Intel® Integrated Performance Primitives

Developer Reference, Volume 1: Signal Processing

Intel IPP 2017 Update 2

Legal Information

Contents

Chapter 4: Vector Initialization Functions

Chapter 5: Essential Functions

Chapter 6: Filtering Functions

Chapter 7: Transform Functions

Chapter 8: String Functions

Chapter 9: Fixed-Accuracy Arithmetic Functions

Chapter 10: Data Compression Functions

Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications. Current characterized errata are available on request.

MPEG-1, MPEG-2, MPEG-4, H.261, H.263, H.264, MP3, DV, VC-1, MJPEG, AC3, AAC, G.711, G.722, G.722.1, G.722.2, AMRWB, Extended AMRWB (AMRWB+), G.167, G.168, G.169, G.723.1, G.726, G.728, G.729, G.729.1, GSM AMR, GSM FR are international standards promoted by ISO, IEC, ITU, ETSI, 3GPP and other organizations. Implementations of these standards, or the standard enabled platforms may require licenses from various entities, including Intel Corporation.

Cilk, Intel, the Intel logo, Intel Atom, Intel Core, Intel Inside, Intel NetBurst, Intel SpeedStep, Intel vPro, Intel Xeon Phi, Intel XScale, Itanium, MMX, Pentium, Thunderbolt, Ultrabook, VTune and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

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

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Third Party

Intel® Integrated Performance Primitives (Intel® IPP) includes content from several 3rd party sources that was originally governed by the licenses referenced below:

- zlib library:

  zlib.h -- interface of the 'zlib' general purpose compression library version 1.2.8, April 28th, 2013

  Copyright (C) 1995-2013 Jean-loup Gailly and Mark Adler

  This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

  Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
  2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
  3. This notice may not be removed or altered from any source distribution.

  Jean-loup Gailly Mark Adler

  jloup@gzip.org madler@alumni.caltech.edu
- bzip2:

  Copyright © 1996 - 2015 julian@bzip.org

© Intel Corporation.

Volume Overview

1

This manual describes the structure, operation and functions of the Intel® Integrated Performance Primitives (Intel® IPP) for Intel® architecture that operate on one-dimensional signals. The manual explains the concepts of Intel IPP, 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. The functions are combined in groups by their functionality. Each group of functions is described in a separate chapter (chapters 3 through 14).

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

What's New

This Developer Reference documents Intel® Integrated Performance Primitives (Intel® IPP) 2017 Update 2 release.

Notational Conventions

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

This manual uses the following notational conventions:

Convention	Explanation	Example
`THIS TYPE STYLE`	Used in the text for the Intel IPP constant identifiers.	`IPPI_MAX_64S`
`This type style`	Mixed with the uppercase in structure names; also used in function names, code examples and call statements.	`IppLibraryVersion`, `void ippsFree()`
`This type style`	Parameters in function prototypes and parameters description.	*`value`*, *`srcStep`*
x(n) and *x*[*n*	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 ≤*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*. 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.

Convention	Explanation	Example
`Ipp<data-domain>` and `Ipp` prefixes	All structures and enumerators, specific for a particular data-domain have the `Ipp<data-domain>` prefix, while those common for entire Intel IPP software have the `Ipp` prefix.	`IppsROI`, `IppLibraryVersion`

See Also

Function Naming

Intel® Integrated Performance Primitives Concepts

2

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

Function Naming

Naming conventions for the Intel IPP functions are similar for all covered domains.

Function names in Intel IPP have the following general format:

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

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

> **NOTE**
> In this document, each function is introduced by its short name (without the `ipps` prefix and modifiers) and a brief description of its purpose.
>
> 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.

Data-Domain

The *data-domain* element is a single character that denotes the group of functionality to which a given function belongs. The main distinction among these groups is the type of input data. Intel IPP supports the following data-domains:

`s`	signal processing (input data is a 1D signal)
`i`	images and video processing (input data is a 2D image)
`m`	small matrix operations (input data is a matrix)
`r`	realistic rendering functionality and 3D data processing (type of input data type depends on supported rendering techniques)
`g`	operations on signals of the fixed length

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

Name

The *name* element identifies what function does and has the following format:

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

The *operation* component is one ore more words, acronyms, and abbreviations that describe the core operation.

The *modifier* component, if present, is a word or abbreviation that denotes a slight modification or variation of the given function.

For example, names without modifiers: `Add`, `Threshold`, `FirGenLowPass`; with modifiers: `ippsFFTInv_CToC`, `Threshold_LT`.

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".

Intel IPP supports the data types of the source and destination for signal processing functions listed in the table below.

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

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> = <src1Datatype>[src2Datatype][dstDatatype]`

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 *dstDatatype* 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.

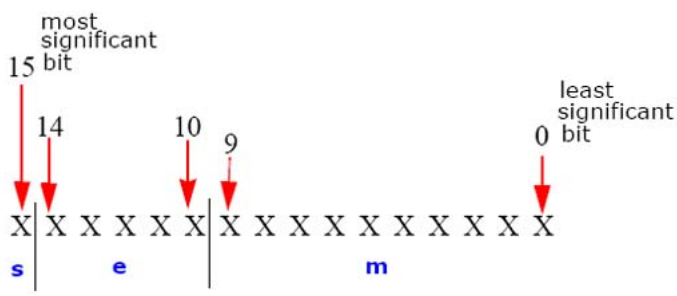
There are several data types, namely `24u`, `24s` and `16f` that are not supported by Intel IPP, but can be readily converted to the supported data types for further processing by the library functions.

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`.

Descriptors

The *`descriptors`* element further describes the operation. Descriptors are individual characters that indicate additional details of the operation.

The following descriptors are used in signal processing functions:

Descriptor	Description	Example
`I`	Operation is in-place (default is not-in-place).	`ippsAdd_16s_I`
`Sfs`	Saturation and fixed scaling mode (default is saturation and no scaling).	`ippsConvert_16s8s_Sfs`
`P`	Operation is performed for the specified number of vectors.	`ippsIIR_32f _P`

If the function has more than one descriptor, they are presented in the function name in alphabetical order.

Many functions have no descriptors listed above. Such functions operate with the default behavior.

Parameters

The *`parameters`* element specifies the function parameters (arguments).

The order of parameters is as follows:

- All source operands. Constants follow vectors.
- All destination operands. Constants follow vectors.

- Other, operation-specific parameters.

A parameter name has the following conventions:

- All parameters defined as pointers start with *p*, defined as double pointers start with *pp*, for example, *pPhase*, *pSrc*, *ppState*. All parameters defined as values start with a lowercase letter, for example, *val*, *src*, *srcLen*.
- Each new part of a parameter name starts with an uppercase character, without underscore; for example, *pSrc*, *lenSrc*, *pDlyLine*.
- Each parameter name specifies its functionality. Source parameters are named *pSrc* or *src*, in some cases followed by names or numbers, for example, *pSrc2*, *srcLen*. Output parameters are named *pDst* or *dst* followed by names or numbers, for example, *pDst2*, *dstLen*. For in-place operations, the input/output parameter contains the name *pSrcDst* or *srcDst*.

Extensions

The *extension* field denotes an Intel IPP extension to which the function belongs. The following extensions are supported in Intel IPP Signal Processing functions:

Extension	Description	Example
L	Intel IPP platform-aware functions	ippsMalloc_16u_L

See Also

Platform-Aware Functions in Signal Processing

Structures and Enumerators

This section describes the structures and enumerators used by 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*, containing version numbers;
- integer field *majorBuild*, containing update number;
- integer field *build*, containing build revision number;
- string field *Name*, containing the Intel IPP version name, for example, "ippSB SSE4.1";
- string field *Version*, containing the version description, for example, "7.1.0 (r93873)".
- string field *BuildDate*, containing the build date.

Complex Data Structures

Complex numbers in Intel IPP are described by the structures that contain two numbers of the respective data type. They are real and imaginary parts of the complex number. For example, a 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 use special structures to store function-specific (context) 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:

- specification structures that are not modified during function operation; they have the suffix `Spec` in their names
- state structures that are modified during operation; they have the suffix `State` in their names.

The function context interpretation is processor dependent. Therefore, these context-related structures are not defined in the public headers, and their fields are not accessible. Intel IPP provides no option of modifying these structures or creating 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 is error-free. See section Error Reporting in this chapter for more information on the set of valid status values and corresponding error messages for signal processing functions.

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

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

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

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

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

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

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

```
typedef enum {
 /* Enumeration:           Processor:                                */
    ippCpuUnknown = 0x0,
    ippCpuPP,           /* Intel(R) Pentium(R) processor             */
    ippCpuPMX,          /* Pentium(R) processor
                           with MMX(TM) technology                   */
    ippCpuPPR,          /* Pentium(R) Pro processor                  */
    ippCpuPII,          /* Pentium(R) II  processor                  */
    ippCpuPIII,         /* Pentium(R) III processor
                           and Pentium(R)III Xeon(R) processor       */
```

```
    ippCpuP4,           /* Pentium(R) 4 processor
                           and Intel(R) Xeon(R) processor            */
    ippCpuP4HT,         /* Pentium(R) 4 processor with HT Technology */
    ippCpuP4HT2,        /* Pentium(R) 4 processor with Intel(R)
                                   Streaming SIMD Extensions 3     */
    ippCpuCentrino,     /* Intel(R) Centrino(R) processor technology   */
    ippCpuCoreSolo,     /* Intel(R) Core(TM) Solo processor          */
    ippCpuCoreDuo,      /* Intel(R) Core(TM) Duo processor           */
    ippCpuITP   = 0x10, /* Intel(R) Itanium(R) processor             */
    ippCpuITP2          /* Intel(R) Itanium(R) 2 processor           */
    ippCpuEM64T = 0x20, /* Intel(R) 64 Instruction Set
                                       Architecture(ISA)           */
    ippCpuC2D,          /* Intel(R) Core(TM) 2 Duo processor         */
    ippCpuC2Q,          /* Intel(R) Core(TM) 2 Quad processor        */
    ippCpuPenryn,     /* Intel(R) Core(TM) 2 processor with
                                  Intel(R) SSE4.1                  */
    ippCpuBonnell,                  /* Intel(R) Atom (TM) processor */
    ippCpuNehalem,      /* Intel (R) Core(TM) i7 processor
    ippCpuNext,
    ippCpuSSE   = 0x40, /* Processor supports Pentium(R) III
                           processor instruction set                 */
    ippCpuSSE2,         /* Processor supports Intel(R) Streaming SIMD
                           Extensions 2 instruction set              */
    ippCpuSSE3,         /* Processor supports Intel(R) Streaming SIMD
                            Extensions 3 instruction set             */
    ippCpuSSSE3,        /* Processor supports Supplemental Streaming
                           SIMD Extensions 3 instruction set         */
    ippCpuSSE41,        /* Processor supports Intel(R) Streaming SIMD
                            Extensions 4.1 instruction set           */
    ippCpuSSE42,        /* Processor supports Intel(R) Streaming SIMD
                            Extensions 4.2 instruction set           */
    ippCpuAVX,        /* Processor supports Intel(R) Advanced Vector
                            Extensions instruction set               */
    ippCpuAES,        /* Processor supports Intel(R) AES
                            new instructions                         */
ippCpuX8664 = 0x60,   /* Processor supports 64 bit extension         */
} IppCpuType;
```

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

The `IppWinType` enumeration defines the type of window to be used by the FIR filter coefficient generating functions:

```
typedef enum {
    ippWinBartlett,
    ippWinBlackman,
    ippWinHamming,
```

```
    ippWinHann,
    ippWinRect
} IppWinType;
```

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

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

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

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

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

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

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

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

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

```
typedef enum {
    IppLZ77InflateStatusInit,
    IppLZ77InflateStatusHuffProcess
    IppLZ77InflateStatusLZ77Process,
    IppLZ77InflateStatusFinal
} IppLZ77InflateStatus;
```

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

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

The `IppInflateState` enumeration defines the decoding parameters that are used by the ZLIB data compression functions:

```
typedef struct IppInflateState {
  const Ipp8u* pWindow;        // pointer to the sliding window
                                 // (the dictionary for the LZ77 algorithm)
  unsigned int winSize;          // size of the sliding window
  unsigned int tableType;       // type of Huffman code tables
                                 // (for example, 0 - tables for
Fixed                                                        //  Huffman codes)
  unsigned int tableBufferSize;  //(ENOUGH = 2048) * (sizeof(code) = 4) -
                                              // sizeof(IppInflateState)
} IppInflateState;
```

The `IppInflateMode` enumeration defines the decode mode that is used by the ZLIB data compression functions:

```
typedef enum {
  ippTYPE,
  ippLEN,
  ippLENEXT
} IppInflateMode;
```

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

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

The `IppAffinityType` enumeration defines which affinity type is used by the core function ippSetAffinity:

```
typedef enum {
    ippAffinityCompactFineCore,
    ippAffinityCompactFineHT,
    ippAffinityCompactAllEnabled,
    ippAffinityCompactRestore,
    ippTstAffinityCompactFineCore,
    ippTstAffinityCompactFineHT
} IppAffinityType;
```

The `IppEnum` enumeration defines the configuration of the algorithm for some functions:

```
typedef int IppEnum;
```

The `IppAlgType` enumeration defines the type of the algorithm implementation:

```
typedef enum {
    ippAlgAuto     = 0x00000000, // default
    ippAlgDirect   = 0x00000001,
    ippAlgFFT      = 0x00000002,
    ippAlgMask    = 0x000000FF,
} IppAlgType;
```

The `IppsNormOp` enumeration defines the type of normalization that should be applied to the output data:

```
typedef enum {
    ippsNormNone  = 0x00000000,// default
    ippsNormA        = 0x00000100,// biased normalization
    ippsNormB        = 0x00000200,// unbiased normalization
    ippsNormMask  = 0x0000FF00,
} IppsNormOp;
```

The `IppFourSymb` structure used in Long Term Evolution (LTE) Wireless Support Functions stores the destination data grouped by four symbols:

```
typedef struct {
     Ipp16sc symb[4];
} IppFourSymb;
```

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:

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

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.

You can also use the `ippsMalloc` function to align the allocated memory pointer on 64 bytes.

Rounding Mode

General signal processing functions use rounding. The default rounding mode is *nearest even*, that is 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.

Some functions have additional rounding modes, which are set by the parameter *roundMode*.

Important

- Functions for data compression, data integrity, string processing and fixed-accuracy arithmetic do not perform rounding.

Integer Scaling

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

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

> **NOTE**
> The result of integer operations is always saturated to the destination data type range.

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 even integer number (see "Rounding Mode").

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 $11*2^{-3} = 1.375$.

See Also

Rounding Mode

Error Reporting

The Intel IPP functions return the status of the performed operation to report errors and warnings to the calling program. The last value of the error status is not stored, and you need to decide whether to check it or not as the function returns. The status values are of the `IppStatus` type and are global constant integers.

The following table lists status codes and corresponding messages reported by Intel IPP for signal processing.

Error Status Values and Messages

Status	Message
`ippStsCpuNotSupportedErr`	The target cpu is not supported.
`ippStsPointAtInfinity`	Point at infinity is detected.
`ippStsI18nUnsupportedErr`	Internationalization (`i18n`) is not supported.
`ippStsI18nMsgCatalogOpenErr`	Message Catalog cannot be opened. For detailed information, use `errno` on Linux* OS and `GetLastError` on Windows* OS.
`ippStsI18nMsgCatalogCloseErr`	Message Catalog cannot be closed. For detailed information, use `errno` on Linux* OS and `GetLastError` for Windows* OS.
`ippStsUnknownStatusCodeErr`	Unknown status code.
`ippStsOFBSizeErr`	Wrong value for crypto OFB block size.
`ippStsLzoBrokenStreamErr`	LZO safe decompression function cannot decode LZO stream.
`ippStsRoundModeNotSupportedErr`	Rounding mode is not supported.
`ippStsMaxLenHuffCodeErr`	Huff: Max length of the Huffman code is more than the expected one.
`ippStsCodeLenTableErr`	Huff: Invalid `codeLenTable`.
`ippStsFreqTableErr`	Huff: Invalid `freqTable`.
`ippStsRegExpOptionsErr`	RegExp: Options for the pattern are incorrect.

ippStsRegExpErr	RegExp: The structure pRegExpState contains wrong data.
ippStsRegExpMatchLimitErr	RegExp: The match limit has been exhausted.
ippStsRegExpQuantifierErr	RegExp: Incorrect quantifier.
ippStsRegExpGroupingErr	RegExp: Incorrect grouping.
ippStsRegExpBackRefErr	RegExp: Incorrect back reference.
ippStsRegExpChClassErr	RegExp: Incorrect character class.
ippStsRegExpMetaChErr	RegExp: Icnorrect metacharacter.
ippStsMP3FrameHeaderErr	Error in fields of the IppMP3FrameHeader structure.
ippStsMP3SideInfoErr	Error in fields of the 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 is not supported.
ippStsAacSectErr	AAC: Invalid number of sections.
ippStsAacTnsNumFiltErr	AAC: Invalid number of TNS filters.
ippStsAacTnsLenErr	AAC: Invalid length of TNS region.
ippStsAacTnsOrderErr	AAC: Invalid order of TNS filter.
ippStsAacTnsCoefResErr	AAC: Invalid bit-resolution for TNS filter coefficients.
ippStsAacTnsCoefErr	AAC: Invalid coefficients of TNS filter.
ippStsAacTnsDirectErr	AAC: Invalid direction of TNS filter.
ippStsAacTnsProfileErr	AAC: Invalid TNS profile.
ippStsAacErr	AAC: Internal error.
ippStsAacBitOffsetErr	AAC: Invalid current bit offset in bitstream.
ippStsAacAdtsSyncWordErr	AAC: Invalid ADTS syncword.
ippStsAacSmplRateIdxErr	AAC: Invalid sample rate index.
ippStsAacWinLenErr	AAC: Invalid window length (not short or long).
ippStsAacWinGrpErr	AAC: Invalid number of groups for current window length.
ippStsAacWinSeqErr	AAC: Invalid window sequence range.
ippStsAacComWinErr	AAC: Invalid common window flag.
ippStsAacStereoMaskErr	AAC: Invalid stereo mask.
ippStsAacChanErr	AAC: Invalid channel number.
ippStsAacMonoStereoErr	AAC: Invalid mono-stereo flag.
ippStsAacStereoLayerErr	AAC: Invalid Stereo Layer flag.
ippStsAacMonoLayerErr	AAC: Invalid 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: Table format is not supported.
ippStsVLCUsrTblEscAlgTypeErr	VLC: Ecs-algorithm is not supported.
ippStsVLCUsrTblEscCodeLengthErr	VLC: Esc-code length inside table header is incorrect.
ippStsVLCUsrTblCodeLengthErr	VLC: Code length inside table is incorrect.
ippStsVLCInternalTblErr	VLC: Invalid internal table.
ippStsVLCInputDataErr	VLC: Invalid input data.
ippStsVLCAACEscCodeLengthErr	VLC: Invalid AAC-Esc code length.
ippStsIncorrectLSPErr	Incorrect values for linear spectral pair.

