippsCbrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {6456.801, 4932.096, -6517.838, 7178.869};
Ipp32f       y[4];

    IppStatus st = ippsCbrt_32f_A21( x, y, 4 );

    printf(" ippsCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Example 12-5 Using ippsCbrt Function (continued)

```
Output results:

 ippsCbrt_32f_A21:
 x = 6456.801 4932.096 -6517.838 7178.869
 y = 18.621 17.022 -18.680 19.291
```

Return Value

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

InvCbrt

Computes inverse cube root of each vector element.

```
IppStatus ippsInvCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function `ippsInvCbrt` is declared in the `ippvm.h` file. This function computes inverse cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsInvCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInvCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInvCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInvCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInvCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \frac{1}{\sqrt[3]{pSrc[n]}}, 0 \le n < len.$$

Example 12-6 shows how to use the function `ippsInvCbrt`.

Example 12-6 Using `ippsInvCbrt` Function

```
IppStatus ippsInvCbrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {914.120, 3644.584, 1473.214, 1659.070};
Ipp32f       y[4];

    IppStatus st = ippsInvCbrt_32f_A21( x, y, 4 );

    printf(" ippsInvCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Example 12-6 Using `ippsInvCbrt` Function (continued)

```
Output results:

 ippsInvCbrt_32f_A21:
 x = 914.120 3644.584 1473.214 1659.070
 y = 0.103 0.065 0.088 0.084
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *`pSrc`* is equal to 0.

Pow

Raises each element of the first vector to the power of corresponding element of the second vector.

```
IppStatus ippsPow_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsPow_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
```

Arguments

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsPow is declared in the ippvm.h file. This function raises each element of vector *pSrc1* to the power of the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsPow_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsPow_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsPow_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsPow_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsPow_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = (pSrc1[n])^{pSrc2[n]}, 0 \le n < len$.

Example 12-7 shows how to use the function ippsPow.

Example 12-7 Using ippsPow Function

```
IppStatus ippsPow_32f_A21_sample(void)
{
```

Example 12-7 Using ippsPow Function (continued)

```
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
Ipp32f       y[4];

    IppStatus st = ippsPow_32f_A21( x1, x2, y, 4 );

    printf(" ippsPow_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}


Output results:

 ippsPow_32f_A21:
 x1 = 0.483 0.565 0.776 0.252
 x2 = 0.823 0.991 0.411 0.692
 y  = 0.549 0.568 0.901 0.386
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc1`* or *`pSrc2`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`IppStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one pair of the source elements meets the following condition: element of *`pSrc1`* is finite, less than 0, and element of *`pSrc2`* is finite, non-integer.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of *`pSrc1`* is equal to 0, and element of *`pSrc2`* is integer and less than 0.

Powx

Raises each element of a vector to the constant power.

```
IppStatus ippsPowx_32f_A11 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_32f_A21 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_32f_A24 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
    Ipp32f* pDst, int len);
IppStatus ippsPowx_64f_A50 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
    Ipp64f* pDst, int len);
IppStatus ippsPowx_64f_A53 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
    Ipp64f* pDst, int len);
```

Arguments

pSrc1	Pointer to the source vector.
ConstValue	Constant value.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsPowx is declared in the ippvm.h file. This function raises each element of the vector *pSrc1* to the constant power *ConstValue* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsPowx_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsPowx_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsPowx_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsPowx_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsPowx_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = (pSrc1[n])^{ConstValue}, 0 \le n < len$.

Example 12-8 shows how to use the function ippsPowx.

Example 12-8 Using ippsPowx Function

```
IppStatus ippsPowx_32f_A21_sample(void)
{
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2 = 0.823;
Ipp32f       y[4];

    IppStatus st = ippsPowx_32f_A21( x1, x2, y, 4 );

    printf(" ippsPowx_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f \n", x2);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}


Output results:

 ippsPowx_32f_A21:
 x1 = 0.483 0.565 0.776 0.252
 x2 = 0.823
 y  = 0.549 0.568 0.901 0.386
```

Return Value

ippStsNoErr	Indicates no error.

`ippStsNullPtrErr`	Indicates an error when *`pSrc1`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`ippStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one pair of the elements meets the following condition: element of *`pSrc1`* is finite, less than 0, and *`ConstValue`* is finite, non-integer.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of *`pSrc1`* is equal to 0, and *`ConstValue`* is integer and less than 0.

Exponential and Logarithmic Functions

Exp

Raises e to the power of each vector element.

```
IppStatus ippsExp_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsExp is declared in the ippvm.h file. This function raises *e* to the power of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsExp_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsExp_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsExp_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsExp_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsExp_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = e^{pSrc[n]}, 0 \le n < len$.

Example 12-9 shows how to use the function ippsExp.

Example 12-9 Using ippsExp Function

```
IppStatus ippsExp_32f_A21_sample(void)
{
const Ipp32f x[4] = {4.885, -0.543, -3.809, -4.953};
Ipp32f      y[4];

    IppStatus st = ippsExp_32f_A21( x, y, 4 );

    printf(" ippsExp_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsExp_32f_A21:
 x = 4.885 -0.543 -3.809 -4.953
 y = 132.324 0.581 0.022 0.007
```

Return Value

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

`IppStsOverflow`	Indicates a warning that the function overflows, that is, at least one of elements of *`pSrc`* is greater than `Ln(FPMAX)`, where `FPMAX` is the maximum representable floating-point number.
`IppStsUnderflow`	Indicates a warning that the function underflows, that is, at least one of elements of *`pSrc`* is less than `Ln(FPMIN)`, where `FPMIN` is the minimum positive floating-point value.

Ln

Computes natural logarithm of each vector element.

```
IppStatus ippsLn_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

`pSrc`	Pointer to the source vector.
`pDst`	Pointer to the destination vector.
`len`	Number of elements in the vectors.

Discussion

The function `ippsLn` is declared in the `ippvm.h` file. This function computes a natural logarithm of each element of *`pSrc`* and stores the result in the corresponding element of *`pDst`*.

For single precision data:

function flavor ippsLn_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsLn_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsLn_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsLn_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsLn_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \log_e(pSrc[n]), 0 \le n < len$.

Example 12-10 shows how to use the function ippsLn.

Example 12-10 Using ippsLn Function

```
IppStatus ippsLn_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.188, 3.841, 5.363, 5.755};
Ipp32f       y[4];

    IppStatus st = ippsLn_32f_A21( x, y, 4 );

    printf(" ippsLn_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsLn_32f_A21:
```

Example 12-10 Using ippsLn Function (continued)

```
x = 0.188 3.841 5.363 5.755
y = -1.670 1.346 1.680 1.750
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.
IppStsSingularity	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Log10

Computes common logarithm of each vector element.

```
IppStatus ippsLog10_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLog10_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsLog10 is declared in the ippvm.h file. This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsLog10_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsLog10_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsLog10_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsLog10_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsLog10_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \log_{10}(pSrc[n]), 0 \le n < len.$

Example 12-11 shows how to use the function ippsLog10.

Example 12-11 Using ippsLog10 Function

```
IppStatus ippsLog10_32f_A21_sample(void)
{
const Ipp32f x[4] = {6.057, 6.111, 1.746, 6.664};
Ipp32f       y[4];

    IppStatus st = ippsLog10_32f_A21( x, y, 4 );

    printf(" ippsLog10_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
```

Example 12-11 Using `ippsLog10` Function (continued)

```
}

Output results:

 ippsLog10_32f_A21:
 x = 6.057 6.111 1.746 6.664
 y = 0.782 0.786 0.242 0.824
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`IppStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *`pSrc`* is less than 0.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *`pSrc`* is equal to 0.

Trigonometric Functions

Cos

Computes cosine of each vector element.

```
IppStatus ippsCos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsCos is declared in the ippvm.h file. This function computes a cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsCos_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsCos_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsCos_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsCos_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsCos_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \cos(pSrc[n]), 0 \le n < len$.

Example 12-12 shows how to use the function ippsCos.

Example 12-12 Using ippsCos Function

```
IppStatus ippsCos_32f_A21_sample(void)
{
const Ipp32f x[4] = {-984.222, -2957.549, -8859.218, 2153.691};
Ipp32f       y[4];

    IppStatus st = ippsCos_32f_A21( x, y, 4 );
```

Example 12-12 Using ippsCos Function (continued)

```
    printf(" ippsCos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsCos_32f_A21:
 x = -984.222 -2957.549 -8859.218 2153.691
 y = -0.619 -0.258 0.997 0.129
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

Sin

Computes sine of each vector element.

```
IppStatus ippsSin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.

len Number of elements in the vectors.

Discussion

The function ippsSin is declared in the ippvm.h file. This function computes a sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsSin_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsSin_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsSin_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsSin_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsSin_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sin(pSrc[n]), 0 \le n < len$.

Example 12-13 shows how to use the function ippsSin.

Example 12-13 Using ippsSin Function

```
IppStatus ippsSin_32f_A21_sample(void)
{
const Ipp32f x[4] = {5666.372, 6052.125, 397.656, -3960.997};
Ipp32f      y[4];

    IppStatus st = ippsSin_32f_A21( x, y, 4 );

    printf(" ippsSin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
```

Example 12-13 Using ippsSin Function (continued)

```
}

Output results:

 ippsSin_32f_A21:
 x = 5666.372 6052.125 397.656 -3960.997
 y = -0.873 0.988 0.970 -0.524
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

SinCos

Computes sine and cosine of each vector element.

```
IppStatus ippsSinCos_32f_A11 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A21 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A24 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A50 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A53 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
```

Arguments

pSrc	Pointer to the first source vector.

pDst1	Pointer to the destination vector for sine values.
pDst2	Pointer to the destination vector for cosine values.
len	Number of elements in the vectors.

Discussion

The function ippsSinCos is declared in the ippvm.h file. This function computes sine of each element of *pSrc* and stores the result in the corresponding element of *pDst1*; computes cosine of each element of *pSrc* and stores the result in the corresponding element of *pDst2*.

For single precision data:

function flavor ippsSinCos_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsSinCos_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsSinCos_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsSinCos_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsSinCos_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst1[n] = \sin(pSrc[n]),\ pDst2[n] = \cos(pSrc[n]),\ 0 \le n < len.$

Example 12-14 shows how to use the function ippsSinCos.

Example 12-14 Using ippsSinCos Function

```
IppStatus ippsSinCos_32f_A21_sample(void)
{
const Ipp32f x[4] = {3857.845, -3939.024, -1468.856, -8592.486};
Ipp32f       y1[4];
Ipp32f       y2[4];
```

Example 12-14 Using ippsSinCos Function (continued)

```
    IppStatus st = ippsSinCos_32f_A21( x, y1, y2, 4 );

    printf(" ippsSinCos_32f_A21:\n");
    printf(" x  = %.3f %.3f %.3f %.3f \n", x[0],  x[1],  x[2],  x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
    return st;
}

Output results:

 ippsSinCos_32f_A21:
 x  = 3857.845 -3939.024 -1468.856 -8592.486
 y1 = -0.031 0.508 0.987 0.228
 y2 = 1.000 0.861 0.161 -0.974
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pDst1`* or *`pDst2`* or *`pSrc`* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`IppStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of the *`pSrc`* elements is equal to ±`INF`.