ippStsNoRootFoundErr	No roots are found for equation.
ippStsLengthErr	Incorrect value for string length.
ippStsFBankFreqErr	Incorrect value for the filter bank frequency parameter.
ippStsFBankFlagErr	Incorrect value for the filter bank parameter.
ippStsFBankErr	Filter bank is not correctly initialized.
ippStsNegOccErr	Ocupation count is negative.
ippStsCdbkFlagErr	Incorrect value for the codebook flag parameter.
ippStsSVDCnvgErr	No convergence of the SVD algorithm.
ippStsToneMagnErr	Tone magnitude is less than or equal to zero.
ippStsToneFreqErr	Tone frequency is negative, or greater than or equal to 0.5.
ippStsTonePhaseErr	Tone phase is negative, or greater than or equal to 2*PI.
ippStsTrnglMagnErr	Triangle magnitude is less than or equal to zero.
ippStsTrnglFreqErr	Triangle frequency is negative, or greater than or equal to 0.5.
ippStsTrnglPhaseErr	Triangle phase is negative, or greater than or equal to 2*PI.
ippStsTrnglAsymErr	Triangle asymmetry is less than -PI, or greater than or equal to PI.
ippStsHugeWinErr	The Kaiser window is too big.
ippStsJaehneErr	Magnitude value is negative.
ippStsStepErr	Step value is not valid.
ippStsDlyLineIndexErr	Invalid value for the delay line sample index.
ippStsStrideErr	Stride value is less than length of the row.
ippStsEpsValErr	Negative epsilon value.
ippStsScaleRangeErr	Scale bounds are out of range.
ippStsThresholdErr	Invalid threshold bounds.
ippStsWtOffsetErr	Invalid offset value for wavelet filter.
ippStsAnchorErr	Anchor point is outside the mask.
ippStsMaskSizeErr	Invalid mask size.
ippStsShiftErr	Shift value is less than zero.
ippStsSampleFactorErr	Sampling factor is less than or equal to zero.
ippStsSamplePhaseErr	Phase value is out of range, $0 \le phase <$ factor.
ippStsFIRMRFactorErr	MR FIR sampling factor is less than or equal to zero.
ippStsFIRMRPhaseErr	MR FIR sampling phase parameter is negative, or greater than or equal to the sampling factor.
ippStsRelFreqErr	Relative frequency value is out of range.
ippStsFIRLenErr	Length of the FIR filter is less than or equal to zero.
ippStsIIROrderErr	Order of the IIR filter is not valid.
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 does not match the operation.
ippStsFftFlagErr	Invalid value for the FFT flag parameter.
ippStsFftOrderErr	Invalid value for the FFT order parameter.
ippStsMemAllocErr	Not enough memory for the operation.
ippStsNullPtrErr	Null pointer error.
ippStsSizeErr	Incorrect value for data size.
ippStsBadArgErr	Incorrect argument/parameter of the function.
ippStsErr	Unknown/unspecified error.
ippStsNoErr	No errors.
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	Argument value is not a number.
ippStsResFloor	All result values are floored.
ippStsOverflow	Overflow in the operation.
ippStsZeroOcc	Zero occupation count.
ippStsUnderflow	Underflow in the operation.
ippStsSingularity	Singularity in the operation.
ippStsDomain	Argument is out of the function domain.
ippStsNotIntelCpu	The target cpu is not Genuine Intel.
ippStsCpuMismatch	Cannot set the library for the given cpu.
ippStsNotIppFunctionFound	Application does not contain IPP function calls.
ippStsDllNotFoundBestUsed	Dispatcher cannot find the newest version of IPP DLLs.
ippStsNoOperationInDll	The function does nothing in the dynamic version of the library.
ippStsOvermuchStrings	Number of destination strings is more than expected.
ippStsOverlongString	Length of one of the destination strings is more than expected.
ippStsSrcSizeLessExpected	DC: The size of source buffer is less than the expected one.
ippStsDstSizeLessExpected	DC: The size of destination buffer is less than the expected one.
ippStsNotSupportedCpu	The CPU is not supported.
ippStsUnkhownCacheSize	The CPU is supported, but the size of the cache is unknown.
ippStsAlgTypeErr	The algorithm type is not supported.

*)

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

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 interrupted. 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.

Platform-Aware Functions in Signal Processing

Intel® Integrated Performance Primitives (Intel® IPP) library provides so-called platform-aware functions. These functions use the special data type `IppSizeL` for object sizes. The `IppSizeL` data type represents memory-related quantities: it can be 32- or 64-bit wide depending on the target architecture.

While the rest of Intel IPP functions support only objects of 32-bit integer size, platform-aware functions can work with 64-bit object sizes if it is supported by the platform. The API of platform-aware functions is similar to the API of other Intel IPP functions and has only slight differences. You can distinguish platform-aware functions by the `L` suffix in the function name, for example, `ippsMalloc_16u_L`.

Currently, the following signal processing functions have platform-aware APIs:

Function Group	Header	Function Name
Support Functions	`ipps_l.h`	`Malloc`

Intel IPP platform-aware functions are documented as additional flavors to the existing functions declared in standard Intel IPP headers (without the `l` suffix). The `ipps_l.h` header is included into `ipps.h`.

Code Examples

The document contains a number of code examples that use the Intel IPP functions. These examples show both some particular features of the primitives and how the primitives can be called. Many of these code examples output result data together with the status code and associated messages in case of an error or a warning condition.

To keep the example code simpler, special definitions of print statements are used that get output strings look exactly the way it is needed for better representation of 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 document by straightforward copying and pasting the example code fragments.

```
#define genPRINT(TYPE,FMT) \
void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus st ) { \
   int n; \
   if( st > ippStsNoErr ) \
      printf( "\n-- warning %d, %s", st, ippGetStatusString( st )); \
   else if( st < ippStsNoErr ) \
      printf( "\n-- error %d, %s", st, ippGetStatusString( st )); \
   printf("\n %s \n", msg ); \
   for( n=0; n<len; ++n ) printf( FMT, buf[n] ); \
   printf("\n" ); \
}
genPRINT( 64f, " %f" )
genPRINT( 32f, " %f" )
genPRINT( 32u, " %u" )
genPRINT( 16s, " %d" )
genPRINT( 8u, " %u" )

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

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

#define genPRINT_2D(TYPE,FMT) \
void printf_##TYPE##_2D(const char* msg, Ipp##TYPE* buf, IppiSize roi, int step, IppStatus st ) { \
   int i, j; \
   if ( st > ippStsNoErr ) { \
      printf( "\n-- warning %d, %s", st, ippGetStatusString( st )); \
   } else if ( st < ippStsNoErr ) { \
      printf( "\n-- error %d, %s", st, ippGetStatusString( st )); \
   } \
   printf("\n %s \n", msg ); \
   for ( i=0; i<roi.height; i++ ) { \
      for ( j=0; j<roi.width; j++ ) { \
         printf( FMT, ((Ipp##TYPE*)(((Ipp8u*)buf) + i*step))[j] ); \
      } \
      printf("\n"); \
   } \
   printf("\n" ); \
}
genPRINT_2D( 8u, " %u" )
genPRINT_2D( 32f, " %.1f" )
```

Support Functions

3

This chapter describes Intel® IPP support functions. Use these functions 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
- Retrieve information about the processor and perform specific auxiliary operations
- Perform internationalization

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 the Intel IPP signal processing software.

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Description

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	Major number of the current library version.
minor	Minor number of the current library version.
majorBuild	Update number.
build	Build revision number.
targetCpu[4]	Intel® processor.
Name	Name of the current library version.
Version	Library version string.

BuildDate	Library version actual build date.

For example, if the library version is “9.0 ”, build revision number is “49671”, library name is “ippSP AVX2”, target CPU is processor with Intel® Advanced Vector Extensions 2 (Intel® AVX2) and build date is “Dec 7 2015”, then the fields in this structure are set as:

major = 9, *minor* = 0, *Name* = “ippSP AVX2”, *Version* = “9.0.1 (r49671)”, *targetCpu[4]*=“h9”, *BuildDate* = “Dec 7 2015”

NOTE
Each sub-library in the signal processing domain has its own similar function to retrieve information about the active library version. Version information functions for sub-libraries have the same interface as `ippsGetLibVersion`.

The following table provides the list of version information functions and respective header files where these functions are declared:

Function Name	Header File
`ippGetLibVersion`	`ippcore.h`
`ippccGetLibVersion`	`ippcc.h`
`ippcvGetLibVersion`	`ippcv.h`
`ippchGetLibVersion`	`ippch.h`
`ippdcGetLibVersion`	`ippdc.h`
`ippvmGetLibVersion`	`ippvm.h`

Example

The following example shows how to use the `ippsGetLibVersion` function :

```
const IppLibraryVersion* lib;
    lib = ippsGetLibVersion();

printf("major = %d\n",lib->major);
printf("minor = %d\n",lib->minor);
printf("majorBuild = %d\n",lib->majorBuild);
printf("build = %d\n",lib->build);
printf("targetCpu = %c%c%c%c\n",lib->targetCpu[0],lib->targetCpu[1],lib->targetCpu[2],lib-
>targetCpu[3]);
printf("Name = %s\n", lib->Name);
printf("Version = %s\n", lib->Version);
printf("BuildDate = %s\n", lib->BuildDate);
```

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 allocated elements *len*.

NOTE
Use the `ippsFree()` to free memory allocated by `ippsMalloc()`. Use `free` to free memory allocated by `malloc` or `calloc`.

Malloc

Allocates memory aligned to 64-byte boundary.

Syntax

Case 1: Memory allocation for blocks of 32-bit length

```
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);
```

Case 2: Memory allocation for platform-aware functions

```
Ipp8u* ippsMalloc_8u_L(IppSizeL len);
Ipp16u* ippsMalloc_16u_L(IppSizeL len);
Ipp32u* ippsMalloc_32u_L(IppSizeL len);
Ipp8s* ippsMalloc_8s_L(IppSizeL len);
Ipp16s* ippsMalloc_16s_L(IppSizeL len);
Ipp32s* ippsMalloc_32s_L(IppSizeL len);
Ipp64s* ippsMalloc_64s_L(IppSizeL len);
Ipp32f* ippsMalloc_32f_L(IppSizeL len);
Ipp64f* ippsMalloc_64f_L(IppSizeL len);
Ipp8sc* ippsMalloc_8sc_L(IppSizeL len);
Ipp16sc* ippsMalloc_16sc_L(IppSizeL len);
Ipp32sc* ippsMalloc_32sc_L(IppSizeL len);
Ipp64sc* ippsMalloc_64sc_L(IppSizeL len);
Ipp32fc* ippsMalloc_32fc_L(IppSizeL len);
```

```
Ipp64fc* ippsMalloc_64fc_L(IppSizeL len);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

len	Number of elements to allocate.

Description

This function allocates memory block aligned to 64-byte boundary for elements of different data types.

Example

The following example shows how to use theippsMalloc_8u function:

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

    ippsFree(pBuf);
}
```

Return Values

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

See Also

Free Frees memory allocated by the function ippsMalloc.

Free

Frees memory allocated by the function ippsMalloc.

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function frees the aligned memory block allocated by the function `ippsMalloc`.

> **NOTE**
> Use the `ippsFree()` to free memory allocated by `ippsMalloc()`. Use `free` to free memory allocated by `malloc` or `calloc`.

Common Functions

This section describes the Intel IPP functions that perform special operations common for all domains. All these functions are grouped in the separate sub-library called `ippcore`.

GetStatusString

Translates a status code into a message.

Syntax

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

Include Files

`ippcore.h`

Parameters

stsCode	Code that indicates the status type (see Error Status Values and Messages).

Description

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

Example

The following code example shows how to use the function `ippGetStatusString`. If you call an Intel IPP function `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"`.

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

Output:

```
-8, Null Pointer Error
```

GetCpuClocks

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

Syntax

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

Include Files

`ippcore.h`

Description

This function reads the current state of the TSC register and returns its value.

GetCpuFreqMhz

Estimates the processor operating frequency.

Syntax

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

Include Files

`ippcore.h`

Parameters

pMhz	Pointer to the result.

Description

This function estimates the processor operating frequency and returns its value, in MHz as an integer stored in *pMhz*. The estimated value can vary depending on the processor workload.

> **NOTE**
> To improve precision of the return value, this function accumulates CPU clocks. This operation takes several seconds and may result in long execution time.

Return Values

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

GetCpuFeatures

Retrieves the processor features.

Syntax

```
IppStatus ippGetCpuFeatures(Ipp64u* pFeaturesMask, Ipp32u pCpuidInfoRegs[4]);
```

Include Files

`ippcore.h`

Parameters

pFeaturesMask	Pointer to the features mask. Possible value is `ippCPUID_GETINFO_A`.
pCpuidInfoRegs	Pointer to the vector with four elements to store the data from the registers `eax`, `ebx`, `ecx`, `edx` of the function `CPUID.1`.

Description

This function retrieves some of the CPU features returned by the function CPUID.1 and stores them consecutively in the mask *pFeaturesMask*. The following table lists the features stored in the mask.

If *pFeaturesMask* does not have any input value, then the function retrieves the features in accordance with `eax=1` and `ecx=0`. If *pFeaturesMask* is set to `ippCPUID_GETINFO_A`, then the function retrieves the features in accordance with the input values of the registers `eax` and `ecx` that are specified in this case by the *pCpuidInfoRegs*[0] and *pCpuidInfoRegs*[2] respectively.

Mask Value	Bit Name	Feature	Mask Bit Number
0x00000001	`ippCPUID_MMX`	MMX™ technology	0
0x00000002	`ippCPUID_SSE`	Intel® Streaming SIMD Extensions	1
0x00000004	`ippCPUID_SSE2`	Intel® Streaming SIMD Extensions 2	2
0x00000008	`ippCPUID_SSE3`	Intel® Streaming SIMD Extensions 3	3
0x00000010	`ippCPUID_SSSE3`	Supplemental Streaming SIMD Extensions	4
0x00000020	`ippCPUID_MOVBE`	MOVBE instruction is supported	5
0x00000040	`ippCPUID_SSE41`	Intel® Streaming SIMD Extensions 4.1	6
0x00000080	`ippCPUID_SSE42`	Intel® Streaming SIMD Extensions 4.2	7
0x00000100	`ippCPUID_AVX`	The processor supports Intel® Advanced Vector Extensions (Intel® AVX) instruction set	8
0x00000200	`ippAVX_ENABLEDBYOS`	The operating system supports Intel® AVX	9
0x00000400	`ippCPUID_AES`	Advanced Encryption Standard (AES) instructions are supported	10
0x00000800	`ippCPUID_CLMUL`	PCLMULQDQ instruction is supported	11
0x00002000	`ippCPUID_RDRAND`	Read Random Number instructions are supported	13

Mask Value	Bit Name	Feature	Mask Bit Number
0x00004000	ippCPUID_F16C	16-bit floating point conversion instructions are supported	14
0x00008000	ippCPUID_AVX2	Intel® Advanced Vector Extensions 2 (Intel® AVX2) instruction set is supported	15
0x00010000	ippCPUID_ADCOX	ADCX and ADOX instructions are supported	16
0x00020000	ippCPUID_RDSEED	Read Random SEED instruction is supported.	17
0x00040000	ippCPUID_PREFETCHW	PREFETCHW instruction is supported	18
0x00080000	ippCPUID_SHA	Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) are supported	19
0x00100000	ippCPUID_AVX512F	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) foundation instructions are supported	20
0x00200000	ippCPUID_AVX512CD	Intel® AVX-512 conflict detection instructions are supported	21
0x00400000	ippCPUID_AVX512ER	Intel® AVX-512 exponential and reciprocal instructions are supported	22
0x80000000	ippCPUID_KNC	Intel® Xeon Phi™ is supported	23

All features returned by the `CPUID.1` function can be stored in the vector with four elements *`pCpuidInfoRegs`* where each element contains data from one of the registers `eax`, `ebx`, `ecx`, `edx` respectively. If these data are not required, the pointer *`pCpuidInfoRegs`* must be set to `NULL`.

NOTE
Intel® Itanium® processors are not supported.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition when the *`pFeaturesMask`* pointer is `NULL`.
`ippStsNotSupportedCpu`	Indicates that the processor is not supported.

GetEnabledCpuFeatures

Returns a features mask for enabled processor features.

Syntax

```
Ipp64u ippGetEnabledCpuFeatures (void);
```

Include Files

`ippcore.h`

Description

This function detects the enabled CPU features for the currently loaded libraries and returns the corresponding features mask.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

See Also

GetCpuFeatures Retrieves the processor features.

GetMaxCacheSizeB

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

Syntax

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

Include Files

`ippcore.h`

Parameters

pSizeByte	Pointer to the output result.

Description

This function finds the maximum size (in bytes) of the L2 and L3 caches of the processor used on your computer system. The result is stored in the *pSizeByte*.

> **NOTE**
> Intel® Itanium® processors are not supported.

If the processor is not supported, or size of cache is unknown, the result is `0`, and the function returns corresponding warning message.

Return Values

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

SetCpuFeatures

Sets the processor-specific library code for the specified processor features.

Syntax

```
IppStatus ippSetCpuFeatures(Ipp64u cpuFeatures);
```

Include Files

`ippcore.h`

Parameters

cpuFeatures	Features to be supported by the library. Refer to `ippdefs.h` for `ippCPUID_xx` definition.

Description

This function sets the processor-specific code of the Intel IPP library according to the processor features specified in `cpuFeatures`. You can use the following predefined sets of features (the `FM` suffix below means *feature mask*):

32-bit code:

```
#define PX_FM ( ippCPUID_MMX | ippCPUID_SSE )
#define W7_FM ( PX_FM | ippCPUID_SSE2 )
#define V8_FM ( W7_FM | ippCPUID_SSE3 | ippCPUID_SSSE3 )
#define S8_FM ( V8_FM | ippCPUID_MOVBE )
#define P8_FM ( V8_FM | ippCPUID_SSE41 | ippCPUID_SSE42 | ippCPUID_AES | ippCPUID_CLMUL |
ippCPUID_SHA )
#define G9_FM ( P8_FM | ippCPUID_AVX | ippAVX_ENABLEDBYOS | ippCPUID_RDRAND | ippCPUID_F16C )
#define H9_FM ( G9_FM | ippCPUID_MOVBE | ippCPUID_AVX2 | ippCPUID_ADCOX | ippCPUID_RDSEED |
ippCPUID_PREFETCHW )
```

64-bit code:

```
#define PX_FM ( ippCPUID_MMX | ippCPUID_SSE | ippCPUID_SSE2 )
#define M7_FM ( PX_FM | ippCPUID_SSE3 )
#define U8_FM ( M7_FM | ippCPUID_SSSE3 )
#define N8_FM ( U8_FM | ippCPUID_MOVBE )
#define Y8_FM ( U8_FM | ippCPUID_SSE41 | ippCPUID_SSE42 | ippCPUID_AES | ippCPUID_CLMUL |
ippCPUID_SHA )
#define E9_FM ( Y8_FM | ippCPUID_AVX | ippAVX_ENABLEDBYOS | ippCPUID_RDRAND | ippCPUID_F16C )
#define L9_FM ( E9_FM | ippCPUID_MOVBE | ippCPUID_AVX2 | ippCPUID_ADCOX | ippCPUID_RDSEED |
ippCPUID_PREFETCHW )
#define K0_FM ( L9_FM | ippCPUID_AVX512F )
```

NOTE
Do not use any other Intel IPP function while `ippSetCpuFeatures` is executing. Otherwise, your application behavior is undefined.

NOTE
To avoid initialization of internal structures for one Intel® architecture and then call of the processing function that is optimized for another architecture, do not use the `ippSetCpuFeatures` function in chains of Intel IPP connected calls like *<processing function*`GetSize` + *<processing function*`Init` + *<processing function>*. Otherwise, Intel IPP functionality behavior is undefined.

Intel IPP library supports two internal sets of CPU features:

- *Real CPU features*: the features that are supported by the CPU at which the library is executed. These features are read-only and can be obtained with the `ippGetCpuFeatures` function.
- *Enabled features*: the features that are enabled externally to Intel IPP by the application. These features are read-write and can be obtained with ippGetEnabledCpuFeatures

  and set with `ippSetCpuFeatures`.

The `ippSetCpuFeatures` function provides additional flexibility in measuring performance improvements reached by using specific CPU features. For example, the call of the `ippInit()` (or the first call of any Intel IPP function for the library version starting with 9.0) function in an application running on the 4th Generation Intel® Core™ i7 processor with 64-bit OS installed dispatches the L9 code version optimized for Intel® Advanced Vector Extensions 2 (Intel® AVX2) with several other features like fast 16-bit floating point support, Intel® AES New Instructions (Intel® AES-NI), PCLMULQDQ new instructions support.

To check performance improvement for all Intel IPP functionality reached by using Intel® AVX2, you can run a benchmark for the currently dispatched version of code and then compare performance with the Intel® Advanced Vector Extensions (Intel® AVX) version of code with Intel® AVX2 disabled. To disable Intel AVX2, call `ippSetCpuFeatures(E9_FM)`. To enable Intel AVX2 back, call `ippSetCpuFeatures( L9_FM )`. Thus, you can use the `ippSetCpuFeatures` function to dispatch any version of Intel IPP code and enable/disable specific CPU features. If you are not well familiar with the features of your CPU, use the `ippInit()` function (or auto-initialization mechanism available starting with Intel IPP 9.0) for the default library behavior.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Return Values

`ippStsNoErr`	Indicates that the required processor-specific code is successfully set.
`ippStsCpuMismatch`	Indicates that the specified processor features are not valid. Previously set code is used. If the requested feature is below the minimal supported by the **px** library - that is Intel® Streaming SIMD Extensions (Intel® SSE) for IA-32 and Intel® SSE2 for Intel® 64 architecture, **px** code is dispatched.
`ippStsFeatureNotSupported`	Indicates that the current CPU does not support at least one of the requested features. If the `ippCPUID_NOCHECK` bit of the *cpuFeatures* parameter is set to 1, these not supported features are enabled, otherwise - disabled.
`ippStsUnknownFeature`	Indicates that at least one of the requested features is unknown. It means that the feature is not defined in the `ippdefs.h` file. Further behavior of the library depends on known features passed to *cpuFeatures*. Unknown features are ignored.
`ippStsFeaturesCombination`	Indicates that the combination of features is not correct. For example, `ippCPUID_AVX2` bit is set to 1 in *cpuFeatures*, but at least one of the `ippCPUID_MMX`, `ippCPUID_SSE`, …, `ippCPUID_AVX` bits is not set. All these missing bits, if supported by CPU, are set to 1. This means that if the library supports the Intel® AVX2 code, it also internally uses all known MMX™, Intel® SSE, and Intel® AVX extensions, which are below Intel® AVX2.

See Also

Init Automatically initializes the library code that is most appropriate for the current processor type.

GetCpuFeatures Retrieves the processor features.

GetEnabledCpuFeatures Returns a features mask for enabled processor features.

SetFlushToZero

Enables or disables flush-to-zero (FTZ) mode.

Syntax

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

Include Files

ippcore.h

Parameters

value	Switch to set or clear the corresponding bit of the MXCSR register. • When *value* is not equal to zero, flush-to-zero (FTZ) mode is enabled • When *value* is set to zero, FTZ mode is disabled
pUMask	Pointer to the current underflow exception mask; may be set to NULL.

Description

This function enables FTZ mode for processors that support Intel® Streaming SIMD Extensions [xx] instructions. The FTZ mode controls the masked response to a SIMD floating-point underflow condition. Use this function to improve performance of applications where underflows are common and rounding the underflow result to zero is acceptable.

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, declare a variable of unsigned integer type in your application and point to it the parameter *pUMask* of the ippSetFlushToZero function. The initial state of mask register is saved in this location and can be restored later. If you do not need to restore the initial mask state, then the pointer *pUMask* may be set to NULL.

Return Values

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

See Also

Bibliography: Casey08

SetDenormAreZeros

Enables or disables denormals-are-zero (DAZ) mode.

Syntax

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

Include Files

ippcore.h

Parameters

value	Switch to set or clear the corresponding bit of the MXCSR register. • When *value* is not equal to zero, denormals-are-zero (DAZ) mode is enabled • When *value* is set to zero, DAZ mode is disabled

Description

This function enables the DAZ mode for processors that support Intel® Streaming SIMD Extensions 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. Use this function to improve processor performance of applications such as streaming media processing, where rounding a denormal operand to zero does not noticeably affect the quality of the processed data.

Return Values

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

See Also

Bibliography: Casey08

AlignPtr

Aligns a pointer to the specified number of bytes.

Syntax

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

Include Files

ippcore.h

Parameters

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

Description

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.

SetNumThreads

Sets the number of threads in the multithreading environment.

Syntax

Case 1: Setting number of threads for operations on objects of 32-bit size:

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

Case 2: Setting number of threads for operations on objects of 64-bit size:

```
IppStatus ippSetNumThreads_L(int numThr);
```

Include Files

`ippcore.h`

`ippcore64x.h` (for 64x flavors)

Parameters

numThr	Number of threads, should be more than zero.

Description

This function sets the number of OpenMP* threads. A number of established threads may be less than specified *numThr*.

Return Values

ippStsNoErr	Indicates no error.
ippStsSizeErr	Indicates an error when *numThr* is less than, or equal to zero.
ippStsNoOperation	Indicates that the function is called from the application linked to the single-threaded version of the library. No operation is performed.

GetNumThreads

Returns the number of existing threads in the multithreading environment.

Syntax

Case 1: Getting number of threads for operations on objects of 32-bit size

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

Case 1: Getting number of threads for operations on objects of 64-bit size

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

Include Files

`ippcore.h`

`ippcore64x.h` (for 64x flavors)

Parameters

pNumThr	Pointer to the number of threads.

Description

This function returns the number of OpenMP* threads specified by the user previously. If it is not specified, the function returns the initial number of threads that depends on the number of logical processors.

Return Values

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

Malloc

Allocates memory aligned to 64-byte boundary.

Syntax

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

Include Files

`ippcore.h`

Parameters

length	Size (in bytes) of the allocated block.

Description

This function allocates a memory block aligned to a 64-byte boundary.

Return Values

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

See Also

Free Frees memory allocated by the function `ippMalloc`.

Free

Frees memory allocated by the function `ippMalloc`.

Syntax

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

Include Files

`ippcore.h`

Parameters

ptr	Pointer to a memory block to be freed.

Description

This function frees an aligned memory block previously allocated by the function `ippMalloc`.

NOTE
Use the `ippFree()` to free memory allocated by `ippMalloc()`. Use `free` to free memory allocated by `malloc` or `calloc`.

Dispatcher Control Functions

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

Init

Automatically initializes the library code that is most appropriate for the current processor type.

Syntax

```
IppStatus ippInit(void));
```

Include Files

`ippcore.h`

Description

This function detects the processor type used in the user computer system and sets the processor-specific code of the Intel IPP library most appropriate for the current processor type.

NOTE
You can not use any other Intel IPP function while the function `ippInit` continues execution.

Return Values

`ippStsNoErr`	Indicates that the required processor-specific code is successfully set.
`ippStsNotSupportedCpu`	Indicates that the CPU is not supported.
`ippStsNonIntelCpu`	Indicates that the target CPU is not Genuine Intel.

Vector Initialization Functions

4

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

Vector Initialization Functions

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

Copy

Copies the contents of one vector into another.

Syntax

```
IppStatus ippsCopy_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsCopy_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsCopy_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsCopy_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCopy_64s(const Ipp64s* pSrc, Ipp64s* pDst, int len);
IppStatus ippsCopy_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCopy_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsCopy_32sc(const Ipp32sc* pSrc, Ipp32sc* pDst, int len);
IppStatus ippsCopy_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCopy_64sc(const Ipp64sc* pSrc, Ipp64sc* pDst, int len);
IppStatus ippsCopy_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

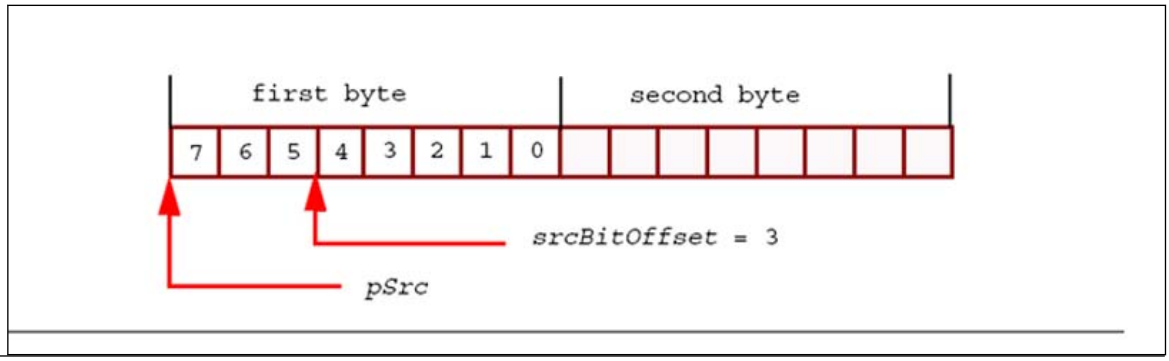
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements to copy.

Description

This function copies the first *len* elements from a source vector *pSrc* into a destination vector *pDst*.

ippsCopy_1u. This function flavor copies elements of a vector that has a `8u` data type. It means that each byte consists of eight consecutive elements of the vector (1 bit per element). You need to specify the start position of the source and destination vectors in the *srcBitOffset* and *dstBitOffset* parameters, respectively. The bit order of each byte is inverse to the element order. It means that the first element in a vector represents the last (seventh) bit of the first byte in a vector, as shown in the figure below.

Bit Layout for the Function ippsCopy_1u.



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, use `ippsMove`.

Return Values

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

Example

The example below shows how to use the `ippsCopy` function.

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

See Also

Move Moves the contents of one vector to another vector.

CopyLE, CopyBE

Copies the contents of one bit vector into another.

Syntax

```
IppStatus ippsCopyLE_1u(const Ipp8u* pSrc, int srcBitOffset, Ipp8u* pDst, int
dstBitOffset, int len);
```

```
IppStatus ippsCopyBE_1u(const Ipp8u* pSrc, int srcBitOffset, Ipp8u* pDst, int
dstBitOffset, int len);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

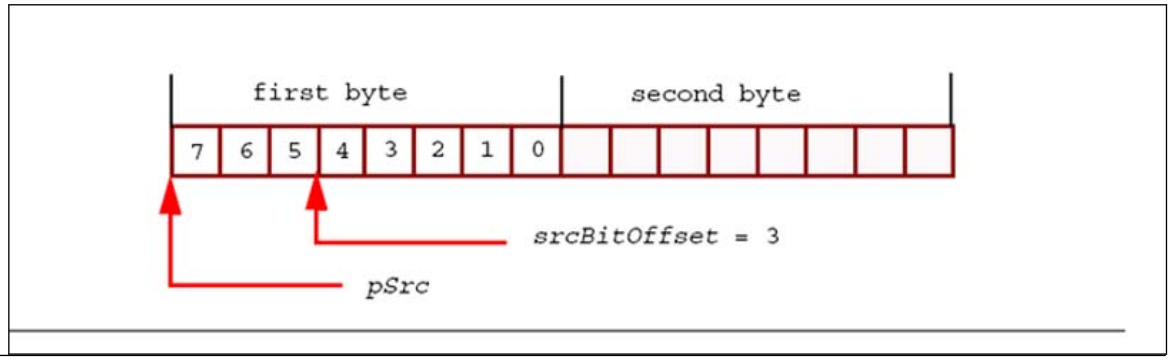
Description

This function copies the first *len* elements from a source vector *pSrc* into a destination vector *pDst*.

These functions copy elements of a vector that has a 8u data type. It means that each byte consists of eight consecutive elements of the vector (1 bit per element). You need to specify the start position of the source and destination vectors in the *srcBitOffset* and *dstBitOffset* parameters, respectively.

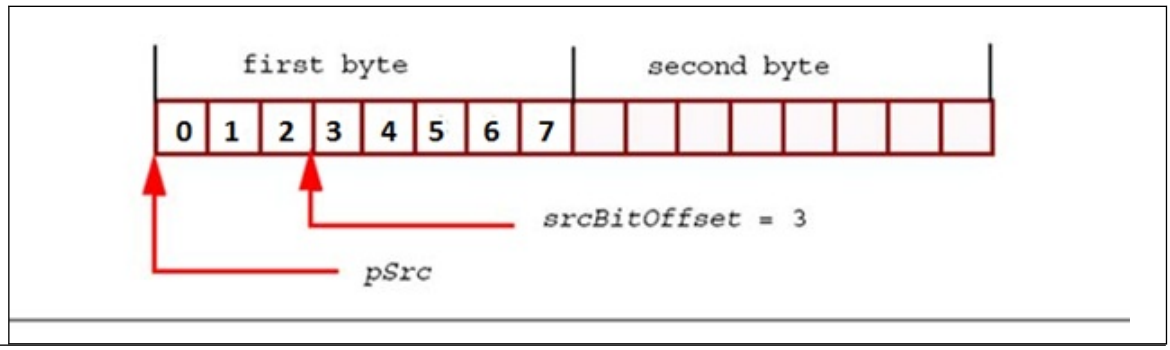
For the ippsCopyLE_1u function, the bit order of each byte is inverse to the element order. It means that the first element in a vector represents the last (seventh) bit of the first byte in a vector, as shown in the figure below.

Bit Layout for the ippsCopyLE_1u Function



For the ippsCopyBE_1u function, the bit order of each byte is ordinary. It means that the first element in a vector represents the last (zero) bit of the first byte in a vector, as shown in the figure below.

Bit Layout for the ippsCopyBE_1u Function



Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when: • *len* is less than, or equal to zero • *srcBitOffset* or *dstBitOffset* is less than zero

Move

Moves the contents of one vector to another vector.