Tan

Computes tangent of each vector element.

```
IppStatus ippsTan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsTan is declared in the ippvm.h file. This function computes the tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsTan_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsTan_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsTan_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsTan_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsTan_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows: $pDst[n] = \tan(pSrc[n]), 0 \le n < len$.

Example 12-15 shows how to use the function ippsTan.

Example 12-15 Using ippsTan Function

```
IppStatus ippsTan_32f_A21_sample(void)
{
const Ipp32f x[4] = {7519.456, 4533.524, 9118.015, 8514.359};
Ipp32f       y[4];

    IppStatus st = ippsTan_32f_A21( x, y, 4 );
```

Example 12-15 Using ippsTan Function (continued)

```
    printf(" ippsTan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsTan_32f_A21:
 x = 7519.456 4533.524 9118.015 8514.359
 y = -18.656 0.209 2.028 0.750
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

Acos

Computes inverse cosine of each vector element.

```
IppStatus ippsAcos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAcos is declared in the ippvm.h file. This function computes the inverse cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAcos_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAcos_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAcos_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAcos_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAcos_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{acos}(pSrc[n]), 0 \le n < len.$

Example 12-16 shows how to use the function ippsAcos.

Example 12-16 Using ippsAcos Function

```
IppStatus ippsAcos_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.079, -0.715, -0.076, -0.529};
Ipp32f       y[4];

    IppStatus st = ippsAcos_32f_A21( x, y, 4 );

    printf(" ippsAcos_32f_A21:\n");
```

Example 12-16 Using ippsAcos Function (continued)

```
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAcos_32f_A21:
 x = 0.079 -0.715 -0.076 -0.529
 y = 1.492 2.368 1.647 2.129
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has an absolute value greater than 1.

Asin

Computes inverse sine of each vector element.

```
IppStatus ippsAsin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAsin is declared in the ippvm.h file. This function computes the inverse sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAsin_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAsin_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAsin_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAsin_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAsin_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows: $pDst[n] = \text{asin}(pSrc[n]), 0 \le n < len$.

Example 12-17 shows how to use the function ippsAsin.

Example 12-17 Using ippsAsin Function

```
IppStatus ippsAsin_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.724, -0.581, 0.559, 0.687};
Ipp32f      y[4];

    IppStatus st = ippsAsin_32f_A21( x, y, 4 );

    printf(" ippsAsin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Example 12-17 Using ippsAsin Function (continued)

```
Output results:

 ippsAsin_32f_A21:
x = 0.724 -0.581 0.559 0.687
y = 0.810 -0.620 0.594 0.758
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has an absolute value greater than 1.

Atan

Computes inverse tangent of each vector element.

```
IppStatus ippsAtan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAtan is declared in the ippvm.h file. This function computes the inverse tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAtan_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAtan_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAtan_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAtan_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAtan_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{atan}(pSrc[n]), 0 \le n < len$.

Example 12-18 shows how to use the function ippsAtan.

Example 12-18 Using ippsAtan Function

```
IppStatus ippsAtan_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.994, 0.999, 0.223, -0.215};
Ipp32f       y[4];

    IppStatus st = ippsAtan_32f_A21( x, y, 4 );

    printf(" ippsAtan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
```

Example 12-18 Using `ippsAtan` Function (continued)

```
}

Output results:

 ippsAtan_32f_A21:
 x = 0.994 0.999 0.223 -0.215
 y = 0.782 0.785 0.219 -0.212
```

Return Value

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

Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

```
IppStatus ippsAtan2_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
    pDst, int len);
IppStatus ippsAtan2_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
IppStatus ippsAtan2_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
    pDst, int len);
```

Arguments

`pSrc1`	Pointer to the first source vector.
`pSrc2`	Pointer to the second source vector.

pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAtan2 is declared in the ippvm.h file. This function computes the angle between the X axis and the line from the origin to the point (X,Y), for each element of *pSrc1* as a Y (the ordinate) and corresponding element of *pSrc2* as an X (the abscissa), and stores the result in the corresponding element of *pDst*. The result angle varies from - π to + π.

For single precision data:

function flavor ippsAtan2_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAtan2_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAtan2_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAtan2_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAtan2_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \arctan2(pSrc1[n], pSrc2[n]) \quad , 0 \le n < len .$

Example 12-19 shows how to use the function ippsAtan2.

Example 12-19 Using ippsAtan2 Function

```
IppStatus ippsAtan2_32f_A21_sample(void)
{
const Ipp32f x1[4] = {1.492, 1.700, 1.147, 1.142};
const Ipp32f x2[4] = {1.064, 1.505, 1.950, 1.905};
Ipp32f       y[4];
```

Example 12-19 Using ippsAtan2 Function (continued)

```
    IppStatus st = ippsAtan2_32f_A21( x1, x2, y, 4 );

    printf(" ippsAtan2_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}


Output results:

 ippsAtan2_32f_A21:
 x1 = 1.492 1.700 1.147 1.142
 x2 = 1.064 1.505 1.950 1.905
 y  = 0.951 0.846 0.532 0.540
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Hyperbolic Functions

Cosh

Computes hyperbolic cosine of each vector element.

```
IppStatus ippsCosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsCosh is declared in the ippvm.h file. This function computes the hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsCosh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsCosh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsCosh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsCosh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsCosh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \cosh(pSrc[n]), 0 \le n < len$.

Example 12-20 shows how to use the function ippsCosh.