Syntax

```
IppStatus ippsMove_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsMove_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMove_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsMove_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMove_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsMove_64s(const Ipp64s* pSrc, Ipp64s* pDst, int len);
IppStatus ippsMove_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsMove_32sc(const Ipp32sc* pSrc, Ipp32sc* pDst, int len);
IppStatus ippsMove_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsMove_64sc(const Ipp64sc* pSrc, Ipp64sc* pDst, int len);
IppStatus ippsMove_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

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 Values

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

Example

The example below shows how to use the function ippsMove.

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

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

Result:

```
pSrc = 123456789
pDst = 123456
```

Set

Initializes vector elements to a specified common value.

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function initializes the first *len* elements of the real or complex vector *pDst* to contain the same value *val*.

Return Values

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

Example

The code example below shows how to use the function ippsSet.

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

Zero

Initializes a vector to zero.

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function initializes the first *len* elements of the vector *pDst* to zero. If *pDst* is a complex vector, both real and imaginary parts are zeroed.

Return Values

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

Example

The code example below shows how to use the ippsZero function.

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

Sample-Generating Functions

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

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

Tone-Generating Functions

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

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

Tone

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

Syntax

IppStatus ippsTone_16s(Ipp16s* *pDst*, int *len*, Ipp16s *magn*, Ipp32f *rFreq*, Ipp32f* *pPhase*, IppHintAlgorithm *hint*);

IppStatus ippsTone_16sc(Ipp16sc* *pDst*, int *len*, Ipp16s *magn*, Ipp32f *rFreq*, Ipp32f* *pPhase*, IppHintAlgorithm *hint*);

```
IppStatus ippsTone_32f(Ipp32f* pDst, int len, Ipp32f magn, Ipp32f rFreq, Ipp32f*
pPhase, IppHintAlgorithm hint);
```

```
IppStatus ippsTone_32fc(Ipp32fc* pDst, int len, Ipp32f magn, float rFreq, Ipp32f*
pPhase, IppHintAlgorithm hint);
```

```
IppStatus ippsTone_64f(Ipp64f* pDst, int len, Ipp64f magn, Ipp64f rFreq, Ipp64f*
pPhase, IppHintAlgorithm hint);
```

```
IppStatus ippsTone_64fc(Ipp64fc* pDst, int len, Ipp64f magn, Ipp64f rFreq, Ipp64f*
pPhase, IppHintAlgorithm hint);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

magn	Magnitude of the tone, that is, the maximum value attained by the wave.
pPhase	Pointer to the phase of the tone relative to a cosine wave. It must be in range [0.0, 2π). You can use the returned value 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 that stores the samples.
len	Number of samples to be computed.
hint	Suggests using specific code. The possible values for the *hint* argument are described in Hint Arguments.

Description

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:

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

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

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

The parameter *hint* suggests using specific code, which provides for either fast but less accurate calculation, or more accurate but slower execution.

Return Values

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

ippStsToneFreqErr	Indicates an error when *rFreq* is negative, or greater than, or equal to 0.5 for real tone and to 1.0 for complex tone.
ippStsTonePhaseErr	Indicates an error when the *pPhase* value is negative, or greater than or equal to IPP_2PI.

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.

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:

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

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

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

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

$H = \pi + h$

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

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

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

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:

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

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

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

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

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

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

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

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

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

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

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

Triangle

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

Syntax

```
IppStatus ippsTriangle_16s(Ipp16s* pDst, int len, Ipp16s magn, Ipp32f rFreq, Ipp32f asym, Ipp32f* pPhase);
IppStatus ippsTriangle_16sc(Ipp16sc* pDst, int len, Ipp16s magn, Ipp32f rFreq, Ipp32f asym, Ipp32f* pPhase);
IppStatus ippsTriangle_32f(Ipp32f* pDst, int len, Ipp32f magn, Ipp32f rFreq, Ipp32f asym, Ipp32f* pPhase);
IppStatus ippsTriangle_32fc(Ipp32fc* pDst, int len, Ipp32f magn, Ipp32f rFreq, Ipp32f asym, Ipp32f* pPhase);
IppStatus ippsTriangle_64f(Ipp64f* pDst, int len, Ipp64f magn, Ipp64f rFreq, Ipp64f asym, Ipp64f* pPhase);
IppStatus ippsTriangle_64fc(Ipp64fc* pDst, int len, Ipp64f magn, Ipp64f rFreq, Ipp64f asym, Ipp64f* pPhase);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

rFreq	Frequency of the triangle relative to the sampling frequency. It must be in range [0.0, 0.5).
pPhase	Pointer to the phase of the triangle relative to a cosine triangular analog wave. It must be in range [0.0, 2π). You can use the returned value 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 that stores the samples.
len	Number of samples to be computed.

Description

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:

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

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

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

See Triangle-Generating Functions for the definition of functions **ct** h and **st** h.

Return Values

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

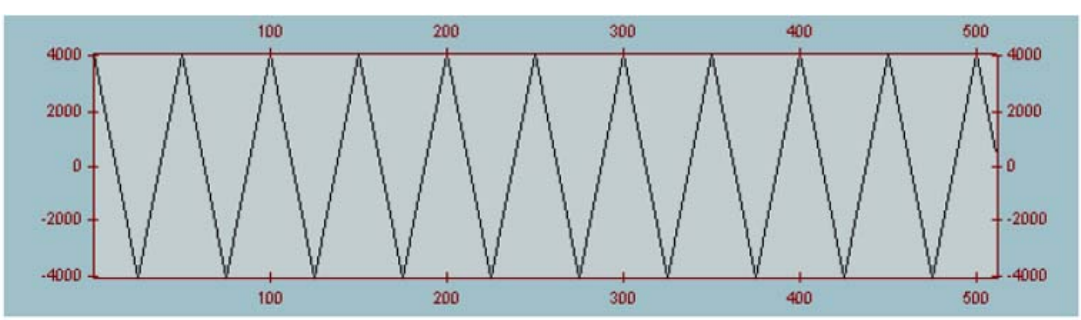
Example

The code example below demonstrates how to use the ippsTriangle function.

```
void func_triangle_direct()
{
    Ipp16s* pDst;
    int len = 512;
    Ipp16s magn = 4095;
    Ipp32f rFreq = 0.02;
    Ipp32f asym = 0.0;
    Ipp32f Phase = 0.0;
    IppStatus status;

    status = ippsTriangle_16s(pDst, len, magn, rFreq, asym, &Phase);
    if(ippStsNoErr != status)
      printf("IPP Error: %s",ippGetStatusString(status));
}
```

Result:



Uniform Distribution Functions

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

RandUniformInit

Initializes a noise generator with uniform distribution.

Syntax

```
IppStatus ippsRandUniformInit_8u(IppsRandUniState_8u* pRandUniState, Ipp8u low, Ipp8u high, unsigned int seed);
IppStatus ippsRandUniformInit_16s(IppsRandUniState_16s* pRandUniState, Ipp16s low, Ipp16s high, unsigned int seed);
IppStatus ippsRandUniformInit_32f(IppsRandUniState_32f* pRandUniState, Ipp32f low, Ipp32f high, unsigned int seed);
IppStatus ippsRandUniformInit_64f(IppsRandUniState_64f* pRandUniState, Ipp64f low, Ipp64f high, unsigned int seed);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function initializes the pseudo-random generator state structure *pRandUniState* in the external buffer. The uniform distribution range is specified by the lower and upper bounds *low* and *high*, respectively. Before using this function, you need to compute the size of the external buffer by using the `ippsRandUniformGetSize` function.

Return Values

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

See Also

RandUniformGetSize Computes the length of the uniform distribution generator structure.

RandUniformGetSize

Computes the length of the uniform distribution generator structure.

Syntax

```
IppStatus ippsRandUniformGetSize_8u(int* pRandUniformStateSize);
IppStatus ippsRandUniformGetSize_16s(int* pRandUniformStateSize);
IppStatus ippsRandUniformGetSize_32f(int* pRandUniformStateSize);
IppStatus ippsRandUniformGetSize_64f(int* pRandUniformStateSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function computes the length (in bytes) *pRandUniformStateSize* of the uniform distribution generator structure that is used by the `ippsRandUniformInit` function.

Return Values

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

See Also

RandUniformInit Initializes a noise generator with uniform distribution.

RandUniform

Generates the pseudo-random samples with a uniform distribution.

Syntax

```
IppStatus ippsRandUniform_8u(Ipp8u* pDst, int len, IppsRandUniState_8u* pRandUniState);
IppStatus ippsRandUniform_16s(Ipp16s* pDst, int len, IppsRandUniState_16s*
pRandUniState);
IppStatus ippsRandUniform_32f(Ipp32f* pDst, int len, IppsRandUniState_32f*
pRandUniState);
IppStatus ippsRandUniform_64f(Ipp64f* pDst, int len, IppsRandUniState_64f*
pRandUniState);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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 function RandUniformInit.

Return Values

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

Gaussian Distribution Functions

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

RandGaussInit

Initializes a noise generator with Gaussian distribution.

Syntax

```
IppStatus ippsRandGaussInit_8u(IppsRandGaussState_8u* pRandGaussState, Ipp8u mean,
Ipp8u stdDev, unsigned int seed);
```

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

```
IppStatus ippsRandGaussInit_32f(IppsRandGaussState_32f* pRandGaussState, Ipp32f mean,
Ipp32f stdDev, unsigned int seed);
```

```
IppStatus ippsRandGaussInit_64f(IppsRandGaussState_64f* pRandGaussState, Ipp64f mean,
Ipp64f stdDev, unsigned int seed);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

mean	Mean of the Gaussian distribution.
stdDev	Standard deviation of the Gaussian distribution.
seed	Seed value used by the pseudo-random number generator algorithm.

Description

This function initializes the pseudo-random generator state structure *pRandGaussState* in the external buffer. This structure contains parameters of the required noise generator that are specified by the *mean*, *stdDev*, and *seed* values. Before using this function, you need to compute the size of the buffer by calling the `ippsRandGaussGetSize` function.

Return Values

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

See Also

RandGaussGetSize Computes the length of the Gaussian distribution generator structure.

RandGaussGetSize

Computes the length of the Gaussian distribution generator structure.

Syntax

```
IppStatus ippsRandGaussGetSize_8u(int* pRandGaussStateSize);
IppStatus ippsRandGaussGetSize_32f(int* pRandGaussStateSize);
IppStatus ippsRandGaussGetSize_16s(int* pRandGaussStateSize);
IppStatus ippsRandGaussGetSize_64f(int* pRandGaussStateSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function computes the length (in bytes) *pRandGaussStateSize* of the uniform distribution generator structure that is used by the `ippsRandGaussInit` function.

Return Values

`ippStsNoErr`	Indicates no error.

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

See Also

RandGaussInit Initializes a noise generator with Gaussian distribution.

RandGauss

Generates the pseudo-random samples with a Gaussian distribution.

Syntax

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

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

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

```
IppStatus ippsRandGauss_64f(Ipp64f* pDst, int len, IppsRandGaussState_64f*
pRandGaussState);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

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 RandGaussInit function.

Return Values

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

Special Vector Functions

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

VectorJaehne

Creates a Jaehne vector.

Syntax

```
IppStatus ippsVectorJaehne_8u(Ipp8u* pDst, int len, Ipp8u magn);
IppStatus ippsVectorJaehne_16u(Ipp16u* pDst, int len, Ipp16u magn);
IppStatus ippsVectorJaehne_16s(Ipp16s* pDst, int len, Ipp16s 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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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:

$pDst[n] = magn * \sin((0.5\pi n^2)/len), 0 \le n < len$

Return Values

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

Example

The code example below shows how to use the function `ippsVectorJaehne`.

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

VectorSlope

Creates a slope vector.

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function creates a slope vector and stores the result in *pDst*. The destination vector elements are computed according to the following formula:

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

Return Values

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

Essential Functions

5

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

Logical and Shift Functions

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

For binary logical operations AND, OR and XOR, the following functions are provided:

`AndC`, `OrC`, `XorC` for vector-scalar operations;

`And`, `Or`, `Xor` for vector-vector operations.

AndC

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

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function computes the bitwise AND of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

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

Return Values

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

And

Computes the bitwise AND of two vectors.

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function computes the bitwise AND of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

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

Return Values

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

OrC

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

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function computes the bitwise OR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

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

Return Values

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

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

Or

Computes the bitwise OR of two vectors.

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function computes the bitwise OR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

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

Return Values

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

XorC

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

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function computes the bitwise XOR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

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

Return Values

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

Xor

Computes the bitwise XOR of two vectors.

Syntax

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

```
IppStatus ippsXor_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsXor_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function computes the bitwise XOR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

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

Return Values

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

Not

Computes the bitwise NOT of the vector elements.

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function computes the bitwise NOT of the corresponding elements of the vectors *pSrc*, and stores the result in the vector *pDst*.

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

Return Values

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

LShiftC

Shifts bits in vector elements to the left.

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function shifts each element of the vector *pSrc* by *val* bits to the left, and stores the result in *pDst*.

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

Return Values

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

RShiftC

Shifts bits in vector elements to the right.

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

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

Description

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.

Return Values

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

Example

The code example below 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].

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

Output:

```
    saturate =  255 0 127 4 5 0 7 8
```

Arithmetic Functions

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

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

AddC

Adds a constant value to each element of a vector.

Syntax

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

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

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

```
IppStatus ippsAddC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int
scaleFactor);
IppStatus ippsAddC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len, int
scaleFactor);
IppStatus ippsAddC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len, int
scaleFactor);
IppStatus ippsAddC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len, int
scaleFactor);
IppStatus ippsAddC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int len,
int scaleFactor);
IppStatus ippsAddC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst, int len,
int scaleFactor);
IppStatus ippsAddC_64u_Sfs(const Ipp64u* pSrc, Ipp64u val, Ipp64u* pDst, Ipp32u len,
int scaleFactor, IppRoundMode rndMode);
IppStatus ippsAddC_64s_Sfs(const Ipp64s* pSrc, Ipp64s val, Ipp64s* pDst, Ipp32u len,
int scaleFactor, IppRoundMode rndMode);
```

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

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

Case 4: In-place operations on integer data.

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
val	Scalar value used to increment each element of the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
scaleFactor	Scale factor, refer to Integer Scaling.
rndMode	Rounding mode, the following values are possible: `ippRndZero` floating-point values are truncated to zero `ippRndNear` floating-point values are rounded to the nearest even integer when the fractional part equals 0.5; otherwise they are rounded to the nearest integer `ippRndFinancial` floating-point values are rounded down to the nearest integer when the fractional part is less than 0.5, or rounded up to the nearest integer if the fractional part is equal or greater than 0.5.

Description

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` suffixe perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result is saturated.

Return Values

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

See Also

Integer Scaling

Add

Adds the elements of two vectors.

Syntax

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

```
IppStatus ippsAdd_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int len);
IppStatus ippsAdd_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_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_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f* pDst, int
len);
```

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

```
IppStatus ippsAdd_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len,
int scaleFactor);
IppStatus ippsAdd_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int
len, int scaleFactor);
IppStatus ippsAdd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int
len, int scaleFactor);
IppStatus ippsAdd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s* pDst, int
len, int scaleFactor);
IppStatus ippsAdd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc* pDst,
int len, int scaleFactor);
IppStatus ippsAdd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, Ipp32sc* pDst,
int len, int scaleFactor);
IppStatus ippsAdd_64s_Sfs(const Ipp64s* pSrc1, const Ipp64s* pSrc2, Ipp64s* pDst, int
len, int scaleFactor);
```

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

```
IppStatus ippsAdd_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsAdd_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsAdd_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsAdd_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsAdd_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsAdd_16s32s_I(const Ipp16s* pSrc, Ipp32s* pSrcDst, int len);
```

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

```
IppStatus ippsAdd_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsAdd_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAdd_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAdd_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAdd_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAdd_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len, int
scaleFactor);
```

Case 5. Not in-place operations on floating point data of 64-bit size.

```
IppStatus ippsAdd_32f_L(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, Ipp64u
len);
```

Include Files

`ipps.h`

`ipps64x.h` (for 64x flavors)

Domain Dependencies

Flavors declared in `ipps.h`:

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Flavors declared in `ipps64x.h`:

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`, `ippcore_tl.lib`, `ipps_tl.lib`

Parameters

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

Description

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

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

Add.c

Add_I.c

See Also

Integer Scaling

AddProductC

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

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source vector.
val	The value by which the source vector is multiplied.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.

Description

This function multiplies each element of the source vector *pSrc* by a value *val* and adds the result to the corresponding element of the accumulator vector *pSrcDst* as given by:

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

Return Values

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

AddProduct

Adds product of two vectors to the accumulator vector.

Syntax

Case 1. Operations on floating point data.

```
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);
```

Case 2. Operations on integer data with scaling.

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

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

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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], 0 ≤ n < *len*.

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

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when any of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

MulC

Multiplies each element of a vector by a constant value.

Syntax

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

```
IppStatus ippsMulC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsMulC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsMulC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsMulC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
IppStatus ippsMulC_Low_32f16s(const Ipp32f* pSrc, Ipp32f val, Ipp16s* pDst, int len);
```

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

```
IppStatus ippsMulC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int
scaleFactor);
IppStatus ippsMulC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len, int
scaleFactor);
IppStatus ippsMulC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len, int
scaleFactor);
IppStatus ippsMulC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len, int
scaleFactor);
IppStatus ippsMulC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int len,
int scaleFactor);
IppStatus ippsMulC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst, int len,
int scaleFactor);
IppStatus ippsMulC_32f16s_Sfs(const Ipp32f* pSrc, Ipp32f val, Ipp16s* pDst, int len,
int scaleFactor);
```

Case 3. In-place operations without scaling.

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

Case 4. In-place operations with scaling.

```
IppStatus ippsMulC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsMulC_64f64s_ISfs(Ipp64f val, Ipp64s* pSrcDst, Ipp32u 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_64s_ISfs(Ipp64s val, Ipp64s* pSrcDst, Ipp32u len, int scaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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*.

The 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.

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

Return Values

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

See Also

Integer Scaling

Mul

Multiplies the elements of two vectors.

Syntax

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

```
IppStatus ippsMul_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int len);
IppStatus ippsMul_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
```

```
IppStatus ippsMul_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, int len);
IppStatus ippsMul_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst, int
len);
IppStatus ippsMul_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst, int
len);
IppStatus ippsMul_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp16u* pDst, int len);
IppStatus ippsMul_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst, int
len);
IppStatus ippsMul_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f* pDst, int
len);
```

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

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

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

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

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

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

```
IppStatus ippsMul_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsMul_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len, int
scaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2* and stores the result in *pDst*.

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

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

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

Return Values

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

See Also

Integer Scaling

SubC

Subtracts a constant value from each element of a vector.

Syntax

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

```
IppStatus ippsSubC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
```

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

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

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

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

```
IppStatus ippsSubC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsSubC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

Case 4. In-place operations on integer data.

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
val	Scalar value used to decrement each element of the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operation.

len	Number of elements in the vector
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function subtracts a value *val* from each element of the vector *pSrc*, and stores the result in *pDst*.

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

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

Return Values

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

SubCRev

Subtracts each element of a vector from a constant value.

Syntax

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

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

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

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

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

```
IppStatus ippsSubCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubCRev_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubCRev_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
```

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

Case 4. In-place operations on integer data.