Example 12-20 Using ippsCosh Function

```
IppStatus ippsCosh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-4.676, -4.054, 6.803, -9.525};
Ipp32f       y[4];

    IppStatus st = ippsCosh_32f_A21( x, y, 4 );

    printf(" ippsCosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsCosh_32f_A21:
 x = -4.676 -4.054 6.803 -9.525
 y = 53.661 28.833 450.219 6849.870
```

Return Value

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

`IppStsOverflow`	Indicates a warning that the function overflows, that is, at least one of elements of *`pSrc`* has the absolute value greater than `Ln(FPMAX) + Ln(2)`, where `FPMAX` is the maximum representable floating-point number.

Sinh

Computes hyperbolic sine of each vector element.

```
IppStatus ippsSinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

`pSrc`	Pointer to the source vector.
`pDst`	Pointer to the destination vector.
`len`	Number of elements in the vectors.

Discussion

The function `ippsSinh` is declared in the `ippvm.h` file. This function computes the hyperbolic sine of each element of *`pSrc`*, and stores the result in the corresponding element of *`pDst`*.

For single precision data:

function flavor ippsSinh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsSinh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsSinh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsSinh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsSinh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sinh(pSrc[n]), 0 \le n < len$.

Example 12-21 shows how to use the function ippsSinh.

Example 12-21 Using ippsSinh Function

```
IppStatus ippsSinh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-2.483, -8.148, 3.544, -8.876};
Ipp32f      y[4];

    IppStatus st = ippsSinh_32f_A21( x, y, 4 );

    printf(" ippsSinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSinh_32f_A21:
 x = -2.483 -8.148 3.544 -8.876
 y = -5.945 -1727.412 17.290 -3577.970
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *`pSrc`* or *`pDst`* pointer is null.
`ippStsSizeErr`	Indicates an error when *`len`* is less than or equal to 0.
`IppStsOverflow`	Indicates a warning that the function overflows, that is, at least one of elements of *`pSrc`* has the absolute value greater than `Ln(FPMAX) + Ln(2)`, where `FPMAX` is the maximum representable floating-point number.

Tanh

Computes hyperbolic tangent of each vector element.

```
IppStatus ippsTanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

`pSrc`	Pointer to the source vector.
`pDst`	Pointer to the destination vector.
`len`	Number of elements in the vectors.

Discussion

The function `ippsTanh` is declared in the `ippvm.h` file. This function computes the hyperbolic tangent of each element of *`pSrc`* and stores the result in the corresponding element of *`pDst`*.

For single precision data:

function flavor ippsTanh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsTanh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsTanh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsTanh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsTanh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[x] = \tanh(pSrc[n]), 0 \le n < len$.

Example 12-22 shows how to use the function ippsTanh.

Example 12-22 Using ippsTanh Function

```
IppStatus ippsTanh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f       y[4];

    IppStatus st = ippsTanh_32f_A21( x, y, 4 );

    printf(" ippsTanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsTanh_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

Example 12-22 Using ippsTanh Function (continued)

Return Value

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

Acosh

Computes inverse (nonnegative) hyperbolic cosine of each vector element.

```
IppStatus ippsAcosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAcosh is declared in the ippvm.h file. This function computes the inverse (nonnegative) hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAcosh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAcosh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAcosh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAcosh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAcosh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[x] = \text{acosh}(pSrc[n]), 0 \le n < len.$

Example 12-23 shows how to use the function ippsAcosh.

Example 12-23 Using ippsAcosh Function

```
IppStatus ippsAcosh_32f_A21_sample(void)
{
const Ipp32f x[4] = {588.321, 691.492, 837.773, 726.767};
Ipp32f      y[4];

    IppStatus st = ippsAcosh_32f_A21( x, y, 4 );

    printf(" ippsAcosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAcosh_32f_A21:
 x = 588.321 691.492 837.773 726.767
 y = 7.070 7.232 7.424 7.282
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 1.

Asinh

Computes inverse hyperbolic sine of each vector element.

```
IppStatus ippsAsinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAsinh is declared in the ippvm.h file. This function computes the inverse hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAsinh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAsinh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAsinh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAsinh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAsinh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{asinh}(pSrc[n]), 0 \le n < len.$

Example 12-24 shows how to use the function ippsAsinh.

Example 12-24 Using ippsAsinh Function

```
IppStatus ippsAsinh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-30.122, -589.282, 487.472, -63.082};
Ipp32f       y[4];

    IppStatus st = ippsAsinh_32f_A21( x, y, 4 );

    printf(" ippsAsinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAsinh_32f_A21:
 x = -30.122 -589.282 487.472 -63.082
 y = -4.099 -7.072 6.882 -4.838
```

Return Value

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

Atanh

Computes inverse hyperbolic tangent of each vector element.

```
IppStatus ippsAtanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsAtanh is declared in the ippvm.h file. This function computes the inverse hyperbolic tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsAtanh_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsAtanh_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsAtanh_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsAtanh_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsAtanh_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{atanh}(pSrc[n]), 0 \le n < len.$

Example 12-25 shows how to use the function ippsAtanh.

Example 12-25 Using ippsAtanh Function

```
IppStatus ippsAtanh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.076, 0.808, 0.440, -0.705};
Ipp32f       y[4];

    IppStatus st = ippsAtanh_32f_A21( x, y, 4 );

    printf(" ippsAtanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAtanh_32f_A21:
 x = -0.076 0.808 0.440 -0.705
 y = -0.076 1.123 0.472 -0.877
```

Return Value

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is null.
`ippStsSizeErr`	Indicates an error when len is less than or equal to 0.
`IppStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has absolute value greater than 1.
`IppStsSingularity`	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* has absolute value equal to 1.

Special Functions

Erf

Computes the error function value.

```
IppStatus ippsErf_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsErf_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsErf is declared in the ippvm.h file. This function computes the error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsErf_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsErf_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsErf_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsErf_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsErf_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{erf}(pSrc[n]), 0 \le n < len$, where $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

Example 12-26 shows how to use the function ippsErf.

Example 12-26 Using ippsErf Function

```
IppStatus ippsErf_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f       y[4];

    IppStatus st = ippsErf_32f_A21( x, y, 4 );

    printf(" ippsErf_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
```

Example 12-26 Using ippsErf Function (continued)

```
    return st;
}

Output results:

 ippsErf_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

Return Value

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

Erfc

Computes the complementary error function value.

```
IppStatus ippsErfc_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsErfc_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

Arguments

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Discussion

The function ippsErfc is declared in the ippvm.h file. This function computes the complementary error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor ippsErfc_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsErfc_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsErfc_32f_A24 guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor ippsErfc_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsErfc_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{erfc}(pSrc[n]), 0 \le n < len$, where $\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

Example 12-27 shows how to use the function ippsErfc.

Example 12-27 Using ippsErfc Function

```
IppStatus ippsErfc_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f       y[4];

    IppStatus st = ippsErfc_32f_A21( x, y, 4 );

    printf(" ippsErfc_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
```

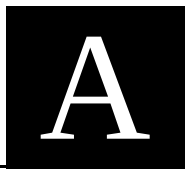
Example 12-27 Using ippsErfc Function (continued)

```
    return st;
}

Output results:

 ippsErfc_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

Return Value

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is null.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsUnderflow	Indicates a warning that the function underflows, that is, at least one of *pSrc* elements is less than some threshold value, where the function result is less than the minimum positive floating-point value in target precision.

Handling of Special Cases



Some mathematical functions implemented in Intel IPP are not defined for all possible argument values. This appendix describes how the corresponding Intel IPP functions used in signal processing data-domain handle situations when their input arguments fall outside the range of function definition or may lead to ambiguously determined output results.

Table A-1 below summarizes these special cases for general vector functions described in chapter 5 and lists result values together with status codes returned by these functions.

Table A-1 Special Cases for Intel IPP Signal Processing Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsSqrt	16s	Sqrt (x <0)	0	ippStsSqrtNegArg
	32f	Sqrt (x <0)	NAN_32F	ippStsSqrtNegArg
	64s	Sqrt (x <0)	0	ippStsSqrtNegArg
	64f	Sqrt (x <0)	NAN_64F	ippStsSqrtNegArg
ippsDiv	8u	Div (0/0)	0	ippStsDivByZero
		Div (x/0)	IPP_MAX_8U	ippStsDivByZero
	16s	Div (0/0)	0	ippStsDivByZero
		Div (x/0), x>0	IPP_MAX_16S	ippStsDivByZero
		Div (x/0), x<0	IPP_MIN_16S	ippStsDivByZero
	16sc	Div (0/0)	0	ippStsDivByZero
		Div (x/0)	0	ippStsDivByZero

Table A-1 Special Cases for Intel IPP Signal Processing Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsDiv	32f	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0), x>0	INF_32F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_32F	ippStsDivByZero
	32fc	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0)	NAN_32F	ippStsDivByZero
				ippStsDivByZero
	64f	Div (0/0)	NAN_64F	
		Div (x/0), x>0	INF_64F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_64F	ippStsDivByZero
	64fc	Div (0/0)	NAN_64F	ippStsDivByZero
		Div (x/0)	NAN_64F	ippStsDivByZero
ippsDivC	All	Div(x/0)	x	ippStsDivByZeroErr
ippsLn	16s, 32s	Ln (0)	0	ippStsLnZeroArg
		Ln (x<0)	0	ippStsLnNegArg
	32f	Ln (x<0)	NAN_32F	ippStsLnNegArg
		Ln(x<IPP_MINABS_32F)	INF_NEG_32F	ippStsLnZeroArg
	64f	Ln (x<0)	NAN_64F	ippStsLnNegArg
		Ln(x<IPP_MINABS_64F)	INF_NEG_64F	ippStsLnZeroArg
ippsExp	16s	overflow	IPP_MAX_16S	ippStsNoErr
	32s	overflow	IPP_MAX_32S	ippStsNoErr
	64s	overflow	IPP_MAX_64S	ippStsNoErr
	32f	overflow	INF_32F	ippStsNoErr
	64f	overflow	INF_64F	ippStsNoErr

Table A-1 Special Cases for Intel IPP Signal Processing Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsThreshold, ippsThresholdLT, ippsThresholdGT, ippsThresholdLTVal, ippsThresholdGTVal	16sc	level<0	x	ippStsThreshNegLevelErr
	32fc	level<0	x	ippStsThreshNegLevelErr
	64fc	level<0	x	ippStsThreshNegLevelErr
ippsThresholdLTInv	32f	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_32F	ippStsInvZero
	32fc	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_32F	ippStsInvZero
	64f	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_64F	ippStsInvZero
	64fc	level<0	x	ippStsThreshNegLevelErr
		level=0, x=0	INF_64F	ippStsInvZero

Here x denotes an input value. For the definition of the constants used, see "Data Ranges" in chapter 2.

Note that flavors of the same math function operating on different data types may produce unlike results for the equal argument values. However, for a given function and a fixed data type, handling of special cases is the same for all function flavors that have different descriptors in their names. For example, logarithm function `ippsLn` operating on `16s` data treats zero argument values in the same way for all its flavors `ippsLn_16s_Sfs` and `ippsLn_16s_ISfs`.