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function subtracts each element of the vector *pSrc* from a value *val* and stores the result in *pDst*.

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

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

Return Values

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

Sub

Subtracts the elements of two vectors.

Syntax

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

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

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

```
IppStatus ippsSub_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len,
int scaleFactor);
IppStatus ippsSub_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int
len, int scaleFactor);
IppStatus ippsSub_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int
len, int scaleFactor);
IppStatus ippsSub_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s* pDst, int
len, int scaleFactor);
IppStatus ippsSub_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, Ipp16sc* pDst,
int len, int scaleFactor);
IppStatus ippsSub_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2, Ipp32sc* pDst,
int len, int scaleFactor);
```

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

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

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

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function subtracts the elements of the vector *pSrc1* from the elements of the vector *pSrc2*, and stores the result in *pDst*.

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

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

Return Values

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

DivC

Divides each element of a vector by a constant value.

Syntax

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

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

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

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

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

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

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

```
IppStatus ippsDivC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsDivC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsDivC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsDivC_64s_ISfs(Ipp64s val, Ipp64s* pSrcDst, Ipp32u len, int scaleFactor);
IppStatus ippsDivC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int scaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function divides each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

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

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

Return Values

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

DivCRev

Divides a constant value by each element of a vector.

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function divides the constant value *val* by each element of the vector *pSrc* and stores the results in *pDst*.

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

Return Values

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

`ippStsDivByZeroErr`	Indicates an error when any element of the vector *pSource* is equal to 0.

Div

Divides the elements of two vectors.

Syntax

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

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

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

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

Case 3. In-place operations on integer data.

```
IppStatus ippsDiv_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsDiv_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
scaleFactor);
IppStatus ippsDiv_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
scaleFactor);
IppStatus ippsDiv_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len, int
scaleFactor);
IppStatus ippsDiv_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
scaleFactor);
```

Case 4. In-place operations on floating point data.

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc1	Pointer to the divisor vector.
pSrc2	Pointer to the dividend vector.
pDst	Pointer to the destination vector.
pSrc	Pointer to the divisor vector for in-place operations.
pSrcDst	Pointer to the source and destination vector for in-place operations.
len	Number of elements in the vector.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function divides the elements of the *pSrc2* vector by the elements of the *pSrc1* vector , 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` suffixe perform scaling of the result in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result is saturated.

If any of the divisor vector elements is equal to zero, the function returns a warning and continues execution with the corresponding result value. For more information see "Handling of Special Cases".

Return Values

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

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`Div.c`

`Div_I.c`

See Also

Integer Scaling

Handling of Special Cases

Div_Round

Divides the elements of two vectors with rounding.

Syntax

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

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

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

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

Case 2. In-place operations on integer data.

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

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

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc1	Pointer to the vector whose elements are used as divisors.
pSrc2	Pointer to the vector whose elements are used as dividends.
pDst	Pointer to the destination vector.
pSrc	Pointer to the source vector whose elements are used as divisors for in-place operations.
pSrcDst	Pointer to the source and destination vector for in-place operations.
len	Number of elements in the vector.
rndMode	Rounding mode, the following values are possible: `ippRndZero` - specifies that floating-point values are truncated toward zero, `ippRndNear` - specifies that floating-point values are rounded to the nearest even integer when the fractional part equals 0.5; otherwise they are rounded to the nearest integer,

	`ippRndFinancial` - specifies that floating-point values are rounded down to the nearest integer when the fractional part is less than 0.5, or rounded up to the nearest integer if the fractional part is equal or greater than 0.5.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

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

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

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

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

Return Values

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

Abs

Computes absolute values of vector elements.

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

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 ippsMagnitudeippsMagnitude.

Return Values

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

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

Abs.c

Abs_I.c

Sqr

Computes a square of each element of a vector.

Syntax

```
IppStatus ippsSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqr_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqr_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqr_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_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_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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
len	Number of elements in the vector
scaleFactor	Scale factor, refer to Integer Scaling.

Description

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.

Return Values

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

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`Sqr.c`

`Sqr_I.c`

Sqrt

Computes a square root of each element of a vector.

Syntax

```
IppStatus ippsSqrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqrt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqrt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqrt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqrt_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_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int
scaleFactor);
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_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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
len	Number of elements in the vector
scaleFactor	Scale factor, refer to Integer Scaling.

Description

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:

$pDst[n] = (pSrc[n])^{1/2}$

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:

$pSrcDst[n] = (pSrcDst[n])^{1/2}$.

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

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

If the function `ippsSqrt` encounters a negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

To increase precision of an integer output, use the scale factor.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, *or pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsSqrtNegArg`	Indicates a warning that a source element has a negative value.

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`Sqrt.c`

`Sqrt_I.c`

Cubrt

Computes cube root of each element of a vector.

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vector
scaleFactor	Scale factor, refer to Integer Scaling.

Description

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:

$pDst[n] = (pSrc[n])^{1/3}$, $0 \le n < len$.

Return Values

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

Exp

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

Syntax

```
IppStatus ippsExp_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_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_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

pSrcDst	Pointer to the source and destination vector *pSrcDst* for the in-place operation.
len	Number of elements in the vector
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function computes the exponential function of each element of the vector *pSrc*, and stores the result in *pDst*.

The computation is performed as follows:

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

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

The computation is performed as follows:

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

When an overflow occurs, the function continues operation with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

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

Application Notes

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

Return Values

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

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

Exp.c

Exp_I.c

Ln

Computes the natural logarithm of each element of a vector.

Syntax

```
IppStatus ippsLn_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_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_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsLn_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsLn_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsLn_64f_I(Ipp64f* pSrcDst, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function computes the natural logarithm of each element of the vector *pSrc* and stores the result in *pDst* as given by

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

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

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

If the function `ippsLn` encounters a zero or negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A "Handling of Special Cases" for more information).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero.
`ippStsLnZeroArg`	Indicates a warning for zero-valued input vector elements.
`ippStsLnNegArg`	Indicates a warning for negative input vector elements.

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`Ln.c`

Ln_I.c

SumLn

Sums natural logarithms of each element of a vector.

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

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:

Return Values

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

Arctan

Computes the inverse tangent of each element of a vector.

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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 ≤ n < *len*.

Return Values

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

Normalize

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

Syntax

Case 1: Not-in-place operations on floating point and integer data

```
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);
```

Case 2: In-place operations on floating point and integer data

```
IppStatus ippsNormalize_32f_I(const Ipp32f* pSrcDst, int len, Ipp32f vSub, Ipp32f
vDiv);
```

```
IppStatus ippsNormalize_64f_I(const Ipp64f* pSrcDst, int len, Ipp64f vSub, Ipp64f
vDiv);
```

```
IppStatus ippsNormalize_32fc_I(const Ipp32fc* pSrcDst, int len, Ipp32fc vSub, Ipp32f
vDiv);
```

```
IppStatus ippsNormalize_64fc_I(const Ipp64fc* pSrcDst, int len, Ipp64fc vSub, Ipp64f
vDiv);
```

```
IppStatus ippsNormalize_16s_ISfs(const Ipp16s* pSrcDst, int len, Ipp16s vSub, int vDiv,
int scaleFactor);
```

```
IppStatus ippsNormalize_16sc_ISfs(const Ipp16sc* pSrcDst, int len, Ipp16sc vSub, int
vDiv, int scaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
vSub	Subtrahend value.
vDiv	Denominator value.
pDst	Pointer to the vector which stores the normalized elements.
len	Number of elements in the vector
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function subtracts *vSub* from elements of the input vector *pSrc* (*pSrcDst* for in-place operations), divides the differences by *vDiv*, and stores the result in *pDst* (*pSrcDst* for in-place operations). The computation is performed as follows:

pDst[n] = (*pSrc*[n] - *vSub*)/*vDiv*.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsDivByZeroErr	Indicates an error when *vDiv* is equal to 0 or less than the minimum floating-point positive number.

Conversion Functions

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

- Sorting all elements of a vector
- Data type conversion (including floating-point to integer and integer to floating-point)
- Joining several vectors
- Extracting components from a complex vector and constructing a complex vector
- Computing the complex conjugates of vectors
- Cartesian to polar and polar to Cartesian coordinate conversion.

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

SortAscend, SortDescend

Sorts all elements of a vector.

Syntax

```
IppStatus ippsSortAscend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortAscend_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSortAscend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortAscend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortAscend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortAscend_64f_I(Ipp64f* pSrcDst, int len);

IppStatus ippsSortDescend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortDescend_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSortDescend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortDescend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortDescend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortDescend_64f_I(Ipp64f* pSrcDst, int len);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

These functions rearrange all elements of the source vector *pSrcDst* in the ascending or descending order, respectively, and store the result in the destination vector *pSrcDst*.

Return Values

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

Example

To better understand how to use these functions, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`SortAscend.c`

`SortDescend.c`

SortIndexAscend, SortIndexDescend

Rearranges elements of the vector and their indexes.

Syntax

```
IppStatus ippsSortIndexAscend_8u_I(Ipp8u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_16u_I(Ipp16u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_16s_I(Ipp16s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_32s_I(Ipp32s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_32f_I(Ipp32f* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_64f_I(Ipp64f* pSrcDst, int* pDstIdx, int len);

IppStatus ippsSortIndexDescend_8u_I(Ipp8u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_16u_I(Ipp16u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_16s_I(Ipp16s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_32s_I(Ipp32s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_32f_I(Ipp32f* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_64f_I(Ipp64f* pSrcDst, int* pDstIdx, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

These functions rearrange all elements of the source vector *pSrcDst* in the ascending or descending order, respectively, and store the elements in the destination vector *pSrcDst*, and their indexes in the desalination vector *pDstIdx*. If some elements are identical, their indexes are not ordered.

Return Values

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

SortRadixGetBufferSize

Computes the size of the buffer for the SortRadixAscend *and* SortRadixDescend *functions.*

Syntax

```
IppStatus ippsSortRadixGetBufferSize(int len, IppDataType dataType, int* pBufSize);
```

```
IppStatus ippsSortRadixGetBufferSize_L(IppSizeL len, IppDataType dataType, IppSizeL*
pBufSize);
```

Include Files

ipps.h

Flavors with the _L suffix: ipps_l.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

len	Number of elements in the vector
dataType	Data type of the vector.
pBufSize	Pointer to the buffer size.

Description

This function calculates the size of the buffer for the ippsSortRadixAscend/ippsSortRadixDescend functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pBufSize* is NULL.
ippStsSizeErr	Indicates an error when *len* is less than, or equal to 0.
ippStsDataTypeErr	Indicates an error when the *dataType* value is not supported.

See Also

SortRadixAscend, SortRadixDescend Sorts all elements of a vector using radix sorting algorithm.

SortRadixAscend, SortRadixDescend

Sorts all elements of a vector using radix sorting algorithm.

Syntax

```
IppStatus ippsSortRadixAscend_8u_I(Ipp8u* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_16u_I(Ipp16u* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_16s_I(Ipp16s* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_32u_I(Ipp32u* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_32s_I(Ipp32s* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_32f_I(Ipp32f* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_64f_I(Ipp64f* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_32s_I_L(Ipp32s* pSrcDst, IppSizeL len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_64s_I_L(Ipp64s* pSrcDst, IppSizeL len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_32f_I_L(Ipp32f* pSrcDst, IppSizeL len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_64f_I_L(Ipp64f* pSrcDst, IppSizeL len, Ipp8u* pBuffer);
IppStatus ippsSortRadixAscend_64s_I_L(Ipp32s* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_8u_I(Ipp8u* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_16u_I(Ipp16u* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_16s_I(Ipp16s* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_32u_I(Ipp32u* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_32s_I(Ipp32s* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_32f_I(Ipp32f* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_64f_I(Ipp64f* pSrcDst, Ipp32s len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_32s_I_L(Ipp32s* pSrcDst, IppSizeL len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_64s_I_L(Ipp64s* pSrcDst, IppSizeL len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_32f_I_L(Ipp32f* pSrcDst, IppSizeL len, Ipp8u* pBuffer);
IppStatus ippsSortRadixDescend_64f_I_L(Ipp64f* pSrcDst, IppSizeL len, Ipp8u* pBuffer);
```

Include Files

ipps.h

Flavors with the _L suffix: ipps_l.h

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrcDst	Pointer to the source and destination vector.
len	Number of elements in the vector
pBuffer	Pointer to the buffer for internal calculations. To compute the required buffer size, use the SortRadixGetBufferSize function.

Description

These functions rearrange all elements of the source vector *pSrcDst* in the ascending or descending order, respectively, using "radix sort" algorithm, and store the result in the destination vector *pSrcDst*.

Flavors with the `_L` suffix operate on larger data size.

These functions require the work buffer for internal calculations, to compute the size of the buffer, use the SortRadixGetBufferSize or SortRadixGetBufferSize_L (for the flavors with the `_L` suffix) function.

Return Values

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

See Also

SortRadixGetBufferSize Computes the size of the buffer for the `SortRadixAscend` and `SortRadixDescend` functions.

SortRadixIndexGetBufferSize

Computes the size of the buffer for the `SortRadixIndexAscend` *and* `SortRadixIndexDescend` *functions.*

Syntax

```
IppStatus ippsSortRadixIndexGetBufferSize(int len, IppDataType dataType, int*
pBufSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

len	Number of elements in the vector
dataType	Data type pf the vector.

pBufSize	Pointer to the buffer size.

Description

This function calculates the size of the buffer for the ippsSortRadixIndexAscend/ ippsSortRadixIndexDescend functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pBufSize* is NULL.
ippStsSizeErr	Indicates an error when *len* is less than, or equal to 0.
ippStsDataTypeErr	Indicates an error when the *dataType* value is not supported.

See Also

SortRadixIndexAscend, SortRadixIndexDescend Indirectly sorts all elements of a vector using radix sorting algorithm.

SortRadixIndexAscend, SortRadixIndexDescend

Indirectly sorts all elements of a vector using radix sorting algorithm.

Syntax

```
IppStatus ippsSortRadixIndexAscend_8u(const Ipp32f* const pSrc, Ipp32s srcStrideBytes,
Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexAscend_16u(const Ipp16u* const pSrc, Ipp32s srcStrideBytes,
Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexAscend_16s(const Ipp16s* const pSrc, Ipp32s srcStrideBytes,
Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexAscend_32s(const Ipp32s* const pSrc, Ipp32s srcStrideBytes,
Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexAscend_32u(const Ipp32u* const pSrc, Ipp32s srcStrideBytes,
Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexAscend_32f(const Ipp32f* const pSrc, Ipp32s srcStrideBytes,
Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexDescend_8u(const Ipp8u* const pSrc, Ipp32s srcStrideBytes,
Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexDescend_16u(const Ipp16u* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexDescend_16s(const Ipp16s* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexDescend_32s(const Ipp32s* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexDescend_32u(const Ipp32u* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

```
IppStatus ippsSortRadixIndexDescend_32f(const Ipp32f* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s len, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer the source sparse keys vector.
srcStrideBytes	Distance in bytes between two consecutive elements of the source vector.
pDstIndx	Pointer to the destination vector of indexes.
len	Number of elements in the vectors.
pBuffer	Pointer to the work buffer for internal calculations. To compute the size of the buffer, use the SortRadixIndexGetBufferSize function.

Description

These functions indirectly sort all elements of the source sparse keys vector *pSrc* in the ascending or descending order, respectively, using "radix sort" algorithm and store the indexes of resulting arrangement order in the destination vector *pDstIndx*. Elements of the source vector are not rearranged.

These functions require the work buffer for internal calculations, to compute the size of the required buffer, use the SortRadixIndexGetBufferSize function. Intervals between the elements of the source sparse vector *pSrc* in memory must be equal to the value of *srcStrideBytes*, minimum value of which is equal to the size of the data type of the key value. The sorting algorithm does not change the relative order of the elements with equal keys.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pBuffer* is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to zero, or *srcStrideBytes* is less than *sizeof(key type)*.

See Also

SortRadixIndexGetBufferSize Computes the size of the buffer for the `SortRadixIndexAscend` and `SortRadixIndexDescend` functions.

SwapBytes

Reverses the byte order of a vector.

Syntax

```
IppStatus ippsSwapBytes_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsSwapBytes_24u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsSwapBytes_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsSwapBytes_64u(const Ipp64u* pSrc, Ipp64u* pDst, int len);
IppStatus ippsSwapBytes_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSwapBytes_24u_I(Ipp8u* pSrcDst, int len);
```

```
IppStatus ippsSwapBytes_32u_I(Ipp32u* pSrcDst, int len);
IppStatus ippsSwapBytes_64u_I(Ipp64u* pSrcDst, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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 Values

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

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`SwapBytes.c`

Convert

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

Syntax

```
IppStatus ippsConvert_8s16s(const Ipp8s* pSrc,Ipp16s* pDst, int len);
IppStatus ippsConvert_8s32f(const Ipp8s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_8u8s_Sfs(const Ipp8u* pSrc, Ipp8s* pDst, int len, IppRoundMode
rndMode, int scaleFactor);
IppStatus ippsConvert_8s8u(const Ipp8s* pSrc, Ipp8u* pDst, int len);
```

```
IppStatus ippsConvert_16s8s_Sfs(const Ipp16s* pSrc, Ipp8s* pDst, Ipp32u len,
IppRoundMode rndMode, int scaleFactor);
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_64s64f(const Ipp64s* pSrc, Ipp64f* pDst, Ipp32u 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_64f8s_Sfs(const Ipp64f* pSrc, Ipp8s* pDst, int len, IppRoundMode
rndmode, int scaleFactor);
IppStatus ippsConvert_64f8u_Sfs(const Ipp64f* pSrc, Ipp8s* pDst, int len, IppRoundMode
rndmode, int scaleFactor);
IppStatus ippsConvert_64f16u_Sfs(const Ipp64f* pSrc, Ipp8s* pDst, int len, IppRoundMode
rndmode, int scaleFactor);
IppStatus ippsConvert_64s32s_Sfs(const Ipp64s* pSrc, Ipp32s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64f16s_Sfs(const Ipp64f* pSrc, Ipp16s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64f32s_Sfs(const Ipp64f* pSrc, Ipp32s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
```

```
IppStatus ippsConvert_64f64s_Sfs(const Ipp64f* pSrc, Ipp64s* pDst, Ipp32u 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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
rndMode	Rounding mode, the following values are possible: `ippRndZero` floating-point values are truncated to zero `ippRndNear` floating-point values are rounded to the nearest even integer when the fractional part equals 0.5; otherwise they are rounded to the nearest integer `ippRndFinancial` floating-point values are rounded down to the nearest integer when the fractional part is less than 0.5, or rounded up to the nearest integer if the fractional part is equal or greater than 0.5.

len	Number of elements in the vector.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function converts the type of data contained in the vector *pSrc* and stores the results in *pDst*.

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

Functions that operate with `16f` data do not support the `ippRndFinancial` rounding mode.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsRoundModeNotSupportedErr`	Indicates an error when the specified rounding mode is not supported.

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`Convert.c`

See Also

Integer Scaling

Conj

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

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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

pDst[n].*re* = *pSrc*[n].*re*

pDst[n].*im* = - *pSrc*[n].*im*

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

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

pSrcDst[n].*re* = *pSrcDst*[n].*re*

pSrcDst[n].*im* = - *pSrcDst*[n].*im*

Return Values

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

ConjFlip

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

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
len	Number of elements in the vector.

Description

This function computes the conjugate of the vector *pSrc* and stores the result, in reverse order, in *pDst*. The complex conjugate, stored in reverse order, is defined as follows:

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

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

Return Values

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

Magnitude

Computes the magnitudes of the elements of a complex vector.

Syntax

```
IppStatus ippsMagnitude_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f* pDst,
int len);
IppStatus ippsMagnitude_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f* pDst,
int len);
IppStatus ippsMagnitude_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsMagnitude_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp32f*
pDst, int len);
IppStatus ippsMagnitude_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16s*
pDst, int len, int scaleFactor);
IppStatus ippsMagnitude_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len, int
scaleFactor);
IppStatus ippsMagnitude_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len, int
scaleFactor);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

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:

$magn[n] = (pSrc[n].re^2 + pSrc[n].im^2)^{1/2}$

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:

$magn[n] = (pSrcRe[n]^2 + pSrcIm[n]^2)^{1/2}$

Return Values

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

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`Magnitude.c`

Phase

Computes the phase angles of elements of a complex vector.