Table A-2 below summarizes special cases for fixed accuracy arithmetic vector functions described in Chapter 12, "Fixed-Accuracy Arithmetic Functions".

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsInv	32f	Inv (x =+0)	INF_32F	ippStsSingularity
		Inv (x =-0)	-INF_32F	ippStsSingularity
	64f	Inv (x =+0)	INF_64F	ippStsSingularity
		Inv (x =-0)	-INF_64F	ippStsSingularity
ippsDiv	32f	Div (x>0, y=+0)	INF_32F	ippStsSingularity
		Div (x>0, y=-0)	-INF_32F	ippStsSingularity
		DIv (x<0, y=+0)	-INF_32F	ippStsSingularity
		Div (x<0, y=-0)	INF_32F	ippStsSingularity
		Div (x=0, y=0)	NAN_32F	ippStsSingularity
	64f	Div (x>0, y=+0)	INF_64F	ippStsSingularity
		Div (x>0, y=-0)	-INF_64F	ippStsSingularity
		DIv (x<0, y=+0)	-INF_64F	ippStsSingularity
		Div (x<0, y=-0)	INF_64F	ippStsSingularity
		Div (x=0, y=0)	NAN_64F	ippStsSingularity
ippsSqrt	32f	Sqrt (x<0)	NAN_32F	ippStsDomain
		Sqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	Sqrt (x<0)	NAN_64F	ippStsDomain
		Sqrt (x=-INF)	NAN_64F	ippStsDomain
ippsInvSqrt	32f	InvSqrt (x<0)	NAN_32F	ippStsDomain
		InvSqrt (x=+0)	INF_32F	ippStsSingularity
		InvSqrt (x=-0)	-INF_32F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	InvSqrt (x<0)	NAN_64F	ippStsDomain
		InvSqrt (x=+0)	INF_64F	ippStsSingularity
		InvSqrt (x=-0)	-INF_64F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_64F	ippStsDomain

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsInvCbrt	32f	InvCbrt (x=+0)	INF_32F	ippStsSingularity
		InvCbrt (x=-0)	-INF_32F	ippStsSingularity
	64f	InvCbrt (x=+0)	INF_64F	ippStsSingularity
		InvCbrt (x=-0)	-INF_64F	ippStsSingularity
ippsPow	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity
ippsPowx	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsExp	32f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_32F	ippStsOverflow
	64f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_64F	ippStsOverflow
ippsLn	32f	Ln (x<0)	NAN_32F	ippStsDomain
		Ln (x=-INF)	NAN_32F	ippStsDomain
		Ln (x=0)	-INF_32F	ippStsSingularity
	64f	Ln (x<0)	NAN_64F	ippStsDomain
		Ln (x=-INF)	NAN_64F	ippStsDomain
		Ln (x=0)	-INF_64F	ippStsSingularity
ippsLog10	32f	Ln (x<0)	NAN_32F	ippStsDomain
		Ln (x=-INF)	NAN_32F	ippStsDomain
		Ln (x=0)	-INF_32F	ippStsSingularity
	64f	Ln (x<0)	NAN_64F	ippStsDomain
		Ln (x=-INF)	NAN_64F	ippStsDomain
		Ln (x=0)	-INF_64F	ippStsSingularity
ippsCos	32f	Cos (INF)	NAN_32F	ippStsDomain
	64f	Cos (INF)	NAN_64F	ippStsDomain
ippsSin	32f	Sin (INF)	NAN_32F	ippStsDomain
	64f	Sin (INF)	NAN_64F	ippStsDomain
ippsSinCos	32f	SinCos (INF)	NAN_32F, NAN_32F	ippStsDomain
	64f	SinCos (INF)	NAN_64F, NAN_64F	ippStsDomain
ippsTan	32f	Tan (INF)	NAN_32F	ippStsDomain
	64f	Tan (INF)	NAN_64F	ippStsDomain
ippsAcos	32f	Acos (x), |x|>1	NAN_32F	ippStsDomain
		Acos (INF)	NAN_32F	ippStsDomain
	64f	Acos (x), |x|>1	NAN_64F	ippStsDomain
		Acos (INF)	NAN_64F	ippStsDomain

Table A-2 Special Cases for Intel IPP Fixed-Accuracy Mathematical Functions (continued)

Function Base Name	Data Type	Case Description	Result Value	Status Code
ippsAsin	32f	Acos (x), |x|>1	NAN_32F	ippStsDomain
		Acos (INF)	NAN_32F	ippStsDomain
	64f	Acos (x), |x|>1	NAN_64F	ippStsDomain
		Acos (INF)	NAN_64F	ippStsDomain
ippsCosh	32f	Cosh (x), |x|>overflow	INF_32F	ippStsOverflow
	64f	Cosh (x), |x|>overflow	INF_64F	ippStsOverflow
ippsSinh	32f	Sinh (x), |x|>overflow	INF_32F	ippStsOverflow
	64f	Sinh (x), |x|>overflow	INF_64F	ippStsOverflow
ippsAcosh	32f	Acosh (x<1)	NAN_32F	ippStsDomain
		Acosh (x=-INF)	NAN_32F	ippStsDomain
	64f	Acosh (x<1)	NAN_64F	ippStsDomain
		Acosh (x=-INF)	NAN_64F	ippStsDomain
ippsAtanh	32f	Atanh (x=1)	INF_32F	ippStsSingularity
		Atanh (x=-1)	-INF_32F	ippStsSingularity
		Atanh (x), |x|>1	NAN_32F	ippStsDomain
		Atanh (INF)	NAN_32F	ippStsDomain
	64f	Atanh (x=1)	INF_64F	ippStsSingularity
		Atanh (x=-1)	-INF_64F	ippStsSingularity
		Atanh (x), |x|>1	NAN_64F	ippStsDomain
		Atanh (INF)	NAN_64F	ippStsDomain

Bibliography

This bibliography provides a list of reference books and other sources of additional information that might be useful to the application programmer. This list is neither complete nor exhaustive, but serves as a starting point. Of all the references listed, [Mit93] will be the most useful to those readers who already have a basic understanding of signal processing. This reference collects the work of 27 experts in the field and has both great breadth and depth.
The books [Opp75], [Opp89], [Jac89], and [Zie83] are undergraduate signal processing texts. [Opp89] is a much revised edition of the classic [Opp75]; [Jac89] is more concise than the others; and [Zie83] also covers continuous-time systems.

[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.

[Bri94] C. Brislawn, *Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks*. Los Alamos Report LA-UR-94-1747, 1994.

[Cap78] V. Cappellini, A. G. Constantinides, and P. Emilani. *Digital Filters and Their Applications*. Academic Press, London, 1978.

[Cap94] Cappé O., *Elimination of the musical noise phenomenon with the Ephraim and Malah noise suppressor*. IEEE Trans. Speech and Audio Processing, vol. 2(2), (1994).

[CCITT] CCITT, Recommendation G.711. *Pulse Code Modulation of Frequencies*, 1984.

[Coh02] I. Cohen and B. Berdugo. *Noise Estimation by Minima Controlled Recursive Averaging for Robust Speech Enhancemen*t. IEEE Signal Proc. Letters, Vol. 9, No. 1, Jan. 2002, pp. 12-15.

[Cro83] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets.* Springer Verlag, Pennsylvania, 1992.

[Eph84] Y. Ephraim and D. Malah. *Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator.* IEEE Trans. ASSP, Vol. 32, No. 6, Dec. 1984, pp. 1109-1121.

[ES201] ETSI ES 201 108 V1.1.2. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Front-end feature extraction algorithm; Compression algorithms.*

[ES202] ETSI ES 202 050 V1.1.1. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Advanced front-end feature extraction algorithm; Compression algorithms.*

[Fei92] E. Feig and S. Winograd. *Fast algorithms for DCT.* IEEE Transactions on Signal Processing, vol.40, No.9, 1992.

[Ham83] R.W. Hamming, *Digital Filters*, Prentice-Hall, New Jersey 1983.

[Har78] F. Harris. *On the Use of Windows*. Proceedings of the IEEE, vol. 66, No.1, IEEE, 1978.

[Hay91] S. Haykin. *Adaptive Filter Theory.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[ISO11172] ISO/IEC 11172-3 - Information technology. *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s.* Part 3: Audio (1993)

[ISO13818] ISO/IEC 13818-3 - Information technology. *Generic coding of moving pictures and associated audio information.* Part 3: Audio (1998).

[ISO14496] ISO/IEC 14496-3 - Information technology. *Coding of audio-visual objects.* Part 3: Audio (2001).

[ITU729] ITU-T Recommendation G.729. *Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)*, (03/96).

[ITU729A] ITU-T Recommendation G.729 Annex A. *Reduced Complexity 8 kbit/s CS-ACELP speech codec*, (11/96).

[ITU729B] ITU-T Recommendation G.729 Annex B. *A silence compression scheme for G.729 optimized for terminals conforming to Recommendation V.70*, (10/96).

[ITU729B1] ITU-T Recommendation G.729 Annex B Corrigendum 1, (02/98).

[ITU723] ITU-T Recommendation G.723.1 . *Dual Rate speech coder for Multimedia Communications transmitting at 5.3 and 6.3 Kbit/s*, (03/96).

[ITU723A] ITU-T Recommendation G.723.1 Annex A. *Silence compression scheme*, (11/96).

[ITUV34] ITU-T Recommendation V.34. *A modem operating at data signalling rates of up to 33 600 bit/s for use on the general switched telephone network and on leased point-to-point 2-wire telephone-type circuit.* (02/98).

[ITU722] ITU-T Recommendation G.722.1 (09/99), *Coding at 24 and 32 8 kbit/s for hand-free operation in systems with low frame loss*.

[Jac89] Leland B. Jackson. *Digital Filters and Signal Processing*. Kluwer Academic Publishers, second edition, 1989.

[Kab86] P. Kabal and P.Ramachandran. *The Computation of line Spectral Frequencies Using Chebyshev Polynomials*. IEEE transaction on acoustic, speech and signal processing, vol. ASSP-34, No.6, 1986.

[Lyn89] Paul A. Lynn. *Introductory Digital Signal Processing with Computer Applications*. John Wiley&Sons, Inc., New York, 1993.

[Mar01] R. Martin. *Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics*. IEEE Trans. Speech and Audio, Vol. 9, No. 5, July 2001, pp. 504-512.

[Med91] Y. Medan, E. Yair, D. Chazan. *Super Resolution Pitch Determination of Speech Signals*. IEEE Transactions on Signal Processing, vol 39, No.1, 1991.

[Mit93] Sanjit K. Mitra and James F. Kaiser editors. *Handbook for Digital Signal Processing*. John Wiley&Sons, Inc., New York, 1993.

[Mit98] S. K. Mitra. *Digital Signal Processing*. McGraw Hill, 1998.

[NIC91] Nam Ik Cho and Sang Uk Lee. *Fast algorithm and implementation of 2D DCT*. IEEE Transactions on Circuits and Systems, vol. 31, No.3, 1991.

[Opp75] Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Opp89] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Rab78] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Rao90] K.R. Rao and P. Yip. *Discrete Cosine Transform. Algorithms, Advantages and Applications*. Academic Press, San Diego, 1990.