Syntax

```
IppStatus ippsPhase_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPhase_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_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_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len, int
scaleFactor);
```

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

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 Values

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

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

Phase.c

PowerSpectr

Computes the power spectrum of a complex vector.

Syntax

```
IppStatus ippsPowerSpectr_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
```

```
IppStatus ippsPowerSpectr_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsPowerSpectr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len, int
scaleFactor);
```

```
IppStatus ippsPowerSpectr_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);
```

```
IppStatus ippsPowerSpectr_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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

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

To compute magnitudes, use the function `ippsMagnitude`.

Return Values

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

Real

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

Syntax

```
IppStatus ippsReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, int len);
```

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function returns the real part of the complex vector *pSrc* in the vector *pDstRe*.

Return Values

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

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`Real.c`

Imag

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

Syntax

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function returns the imaginary part of a complex vector *pSrc* in the vector *pDstIm*.

Return Values

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

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

Imag.c

RealToCplx

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

Syntax

```
IppStatus ippsRealToCplx_16s(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16sc* pDst,
int len);
```

```
IppStatus ippsRealToCplx_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32fc* pDst,
int len);
```

```
IppStatus ippsRealToCplx_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64fc* pDst,
int len);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function returns a complex vector *pDst* constructed from the real and imaginary parts of the input vectors *pSrcRe* and *pSrcIm*.

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

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

Note that the pointers cannot be both NULL.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* pointer is *NULL*NULL. The pointer *pSrcRe* or *pSrcIm* can be NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

RealToCplx.c

CplxToReal

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

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function returns the real and imaginary parts of a complex vector *pSrc* in two vectors *pDstRe* and *pDstIm*.

Return Values

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

Threshold

Performs the threshold operation on the elements of a vector by limiting the element values by specified value.

Syntax

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

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

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* parameter 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: when its value is ippCmpGreater – "greater than", when ippCmpLess - "less than", and determines whether *level* is an upper or lower bound for the input, respectively.

The formula for ippsThreshold called with the *relOp* = ippCmpLess is:

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

The formula for ippsThreshold called with the *relOp* = ippCmpGreater 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 *relOp* = ippCmpLess is:

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

The formula for complex `ippsThreshold` called with the *relOp* = `ippCmpGreater` 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 zero.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsBadArgErr`	Indicates an error when *relOp* has an invalid value.
`ippStsThreshNegLevelErr`	Indicates an error when *level* for the complex version is negative (see appendix A "Handling of Special Cases" for more information).

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`Threshold.c`

Threshold_LT, Threshold_GT

Performs the threshold operation on the elements of a vector by limiting the element values by the specified value.

Syntax

```
IppStatus ippsThreshold_LT_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
level);
```

```
IppStatus ippsThreshold_LT_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, Ipp32s
level);
```

```
IppStatus ippsThreshold_LT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
level);
```

```
IppStatus ippsThreshold_LT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f
level);
```

```
IppStatus ippsThreshold_LT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, Ipp32s level);
IppStatus ippsThreshold_GT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_GT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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 parameter *level* is always real.

The formula for complex ippsThreshold_LT is:

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

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

The formula for ippsThreshold_GT is the following:

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

For complex versions of the function ippsThreshold_GT, the parameter *level* 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 Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0
`ippStsThreshNegLevelErr`	Indicates an error when *level* for the complex version is negative (see appendix A "Handling of Special Cases" for more information).

Threshold_LTAbs, Threshold_GTAbs

Performs the threshold operation on the absolute values of elements of a vector.

Syntax

```
IppStatus ippsThreshold_LTAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
level);
IppStatus ippsThreshold_LTAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, Ipp32s
level);
IppStatus ippsThreshold_LTAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
level);
IppStatus ippsThreshold_LTAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f
level);

IppStatus ippsThreshold_GTAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
level);
IppStatus ippsThreshold_GTAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, Ipp32s
level);
IppStatus ippsThreshold_GTAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
level);
IppStatus ippsThreshold_GTAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f
level);
IppStatus ippsThreshold_GTAbs_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
```

```
IppStatus ippsThreshold_GTAbs_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_GTAbs_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GTAbs_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LTAbs_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_LTAbs_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LTAbs_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTAbs_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of source vector. This argument can not be negative.

Description

These functions implement thresholding of the vector *pSrc* by limiting absolute value of each element by the threshold value *level*. These functions perform the compare operation of the fixed type: `ippsThreshold_LTAbs` is for the "less than" comparison, while `ippsThreshold_GTAbs` is for the "greater than" comparison. Elements of the result vector *pDst* have the same sign that the source elements.

The in-place flavors perform the threshold operation on the vector *pSrcDst*.

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

$$pDst[n] = \begin{cases} level \text{ if } abs(pSrc[n]) < level, & pSrc[n] \geq 0 \\ -level \text{ if } abs(pSrc[n]) < level, & pSrc[n] < 0 \\ pSrc[n], \quad otherwise \end{cases}$$

ippsThreshold_GTAbs. The function `ippsThreshold_GTAbs` performs the operation “greater than” and *level* is an upper bound for the input. The formula for `ippsThreshold_GTAbs` is the following:

$$pDst[n] = \begin{cases} level \text{ if } abs(pSrc[n]) > level, & pSrc[n] \geq 0 \\ -level \text{ if } abs(pSrc[n]) > level, & pSrc[n] < 0 \\ pSrc[n], \quad otherwise \end{cases}$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error if *len* is less than or equal to zero.
ippStsThreshNegLevelErr	Indicates an error if *level* is negative.

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

Threshold_LtAbs.c

Threshold_LtAbs_I.c

Threshold_LTVal, Threshold_LTAbsVal, Threshold_GTVal, Threshold_LTValGTVal

Performs the threshold operation on the elements of a vector by limiting the element values by the specified level and replacing them with the specified value.

Syntax

```
IppStatus ippsThreshold_LTAbsVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTAbsVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTAbsVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTAbsVal_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, Ipp32s level, Ipp32s value);
IppStatus ippsThreshold_LTAbsVal_32f_I(const Ipp32f* pSrcDst, int len, Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTAbsVal_64f_I(const Ipp64f* pSrcDst, int len, Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTAbsVal_16s_I(const Ipp16s* pSrcDst, int len, Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTAbsVal_32s_I(const Ipp32s* pSrcDst, int len, Ipp32s level, Ipp32s value);
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_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_LTValGTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);
IppStatus ippsThreshold_LTValGTVal_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
Ipp32s levelLT, Ipp32s valueLT, Ipp32s levelGT, Ipp32s valueGT);
IppStatus ippsThreshold_LTValGTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);
IppStatus ippsThreshold_LTValGTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
Ipp64f levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);
IppStatus ippsThreshold_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_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_I(Ipp16s* pSrcDst, int len, Ipp16s levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);
IppStatus ippsThreshold_LTValGTVal_32s_I(Ipp32s* pSrcDst, int len, Ipp32s levelLT, Ipp32s valueLT, Ipp32s levelGT, Ipp32s valueGT);
IppStatus ippsThreshold_LTValGTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);
IppStatus ippsThreshold_LTValGTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
len	Number of elements in the vector.
level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real. For complex versions, it must be positive and represent magnitude.
levelLT	Low bound used to limit each element of *pSrc* or *pSrcDst* for the `ippsThreshold_LTValGTVal` function.
levelGT	Upper bound used to limit each element of *pSrc* or *pSrcDst* for the `ippsThreshold_LTValGTVal` function.
value	Value to be assigned to vector elements which are “less than” or “greater than” *level*.
valueLT	Value to be assigned to vector elements which are less than *levelLT* for the `ippsThreshold_LTValGTVal` function.
valueGT	Value to be assigned to vector elements which are greater than *levelGT* for the `ippsThreshold_LTValGTVal` function.

Description

These functions perform the threshold operation on the vector *pSrc* by limiting each element by the threshold level and replacing it with the specified 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_LTAbsVal. The `ippsThreshold_LTAbsVal` function substitutes each element of the source vector that is less by absolute value than specified *level* with the specified constant *value*.

The formula for `ippsThreshold_LTAbsVal` is:

```
if( ABS(x[i]) < level ) y[i] = value;
else y[i] = x[i];
```

ippsThreshold_LTVal. The function `ippsThreshold_LTVal` 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 parameter *`level`* 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 function `ippsThreshold_GTVal` 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 parameter *`level`* 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 function `ippsThreshold_LTValGTVal` checks both the "less than" and "greater than" conditions. The parameter *`levelLT`* is a lower bound and the parameter *`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`* must be less than or equal to *`levelGT`*.

The formula for `ippsThreshold_LTValGTVal` is:

$$pDst[n] = \begin{cases} valueLT, & pSrc[n] < levelLT \\ pSrc[n], & levelLT \le pSrc[n] \le levelGT \\ valueGT, & pSrc[n] > levelGT \end{cases}$$

For all complex versions, *level* must be positive and represent a magnitude.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsThresholdErr	Indicates an error when *levelLT* is greater than *levelGT*.
ippStsThreshNegLevelErr	Indicates an error when *level* for the complex version is negative (see appendix A "Handling of Special Cases" for more information).

Threshold_LTInv

Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.

Syntax

```
IppStatus ippsThreshold_LTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

level	Value used to limit each element of *pSrc* or *pSrcDst*.This argument must always be real and positive.

Description

This function computes the inverse of elements of the vector *pSrc* and stores the result in *pDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The in-place flavors of ippsThreshold_LTInv compute the inverse of elements of the vector *pSrcDst* and store the result in *pSrcDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The threshold operation is performed to avoid division by zero. Since *level* represents a magnitude, it is always real and must be positive. The formula for ippsThreshold_LTInv is the following:

$$pDst[n] = \begin{cases} \frac{1}{level}, & abs(pSrc[n]) = 0 \\ \frac{abs(pSrc[n])}{pSrc[n] \cdot level}, & 0 < abs(pSrc[n]) < level \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

If the function encounters zero-valued vector elements and *level* is also 0 (see appendix A "Handling of Special Cases"), the output value is set to Inf (infinity), but operation execution is not aborted:

$$pDst[n] = \begin{cases} Inf, & pSrc[n] = 0 \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to zero.
ippStsThreshNegLevelErr	Indicates an error when *level* is negative.
ippStsInvZero	Indicates a warning when *level* and a vector element are equal to zero. Operation execution is not aborted. The value of the destination vector element is Inf.

Example

To better understand how to use this function, refer to the following examples in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

Threshold_LtInv.c

Threshold_LtInv_I.c

CartToPolar

Converts the elements of a complex vector to polar coordinate form.

Syntax

```
IppStatus ippsCartToPolar_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
pDstMagn, Ipp32f* pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
pDstMagn, Ipp64f* pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_32fc(const Ipp32fc* pSrc, Ipp32f* pDstMagn, Ipp32f*
pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_64fc(const Ipp64fc* pSrc, Ipp64f* pDstMagn, Ipp64f*
pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDstMagn, Ipp16s*
pDstPhase, int len, int magnScaleFactor, int phaseScaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pSrcRe	Pointer to the source vector which stores the real components of Cartesian X,Y pairs.
pSrcIm	Pointer to the source vector which stores the imaginary components of Cartesian X,Y pairs.
pDstMagn	Pointer to the vector which stores the magnitude (radius) component of the elements of the vector *pSrc*.
pDstPhase	Pointer to the vector which stores the phase (angle) component of the elements of the vector *pSrc* in radians. Phase values are in the range (-π, π].
len	Number of elements in the vector.
magnScaleFactor	Integer scale factor for the magnitude component, refer to Integer Scaling.
phaseScaleFactor	Integer scale factor for the phase component, refer to Integer Scaling.

Description

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*.

Return Values

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

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

CartToPolar.c

PolarToCart

Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.

Syntax

```
IppStatus ippsPolarToCart_32f(const Ipp32f* pSrcMagn, const Ipp32f* pSrcPhase, Ipp32f*
pDstRe, Ipp32f* pDstIm, int len);
```

```
IppStatus ippsPolarToCart_64f(const Ipp64f* pSrcMagn, const Ipp64f* pSrcPhase, Ipp64f*
pDstRe, Ipp64f* pDstIm, int len);
```

```
IppStatus ippsPolarToCart_32fc(const Ipp32f* pSrcMagn, const Ipp32f* pSrcPhase,
Ipp32fc* pDst, int len);
```

```
IppStatus ippsPolarToCart_64fc(const Ipp64f* pSrcMagn, const Ipp64f* pSrcPhase,
Ipp64fc* pDst, int len);
```

```
IppStatus ippsPolarToCart_16sc_Sfs(const Ipp16s* pSrcMagn, const Ipp16s* pSrcPhase,
Ipp16sc* pDst, int len, int magnScaleFactor, int phaseScaleFactor);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrcMagn	Pointer to the source vector which stores the magnitude (radius) components of the elements in polar coordinate form.
pSrcPhase	Pointer to the vector which stores the phase (angle) components of the elements in polar coordinate form in radians.
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.

magnScaleFactor	Integer scale factor for the magnitude component, refer to Integer Scaling.
phaseScaleFactor	Integer scale factor for the phase component, refer to Integer Scaling.

Description

This function converts the polar form magnitude/phase pairs stored in the input vectors *pSrcMagn* and *pSrcPhase* into a complex vector and stores the results in the vector *pDst*, or stores the real components of the result in the vector *pDstRe* and the imaginary components in the vector *pDstIm*.

Return Values

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

MaxOrder

Computes the maximum order of a vector.

Syntax

```
IppStatus ippsMaxOrder_16s(const Ipp16s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32s(const Ipp32s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32f(const Ipp32f* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_64f(const Ipp64f* pSrc, int len, int* pOrder);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source vector.
len	Number of elements in the vector.
pOrder	Pointer to the result value.

Description

This function finds the maximum binary number in elements of the exponent vector *pSrc*, and stores the result in *pOrder*.

Return Values

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

ippStsNanArg	Indicates a warning when NaN is encountered in the input data vector.

Flip

Reverses the order of elements in a vector.

Syntax

```
IppStatus ippsFlip_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsFlip_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsFlip_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsFlip_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsFlip_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsFlip_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsFlip_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsFlip_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsFlip_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsFlip_64fc_I(Ipp64fc* pSrcDst, int len);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function stores the elements of a source vector *pSrc* to a destination vector *pDst* in reverse order according to the following formula:

pDst[n] = *pSrc*[*len*- n-1], n = 0 .. *len*-1

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

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

FindNearestOne

Finds an element of the table which is closest to the specified value.

Syntax

```
IppStatus ippsFindNearestOne_16u(Ipp16u inpVal, Ipp16u* pOutVal, int* pOutIndex, const
Ipp16u *pTable, int tblLen);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

inpVal	Reference value.
pOutVal	Pointer to the output value.
pOutIndex	Pointer to the output index.
pTable	Pointer to the table for searching.
tblLen	Number of elements in the table.

Description

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 must satisfy the condition *pTable*[n] ≤*pTable*[n+1]. The function uses the following distance criterion for determining the table closest element closest: min(|*inpVal* -*pTable*[n]|).

Return Values

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

FindNearest

Finds table elements that are closest to the elements of the specified vector.

Syntax

```
IppStatus ippsFindNearest_16u(const Ipp16u* pVals, Ipp16u* pOutVals, int* pOutIndexes,
int len, const Ipp16u *pTable, int tblLen);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pVals	Pointer to the vector containing reference values.
pOutVals	Pointer to the output vector.
pOutIndexes	Pointer to the array that stores output indexes.
len	Number of elements in the input vector.
pTable	Pointer to the table for searching.
tblLen	Number of elements in the table.

Description

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 must satisfy the condition *pTable*[n] ≤*pTable*[n+1]. The function uses the following distance criterion for determining the table element closest to *pVals*[k] : min(|*pVals*[k] – *pTable*[n]|).

Return Values

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

Example

To better understand how to use this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`FindNearest.c`

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
- WinHann windowing function
- WinKaiser 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 the following code example.

Example

```
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 );
     }
}
```

For more information on windowing, see: [Jack89], section 7.3, *Windows in Spectrum Analysis*; [Jack89], section 9.1, *Window-Function Technique*; and [Mit93], section 16-2, *Fourier Analysis of Finite-Time Signals*. For more information on these references, see also the Bibliography at the end of this document.

WinBartlett

Multiplies a vector by a Bartlett windowing function.

Syntax

```
IppStatus ippsWinBartlett_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBartlett_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBartlett_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBartlett_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinBartlett_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinBartlett_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinBartlett_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBartlett_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBartlett_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBartlett_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBartlett_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBartlett_64fc_I(Ipp64fc* pSrcDst, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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:

$$w_{bartlett}(n) = \begin{cases} \frac{2n}{len-1}, & 0 \le n \le \frac{len-1}{2} \\ 2 - \frac{2n}{len-1}, & \frac{len-1}{2} < n \le len-1 \end{cases}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or*pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than 3.

Example

The example below shows how to use the function `ippsWinBartlett_32f_I`.

```
void bartlett(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBartlett_32f_I(x, 8);
      printf_32f("bartlett (half) =", x, 4, ippStsNoErr);
}
```

Output:

```
    bartlett (half) =  0.000000 0.285714 0.571429 0.857143
Matlab* Analog:
    >> b = bartlett(8); b(1:4)'
```

WinBlackman

Multiplies a vector by a Blackman windowing function.

Syntax

```
IppStatus 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(const Ipp64f* pSrc, Ipp64f* pDst, int len, double alpha);
IppStatus ippsWinBlackman_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, 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 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 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 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);
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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
alpha	Adjustable parameter associated with the Blackman windowing equation.
len	Number of elements in the vector

Description

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:

$$w_{blackman}(n) = \frac{alpha+1}{2} - 0.5\cos\left(\frac{2\pi n}{len-1}\right) - \frac{alpha}{2}\cos\left(\frac{4\pi n}{len-1}\right)$$

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

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 4 for the function ippsWinBlackmanOpt and less than 3 for all other functions of the family.

Example

The example below shows how to use the function ippsWinBlackmanStd_32f_I

```
void blackman(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBlackmanStd_32f_I(x, 8);
      printf_32f("blackman (half) =", x, 4, ippStsNoErr);
}
```

Output:

```
    blackman(half) = 0.000000 0.090453 0.459183 0.920364
Matlab* Analog:
    >> b = blackman(8)'; b(1:4)
```

WinHamming

Multiplies a vector by a Hamming windowing function.

Syntax

```
IppStatus ippsWinHamming_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHamming_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHamming_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHamming_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHamming_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHamming_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHamming_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHamming_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHamming_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHamming_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHamming_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHamming_64fc_I(Ipp64fc* pSrcDst, int len);
```

Include Files

ipps.h

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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:

$$w_{hamming}(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{len-1}\right)$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than 3.

Example

The example below shows how to use the function `ippsWinHamming_32f_I`.

```
void hamming(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinHamming_32f_I(x, 8);
      printf_32f("hamming(half) =", x, 4, ippStsNoErr);
}
```

Output:

```
 hamming(half) =  0.080000 0.253195 0.642360 0.954446
Matlab* Analog:
    >> b = hamming(8); b(1:4)'
```

WinHann

Multiplies a vector by a Hann windowing function.

Syntax

```
IppStatus ippsWinHann_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHann_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHann_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHann_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHann_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHann_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHann_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHann_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHann_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHann_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHann_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHann_64fc_I(Ipp64fc* pSrcDst, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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:

$$w_{hann}(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{len-1}\right)$$

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 3.

Example

The example below shows how to use the function ippsWinHann_32f_I

```
void hann(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinHann_32f_I(x, 8);
      printf_32f("hann(half) =", x, 4, ippStsNoErr);
}
```

Output:

```
 hann(half) =  0.000000 0.188255 0.611260 0.950484
Matlab* Analog:
    >> N = 8; n = 0:N-1; 0.5*(1-cos(2*pi*n/(N-1)))
```

WinKaiser

Multiplies a vector by a Kaiser windowing function.

Syntax

```
IppStatus ippsWinKaiser_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, float alpha);
IppStatus ippsWinKaiser_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, float alpha);
IppStatus ippsWinKaiser_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f 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, Ipp64f
alpha);
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, Ipp64f 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, Ipp64f alpha);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operation.
alpha	Adjustable parameter associated with the Kaiser windowing equation.
len	Number of elements in the vector.

Description

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.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst*, *pSrc*, or *pSrcDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than 1.
`ippStsHugeWinErr`	Indicates an error when the Kaiser window is too big.

Example

The example below shows how to use the function `ippsWinKaiser_32f_I`.

```
void kaiser(void) {
     Ipp32f x[8];
     IppStatus st;
     ippsSet_32f(1, x, 8);
     st = ippsWinKaiser_32f_I( x, 8, 1.0f );
     printf_32f("kaiser(half) =", x, 4, ippStsNoErr);
}
```

Output:

```
 kaiser(half) =  0.135534 0.429046 0.755146 0.970290
Matlab* Analog:
    >> kaiser(8,7/2)'
```

Statistical Functions

This section describes the Intel IPP functions that compute the vector measure values: maximum, minimum, mean, and standard deviation.

Sum

Computes the sum of the elements of a vector.

Syntax

```
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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pSum	Pointer to the output result.
len	Number of elements in the vector.
hint	Suggests using specific code. The possible values for the *hint* argument are described in Hint Arguments.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

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.

Return Values

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

Example

The example below shows how to use the function ippsSum.

```
void sum(void) {
      Ipp16s x[4] = {-32768, 32767, 32767, 32767}, sm;
      ippsSum_16s_Sfs(x, 4, &sm, 1);
      printf_16s("sum =", &sm, 1, ippStsNoErr);
}
```

Output:

```
  sum = 32766
Matlab* Analog:
    >> x = [-32768, 32767, 32767, 32767]; sum(x)/2
```

Max

Returns the maximum value of a vector.

Syntax

```
IppStatus ippsMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax);
IppStatus ippsMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax);
IppStatus ippsMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax);
IppStatus ippsMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function returns the maximum value of the input vector *pSrc*, and stores the result in *pMax*.

Return Values

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

MaxIndx

Returns the maximum value of a vector and the index of the maximum element.

Syntax

```
IppStatus ippsMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax, int* pIndx);
IppStatus ippsMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax, int* pIndx);
IppStatus ippsMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax, int* pIndx);
IppStatus ippsMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax, int* pIndx);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source vector.
pMax	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the maximum element.

Description

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 Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example

The code example below demonstrates how to use the function ippsMaxIndx.

```
Ipp16s src[] = { 1, -2, 3, 8, -6 };
Ipp16s max;
int len = 5;
int  indx;

ippsMaxIndx_16s ( src, len, &max, &indx );
```

Result:

```
max = 8  indx = 3
```

MaxAbs

Returns the maximum absolute value of a vector.

Syntax

```
IppStatus ippsMaxAbs_16s(const Ipp16s* pSrc, int len, Ipp16s* pMaxAbs);
IppStatus ippsMaxAbs_32s(const Ipp32s* pSrc, int len, Ipp32s* pMaxAbs);
IppStatus ippsMaxAbs_32f(const Ipp32f* pSrc, int len, Ipp32f* pMaxAbs);
IppStatus ippsMaxAbs_64f(const Ipp64f* pSrc, int len, Ipp64f* pMaxAbs);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function returns the maximum absolute value of the input vector *pSrc*, and stores the result in *pMaxAbs*.

Return Values

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

Example

The example below shows how to use the function `ippsMaxAbs_16s`.

```
Ipp16s src[5] = { 2, -8, -3, -1, 7 };
Ipp16s maxAbs;
ippsMaxAbs_16s ( src, 5, &maxAbs );
```

Result:

```
maxAbs = 8
```

MaxAbsIndx

Returns the maximum absolute value of a vector and the index of the corresponding element.

Syntax

```
IppStatus ippsMaxAbsIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMaxAbs, int* pIndx);
IppStatus ippsMaxAbsIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMaxAbs, int* pIndx);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pMaxAbs	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the maximum element.

Description

This function returns the maximum absolute value *pMaxAbs* of the input vector *pSrc*, and the index of the corresponding element *pIndx*. If there are several elements with the equal maximum absolute value, the first index from the beginning is returned.

Return Values

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

Min

Returns the minimum value of a vector.

Syntax

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

```
IppStatus ippsMin_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin);
IppStatus ippsMin_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin);
IppStatus ippsMin_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

This function returns the minimum value of the input vector *pSrc*, and stores the result in *pMin*.

Return Values

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

Example

The example below shows how to use the function `ippsMin`.

```
Ipp16s src = { 1, -2, 3, 8, -6};
Ipp16s min;
int len = 5;
ippsMin_16s (src, len, &min );
```

Result:

```
min = -6
```

MinIndx

Returns the minimum value of a vector and the index of the minimum element.

Syntax

```
IppStatus ippsMinIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, int* pIndx);
IppStatus ippsMinIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, int* pIndx);
IppStatus ippsMinIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, int* pIndx);
IppStatus ippsMinIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, int* pIndx);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source vector.
pMin	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the minimum element.

Description

This function returns the minimum value of the input vector *pSrc* and stores the result in *pMin*. If *pIndx* is not a NULL pointer, the function returns the index of the minimum element and stores it in *pIndx*. If there are several equal minimum elements, the first index from the beginning is returned.

Return Values

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

MinAbs

Returns the minimum absolute value of a vector.

Syntax

```
IppStatus ippsMinAbs_16s(const Ipp16s* pSrc, int len, Ipp16s* pMinAbs);
IppStatus ippsMinAbs_32s(const Ipp32s* pSrc, int len, Ipp32s* pMinAbs);
IppStatus ippsMinAbs_16f(const Ipp16f* pSrc, int len, Ipp16f* pMinAbs);
IppStatus ippsMinAbs_32f(const Ipp32f* pSrc, int len, Ipp32f* pMinAbs);
IppStatus ippsMinAbs_64f(const Ipp64f* pSrc, int len, Ipp64f* pMinAbs);
```

Include Files

ipps.h

Parameters

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

Description

This function returns the minimum absolute value of the input vector *pSrc*, and stores the result in *pMinAbs*.

Return Values

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

MinAbsIndx

Returns the minimum absolute value of a vector and the index of the corresponding element.

Syntax

```
IppStatus ippsMinAbsIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMinAbs, int* pIndx);
IppStatus ippsMinAbsIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMinAbs, int* pIndx);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source vector.
pMinAbs	Pointer to the output result.
len	Number of elements in the vector.
pIndx	Pointer to the index value of the corresponding element.

Description

This function returns the minimum absolute value *pMinAbs* of the input vector *pSrc*, and the index of the corresponding element *pIndx*. If there are several elements with equal maximum absolute value, the first index from the beginning is returned.

Return Values

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

MinMax

Returns the maximum and minimum values of a vector.

Syntax

```
IppStatus ippsMinMax_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, Ipp8u* pMax);
IppStatus ippsMinMax_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin, Ipp16u* pMax);
```

```
IppStatus ippsMinMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, Ipp16s* pMax);
IppStatus ippsMinMax_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin, Ipp32u* pMax);
IppStatus ippsMinMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, Ipp32s* pMax);
IppStatus ippsMinMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, Ipp32f* pMax);
IppStatus ippsMinMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, Ipp64f* pMax);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pMin	Pointer to the minimum value.
pMax	Pointer to the maximum value.
len	Number of elements in the vector.

Description

This function returns the minimum and maximum values of the input vector *pSrc*, and stores the results in *pMin* and *pMax*, respectively.

Return Values

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

MinMaxIndx

Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.

Syntax

```
IppStatus ippsMinMaxIndx_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, int* pMinIndx,
Ipp8u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin, int* pMinIndx,
Ipp16u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, int* pMinIndx,
Ipp16s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin, int* pMinIndx,
Ipp32u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, int* pMinIndx,
Ipp32s* pMax, int* pMaxIndx);
```

```
IppStatus ippsMinMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, int* pMinIndx,
Ipp32f* pMax, int* pMaxIndx);
```

```
IppStatus ippsMinMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, int* pMinIndx,
Ipp64f* pMax, int* pMaxIndx);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pMin	Pointer to the minimum value.
pMax	Pointer to the maximum value.
len	Number of elements in the vector.
pMinIndx	Pointer to the index value of the minimum element.
pMaxIndx	Pointer to the index value of the maximum element.

Description

This function returns the minimum and maximum values of the input vector *pSrc* and stores the result in *pMin* and *pMax*, respectively. The function also returns the indexes of the minimum and maximum elements and stores them in *pMinIndx* and *pMaxIndx*, respectively. If there are several equal minimum or maximum elements, the first index from the beginning is returned.

Return Values

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

ReplaceNAN

Replaces not-a-number (NaN) values of vector elements with a constant value.

Syntax

```
IppStatus ippsReplaceNAN_32f_I(Ipp32f* pSrcDst, int len, Ipp32f value);
```

```
IppStatus ippsReplaceNAN_64f_I(Ipp64f* pSrcDst, int len, Ipp64f value);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrcDst	Pointer to the source and destination vector.
len	Number of elements in the vector.
value	Constant value to be assigned to NaN elements of the vector.

Description

This function replaces not-a-number (NaN) elements of the source vector with *value*, other vector elements remain unchanged:

```
pSrcDst[i] = (pSrcDst[i] == NAN) ? value : pSrcDst[i]
```

Return Values

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

Mean

Computes the mean value of a vector.

Syntax

```
IppStatus ippsMean_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean, IppHintAlgorithm hint);
IppStatus ippsMean_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pMean, IppHintAlgorithm hint);
IppStatus ippsMean_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean);
IppStatus ippsMean_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pMean);
IppStatus ippsMean_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean, int scaleFactor);
IppStatus ippsMean_32s_Sfs(const Ipp32s* pSrc,int len,Ipp32s* pMean,int scaleFactor);
IppStatus ippsMean_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pMean, int scaleFactor);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

hint	Suggests using specific code. The possible values for the *hint* argument are described in Hint Arguments.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

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.

Return Values

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

Example

The example below shows how to use the function ippsMean_32f

```
void mean(void) {
      Ipp32f *x = ippsMalloc_32f(1000), mean;
      int i;
      for(i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsMean_32f(x, 1000, &mean, ippAlgHintFast);
      printf_32f("mean =", &mean, 1, ippStsNoErr);
      ippsFree(x);
}
```

Output:

```
 mean = 0.492591
Matlab* Analog:
    >> x = rand(1,1000); mean(x)
```

StdDev

Computes the standard deviation value of a vector.

Syntax

```
IppStatus ippsStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pStdDev, IppHintAlgorithm hint);
```

```
IppStatus ippsStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pStdDev);
```

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

```
IppStatus ippsStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pStdDev, int scaleFactor);
```

Include Files

ipps.h

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pStdDev	Pointer to the output result.
len	Number of elements in the vector.
hint	Suggests using specific code. The possible values for the *hint* argument are described in Hint Arguments.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function computes the standard deviation of the input vector *pSrc*, and stores the result in *pStdDev*. The vector length can not be less than 2. The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$stdDev = \sqrt{\frac{\sum_{n=0}^{len-1} (pSrc[n] - mean(pSrc))^2}{len-1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pStdDev* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 1.

Example

The example below shows how to use the function `ippsStdDev_32f`.

```
void stdev(void) {
      Ipp32f *x = ippsMalloc_32f(1000), stdev;
      int i;
      for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsStdDev_32f(x, 1000, &stdev, ippAlgHintFast);
      printf_32f("stdev =", &stdev, 1, ippStsNoErr);
      ippsFree(x);
}
```

Output:

```
 stdev = 0.286813
Matlab* Analog:
    >> x = rand(1,1000); std(x)
```

MeanStdDev

Computes the mean value and the standard deviation value of a vector.

Syntax

```
IppStatus ippsMeanStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean, Ipp32f*
pStdDev, IppHintAlgorithm hint);
```

```
IppStatus ippsMeanStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean, Ipp64f*
pStdDev);
```

```
IppStatus ippsMeanStdDev_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean, Ipp16s*
pStdDev, int scaleFactor);
```

```
IppStatus ippsMeanStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pMean, Ipp32s*
pStdDev, int scaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pMean	Pointer to the output result - mean value.
pStdDev	Pointer to the output result - standard deviation.
len	Number of elements in the vector
hint	Suggests using specific code. The possible values for the *hint* argument are described in Hint Arguments.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function computes both the mean value and the standard deviation of the input vector *pSrc*, and stores the results in *pMean* and *pStdDev* respectively. The vector length can not be less than 2. The mean of *pSrc* is defined by the formula:

$$mean = \frac{1}{len}\sum_{n=0}^{len-1} pSrc[n]$$

The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$stdDev = \sqrt{\frac{\sum_{n=0}^{len-1} (pSrc[n] - mean(pSrc))^2}{len-1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 1.

Norm

Computes the C, L1, L2, or L2Sqr norm of a vector.

Syntax

```
IppStatus ippsNorm_Inf_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_32fc32f(const Ipp32fc* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);

IppStatus ippsNorm_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_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_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s* pNorm, int
scaleFactor);
IppStatus ippsNorm_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s* pNorm, int
scaleFactor);
IppStatus ippsNorm_Inf_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm, int
scaleFactor);
IppStatus ippsNorm_L1_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm, int
scaleFactor);
```

```
IppStatus ippsNorm_L1_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s* pNorm, int scaleFactor);
IppStatus ippsNorm_L2_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm, int scaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pNorm	Pointer to the output result.
len	Number of elements in the vector.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function computes the `C`, `L1`, `L2`, or `L2Sqr` norm of the source vector *pSrc* and stores the result in *pNorm*.

ippsNorm_Inf. The function `ippsNorm_Inf` computes the `C` norm defined by the formula:

$$Norm_C = \max_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L1. The function `ippsNorm_L1` computes the `L1` norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L2. The function `ippsNorm_L2` computes the `L2` norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc[n]|^2}$$

ippsNorm_L2Sqr. The function `ippsNorm_L2Sqr` computes the `L2Sqr` norm defined as square of the `L2` norm.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value.