[Seg78] R. Sedgewick. *Implementing quicksort programs*. Communications of the ACM, Vol. 21, No. 10, pp. 847-857, Oct. 1978.

[Str96] G. Strang and T. Nguyen. *Wavelet and Filter Banks*. Wellesley-Cambridge Press, 1996.

[Tuc92] R. Tucker. *Voice Activity Detection Using a Periodicity Measure*. *I*EE Proceedings-I, Vol. 139, No. 4, August 1992, pp. 377-380.

[Vai93] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, Englewood Cliffs, New Jersey.

[Wid85] B. Widrow and S.D. Stearns. *Adaptive Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Zie83] Rodger E. Ziemer, William H. Tranter and D. Ronald Fannin. *Signals and Systems: Continuous and Discrete*. Macmillan Publishing Co., New York, 1983.

Glossary

adaptive filter	An adaptive filter varies its filter coefficients (taps) over time. Typically, the filter's coefficients are varied to make its output match a prototype "desired" signal as closely as possible. Non-adaptive filters do not vary their filter coefficients over time.
arithmetic operation	An operation that adds, subtracts, multiplies, or squares the image pixel values.
BQ	One of the modes, which indicates that the IIR initialization function initializes a cascade of biquads.
CCS	See complex conjugate-symmetric.
companding functions	The functions that perform an operation of data compression by using a logarithmic encoder-decoder. Companding allows you to maintain the percentage error constant by logarithmically spacing the quantization levels.
complex conjugate-symmetric	A kind of symmetry that arises in the Fourier transform of real signals. A complex conjugate-symmetric signal has the property that `x(-n) = x(n)*`, where "`*`" denotes conjugation.
conjugate	The conjugate of a complex number `a + bj` is `a - bj`.
conjugate-symmetric	See complex conjugate-symmetric.
DCT	Acronym for the discrete cosine transform.
decimation	Filtering a signal followed by down-sampling. Filtering prevents aliasing distortion in the subsequent down-sampling. See down-sampling.

down-sampling	Down-sampling conceptually decreases a signal's sampling rate by removing samples from between neighboring samples of a signal. See decimation.
element-wise	An element-wise operation performs the same operation on each element of a vector, or uses the elements of the same position in multiple vectors as inputs to the operation. For example, the element-wise addition of the vectors $\{x_0, x_1, x_2\}$ and $\{y_0, y_1, y_2\}$ is performed as follows: $\{x_0, x_1, x_2\} + \{y_0, y_1, y_2\} = \{x_0 + y_0, x_1 + y_1, x_2 + y_2\}$.
FIR	Abbreviation for finite impulse response filter. Finite impulse response filters do not vary their filter coefficients (taps) over time.
FIR LMS	Abbreviation for least mean squares finite impulse response filter.
fixed-point data format	A format that assigns one bit for a sign and all other bits for fractional part. This format is used for optimized conversion operations with signed, purely fractional vectors. For example, S.31 format assumes a sign bit and 31 fractional bits; S15.16 assumes a sign bit, 15 integer bits, and 16 fractional bits.
IIR	Abbreviation for infinite impulse response filters.
in-place	A function that performs its operation in-place, takes its input from an array and returns its output to the same array. See not-in-place.
interpolation	Up-sampling a signal followed by filtering. The filtering gives the inserted samples a value close to the samples of their neighboring samples in the original signal. See up-sampling.
LMS	Abbreviation for least mean square, an algorithm frequently used as a measure of the difference between two signals. Also used as shorthand for an adaptive FIR filter employing the LMS algorithm for adaptation.

LTI	Abbreviation for linear time-invariant systems. In LTI systems, if an input consists of the sum of a number of signals, then the output is the sum of the system's responses to each signal considered separately [Lyn89].
MMX™ technology	An enhancement to Intel architecture aimed at better performance in multimedia and communications applications. The technology uses four additional data types, eight 64-bit MMX registers, and 57 additional instructions implementing the SIMD (single instruction, multiple data) technique.
MR	One of the modes, indicating the multi-rate variety of the function.
multi-rate	An operation or signal processing system involving signals with multiple sample rates. Decimation and interpolation are examples of multi-rate operations.
not-in-place	A function that performs its operation not-in-place takes its input from a source array and puts its output in a second, destination array.
polyphase	A computationally efficient method for multi-rate filtering. For example, interpolation or decimation.
CCS	A representation of a complex conjugate-symmetric sequence which is easier to use than the Pack or Perm formats.
Pack	A compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms ("natural" in the sense that bit-reversed order is natural for radix-2 complex FFTs).
Perm	A format for storing the values for the FFT algorithm. RCPerm format stores the values in the order in which the FFT algorithm uses them. That is, the real and imaginary parts of a given sample need not be adjacent.

saturation	Using saturation arithmetic, when a number exceeds the data-range limit for its data type, it saturates to the upper data-range limit. For example, a signed word greater than 7FFFh saturates to 7FFFh. When a number is less than the lower data-range limit, it saturates to the lower data-range. For example, a signed word less than 8000h saturates to 8000h.
sinusoid	See tone.
Streaming SIMD Extensions	The major enhancement to Intel architecture instruction set. Incorporates a group of general-purpose floating-point instructions operating on packed data, additional packed integer instructions, together with cacheability control and state management instructions. These instructions significantly improve performance of applications using compute-intensive processing of floating-point and integer data.
tone	A sinusoid of a given frequency, phase, and magnitude. Tones are used as test signals and as building blocks for more complex signals.
up-sampling	Up-sampling conceptually increases the signal sampling rate by inserting zero-valued samples between neighboring samples of a signal.
window	A mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

Index

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z