Return Values

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

NormDiff

Computes the C, L1, L2, or L2Sqr norm of two vectors' difference.

Syntax

```
IppStatus ippsNormDiff_Inf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_32fc32f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int len,
Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int len,
Ipp64f* pNorm);
IppStatus ippsNormDiff_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_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_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
len, Ipp64s* pNorm, int scaleFactor);
IppStatus ippsNormDiff_Inf_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
len, Ipp32s* pNorm, int scaleFactor);
```

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

```
IppStatus ippsNormDiff_L1_16s64s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len,
Ipp64s* pNorm, int scaleFactor);
```

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

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc1, *pSrc2*	Pointers to the two source vectors; *pSrc2* can be `NULL`.
pNorm	Pointer to the output result.
len	Number of elements in the vector.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function computes the `C`, `L1`, `L2`, or `L2Sqr` norm of the source vectors' difference, and stores the result in *pNorm*.

ippsNormDiff_Inf. The function `ippsNormDiff_Inf` computes the `C` norm defined by the formula:

$$Norm_{Inf} = \max_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L1. The function `ippsNormDiff_L1` computes the `L1` norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L2. The function `ippsNormDiff_L2` computes the `L2` norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|^2}$$

ippsNormDiff_L2Sqr. The function `ippsNormDiff_L2Sqr` computes the `L2Sqr` norm defined as square of the `L2` norm.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrc1*, *pSrc2*, or *pNorm* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function ippsNormDiff.

```
int norm( void ) {
      Ipp16s x[LEN];
      Ipp32f Norm[3];
      IppStatus st;
      int i;
      for( i=0; i<LEN; ++i ) x[i] = (Ipp16s)rand();
      ippsNormDiff_Inf_16s32f( x, 0, LEN, Norm );
      ippsNormDiff_L1_16s32f( x, 0, LEN, Norm+1 );
      st = ippsNormDiff_L2_16s32f( x, 0, LEN, Norm+2 );
      printf_32f("Norm (oo,L1,L2) =", Norm, 3, st );
      return Norm[2] <= Norm[1] && Norm[1] <= LEN*Norm[0];
}
```

Output:

```
 Norm (oo,L1,L2) =  31993.000000 1526460.000000 180270.781250
Matlab* analog:
    >> x = 32767*rand(1,100);norm(x,inf),norm(x,1),norm(x,2)
```

DotProd

Computes the dot product of two vectors.

Syntax

IppStatus ippsDotProd_32f(const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, int *len*, Ipp32f* *pDp*);

IppStatus ippsDotProd_32fc(const Ipp32fc* *pSrc1*, const Ipp32fc* *pSrc2*, int *len*, Ipp32fc* *pDp*);

IppStatus ippsDotProd_32f32fc(const Ipp32f* *pSrc1*, const Ipp32fc* *pSrc2*, int *len*, Ipp32fc* *pDp*);

IppStatus ippsDotProd_32f64f(const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, int *len*, Ipp64f* *pDp*);

IppStatus ippsDotProd_32fc64fc(const Ipp32fc* *pSrc1*, const Ipp32fc* *pSrc2*, int *len*, Ipp64fc* *pDp*);

IppStatus ippsDotProd_32f32fc64fc(const Ipp32f* *pSrc1*, const Ipp32fc* *pSrc2*, int *len*, Ipp64fc* *pDp*);

IppStatus ippsDotProd_64f(const Ipp64f* *pSrc1*, const Ipp64f* *pSrc2*, int *len*, Ipp64f* *pDp*);

IppStatus ippsDotProd_64fc(const Ipp64fc* *pSrc1*, const Ipp64fc* *pSrc2*, int *len*, Ipp64fc* *pDp*);

IppStatus ippsDotProd_64f64fc(const Ipp64f* *pSrc1*, const Ipp64fc* *pSrc2*, int *len*, Ipp64fc* *pDp*);

```
IppStatus ippsDotProd_16s64s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len, Ipp64s* pDp);
IppStatus ippsDotProd_16sc64sc(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2, int len, Ipp64sc* pDp);
IppStatus ippsDotProd_16s16sc64sc(const Ipp16s* pSrc1, const Ipp16sc* pSrc2, int len, Ipp64sc* pDp);
IppStatus ippsDotProd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len, Ipp32f* pDp);
IppStatus ippsDotProd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, int len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_16s32s32s_Sfs(const Ipp16s* pSrc1, const Ipp32s* pSrc2, int len, Ipp32s* pDp, int scaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc1	Pointer to the first vector to compute the dot product value.
pSrc2	Pointer to the second vector to compute the dot product value.
pDp	Pointer to the output result.
len	Number of elements in the vector.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

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:

$$dp = \sum_{n=0}^{len-1} pSrc1[n] * pSrc2[n]$$

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.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDp*, *pSrc1*, or *pSrc*2 pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example

The example below 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.

```
void dotprod(void) {
      Ipp64f x[10], dp;
      int n;
      for (n = 0; n<10; ++n) x[n] = sin(IPP_2PI * n / 8);
      ippsDotProd_64f(x, x+2, 8, &dp);
      printf_64f("dp =", &dp, 1, ippStsNoErr);
}
```

Output:

```
 dp = 0.000000
Matlab* Analog:
    >> n = 0:9; x = sin(2*pi*n/8); a = x(1:8); b = x(3:10); a*b'
```

MaxEvery, MinEvery

Computes maximum or minimum value for each pair of elements of two vectors.

Syntax

```
IppStatus ippsMaxEvery_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, Ipp32u len);
IppStatus ippsMaxEvery_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, Ipp32u len);
IppStatus ippsMaxEvery_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, Ipp32u len);
IppStatus ippsMaxEvery_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, Ipp32u len);
IppStatus ippsMinEvery_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, Ipp32u len);
IppStatus ippsMinEvery_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, Ipp32u len);
IppStatus ippsMinEvery_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, Ipp32u len);
IppStatus ippsMinEvery_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, Ipp32u len);
IppStatus ippsMaxEvery_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsMaxEvery_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsMaxEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMaxEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMaxEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMaxEvery_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, Ipp32u len);
IppStatus ippsMinEvery_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsMinEvery_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
```

```
IppStatus ippsMinEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMinEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMinEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMinEvery_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, Ipp32u len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc, *pSrc1*, *pSrc2*	Pointer to the input vector.
pSrcDst, *pDst*	Pointer to the vector which stores the result.
len	Number of elements in the vector.

Description

This function computes the maximum between each pair of corresponding elements of two input vectors and stores the result in *pSrcDst*.

This function computes minimum values likewise.

Return Values

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

ZeroCrossing

Computes specific zero crossing measure.

Syntax

```
IppStatus ippsZeroCrossing_16s32f(const Ipp16s* pSrc, Ipp32u len, Ipp32f* pValZCR,
IppsZCType zcType);
IppStatus ippsZeroCrossing_32f(const Ipp32f* pSrc, Ipp32u len, Ipp32f* pValZCR,
IppsZCType zcType);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
len	Number of elements in the vector.
pValZCR	Pointer to the output value of the zero crossing measure.
zcType	Type of the zero crossing measure, possible values are `ippsZCR`, `ippsZCXor` or `ippsZCC`.

Description

This function computes specific zero crossing measure according to the parameter *zcType*. The result of zero crossing measurement is stored in *pValZCR*. The calculations are performed in accordance with the formulas below.

If *zcType* = `ippZCR`, the function uses the following formula:

$$\sum_{i=1}^{len-1} (x_i \cdot x_{i-1}) < 0$$

If *zcType* = `ippZCXor`, the function uses the following formula:

$$\sum_{i=1}^{len-1} sign(x_i) \wedge sign(x_{i-1}) \text{ , where } sign(x) = \begin{cases} 0: & x > 0, \ x = +0 \\ 1: & x < 0, \ x = -0 \end{cases};$$

If *zcType* = `ippZCC`, the function uses the following formula:

$$\sum_{i=1}^{len-1} \frac{abs(sign(x_i) - sign(x_{i-1}))}{2} \text{, where } sign(x) = \begin{cases} 1: & x > 0 \\ 0: & x = 0. \\ -1: & x < 0 \end{cases}$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pSrc* or *pValZCR* pointer is *NULL*.
`ippStsRangeErr`	Indicates an error when *zcType* has an invalid value.

CountInRange

Computes the number of elements of the vector whose values are in the specified range.

Syntax

```
IppStatus ippsCountInRange_32s(const Ipp32s* pSrc, int len, int* pCounts, Ipp32s
lowerBound, Ipp32s upperBound);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the first input vector.
pCounts	Pointer to the second input vector which stores the result.
len	Number of elements in the vector.
lowerBound	Lower boundary of the range.
upperBound	Upper boundary of the range.

Description

This function computes the number of elements of the vector *pSrc* whose values are in the range *lowerBound* < *pSrc*[n] < *upperBound*. The total number of such elements are stored in the *pCounts*.

Return Values

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

Sampling Functions

The functions described in this section manipulate signal samples. Sampling functions are used to change the sampling rate of the input signal and thus to obtain the signal vector of a required length. The functions perform the following operations:

- Insert zero-valued samples between neighboring samples of a signal (up-sample).
- Remove samples from between neighboring samples of a signal (down-sample).

The upsampling and downsampling functions are used by some filtering functions described in Chapter 6.

SampleUp

Up-samples a signal, conceptually increasing its sampling rate by an integer factor.

Syntax

```
IppStatus ippsSampleUp_16s (const Ipp16s* pSrc, int srcLen, Ipp16s* pDst, int* pDstLen,
int factor, int* pPhase);
```

```
IppStatus ippsSampleUp_32f (const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int* pDstLen,
int factor, int* pPhase);
```

```
IppStatus ippsSampleUp_64f (const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int* pDstLen,
int factor, int* pPhase);
```

```
IppStatus ippsSampleUp_16sc (const Ipp16sc* pSrc, int srcLen, Ipp16sc* pDst, int*
pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleUp_32fc (const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst, int*
pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleUp_64fc (const Ipp64fc* pSrc, int srcLen, Ipp64fc* pDst, int*
pDstLen, int factor, int* pPhase);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source array (the signal to be up-sampled).
srcLen	Number of samples in the source array *pSrc*.
pDst	Pointer to the destination array.
pDstLen	Pointer to the length of the destination array *pDst*.
factor	Factor by which the signal is up-sampled. That is, *factor*–1 zeros are inserted after each sample of the source 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].

Description

This function up-samples the *srcLen*-length source 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 destination array begin with the corresponding source array sample, the other *factor*–1 samples are equal to 0. The length of the destination array is stored by the *pDstLen* address.

The *pPhase* value is the phase of an source array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

The `ippsSampleUp` functionality can be described as follows:

$pDst[factor * n + phase] = pSrc[n], \ 0 \le n < srcLen$

$pDst[factor * n + m] = 0, \ 0 \le n < srcLen, \ 0 \le m < factor, \ m \ne phase$

pDstLen = *factor* * *srcLen*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pDst*, *pSrc*, *pDstLen*, or *pPhase* pointer is NULL.
ippStsSizeErr	Indicates an error if *srcLen* is less than or equal to 0.
ippStsSampleFactorErr	Indicates an error if *factor* is less than or equal to 0.
ippStsSamplePhaseErr	Indicates an error when *pPhase* is negative, or bigger than or equal to *factor*.

SampleDown

Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

Syntax

```
IppStatus ippsSampleDown_16s(const Ipp16s* pSrc, int srcLen, Ipp16s* pDst, int*
pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int*
pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int*
pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_16sc(const Ipp16sc* pSrc, int srcLen, Ipp16sc* pDst, int*
pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst, int*
pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc* Dst, int*
pDstLen, int factor, int* pPhase);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source array holding the samples to be down-sampled.
srcLen	Number of samples in the input array *pSrc*.
pDst	Pointer to the destination array.
pDstLen	Pointer to the length of the destination 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].

Description

This function down-samples the *srcLen*-length source 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 length of the destination array is stored by the *pDstLen* address.

The *pPhase* value is the phase of an source 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 ≤ n < *pDstLen*

phase = (*factor*+ *phase* - *srcLen* %*factor*)%*factor*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pDst*, *pSrc*, *pDstLen*, or *pPhase* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *srcLen* is less than or equal to 0.
`ippStsSampleFactorErr`	Indicates an error when *factor* is less than or equal to 0.
`ippStsSamplePhaseErr`	Indicates an error when *pPhase* is negative, or bigger than or equal to *factor*.

Example

The example below shows how to use the function `ippsSampleDown`.

```
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-sampling =", z, dstLen1+dstLen2, st);
}
```

Output:

```
down-sampling = 3 6 9 12 15
```

Filtering Functions

6

This chapter describes the Intel® IPP functions that perform convolution and correlation operations, as well as linear and non-linear filtering.

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 in this section estimate either the auto-correlation of a source vector or the cross-correlation of two vectors.

Special Arguments

Some Convolution and Correlation functions described in this section have two implementations of the algorithm:

- For small data size, function processes data as described by the formula
- For big data size, function uses FFT-inherited algorithms

The optimal algorithm is selected automatically according to the input data size. You can manually choose which algorithm to use by passing one of the following predefined values to the *algType* parameter of the function:

ippAlgAuto	Select the optimal algorithm automatically.
ippAlgDirect	Use direct algorithm as described by the formula.
ippAlgFFT	Use FFT-based algorithm implementation.

These values are declared in the `IppAlgType` enumerator.

Several functions support normalization of the output data. You can choose which normalization to apply by passing one of the following values to the function:

ippsNormNone	No normalization (default).
ippsNormA	Biased normalization.
ippsNormB	Unbiased normalization.

These values are declared in the `IppsNormOp` enumerator.

See Also

Enumerators

AutoCorrNormGetBufferSize

Computes the size of the work buffer for the `ippsAutoCorrNorm` *function.*

Syntax

```
IppStatus ippsAutoCorrNormGetBufferSize (int srcLen, int dstLen, IppDataType dataType,
IppEnum algType, int* pBufferSize);
```

Include Files

ipps.h

Parameters

srcLen	Number of elements in the source vector.
dstLen	Number of elements in the destination vector (length of auto-correlation).
dataType	Data type for auto-correlation. Possible values are ipp32f, ipp32fc, ipp64f, or ipp64fc.
algType	Bit-field mask for the algorithm type definition. Possible values are the results of composition of the IppAlgType and IppsNormOp values.
pBufferSize	Pointer to the size of the work buffer.

Description

The ippsAutoCorrNormGetBufferSize function computes the size in bytes of the external work buffer needed for the function that performs auto-correlation. The result is stored in the *pBufferSize* parameter.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pBufferSize* is NULL.
ippStsSizeErr	Indicates an error when *srcLen* or *dstLen* is less than, or equal to zero.
ippStsAlgTypeErr	Indicates an error when: • the result of the bitwise AND operation between the algType and ippAlgMask differs from the ippAlgAuto, ippAlgDirect, or ippAlgFFT values; • the result of the bitwise AND operation between the algType and ippsNormMask differs from the ippsNormNone, ippsNormA, or ippsNormB values.
ippStsDataTypeErr	Indicates an error when the *dataType* value differs from the ipp32f, ipp32fc, ipp64f, or ipp64fc.

See Also

Enumerators
Special Arguments
AutoCorrNorm Calculates normal, biased, and unbiased auto-correlation of a vector.

AutoCorrNorm

Calculates normal, biased, and unbiased auto-correlation of a vector.

Syntax

```
IppStatus ippsAutoCorrNorm_32f (const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int
dstLen, IppEnum algType, Ipp8u* pBuffer);
```

```
IppStatus ippsAutoCorrNorm_64f (const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int
dstLen, IppEnum algType, Ipp8u* pBuffer);
```

```
IppStatus ippsAutoCorrNorm_32fc (const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst, int
dstLen, IppEnum algType, Ipp8u* pBuffer);
```

```
IppStatus ippsAutoCorrNorm_64fc (const Ipp64fc* pSrc, int srcLen, Ipp64fc* pDst, int
dstLen, IppEnum algType, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Parameters

pSrc	Pointer to the source vector.
srcLen	Number of elements in the source vector.
pDst	Pointer to the destination vector. This vector stores the calculated auto-correlation of the source vector.
dstLen	Number of elements in the destination vector (length of auto-correlation).
algType	Bit-field mask for the algorithm type definition. Possible values are the results of composition of the `IppAlgType` and `IppsNormOp` values.
pBuffer	Pointer to the buffer for internal calculations.

Description

Before using these functions, you need to compute the size of the work buffer using the `ippsAutoCorrNormGetBufferSize` function.

These functions calculate the normalized auto-correlation of the *pSrc* vector of *srcLen* length and store the results in the *pDst* vector of *dstLen* length. The result vector *pDst* is calculated 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

$$pSrc[i] = \begin{cases} pSrc[i], & 0 \le i < srcLen \\ 0, & otherwise \end{cases}$$

NOTE
The auto-correlation is computed for positive lags only. Auto-correlation for a negative lag is a complex conjugate of the auto-correlation for the equivalent positive lag.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *`srcLen`* or *`dstLen`* is less than, or equal to zero.
`ippStsAlgTypeErr`	Indicates an error when: • the result of the bitwise AND operation between the `algType` and `ippAlgMask` differs from the `ippAlgAuto`, `ippAlgDirect`, or `ippAlgFFT` values; • the result of the bitwise AND operation between the `algType` and `ippsNormMask` differs from the `ippsNormNone`, `ippsNormA`, or `ippsNormB` values.

Example

The code example below demonstrates how to use the `ippsAutoCorrNormGetBufferSize` and `ippsAutoCorrNorm` functions.

```
IppStatus AutoCorrNormExample (void) {
   IppStatus status;
   const int srcLen = 5, dstLen = 10;
   Ipp32f pSrc[srcLen] = {0.2f, 3.1f, 2.0f, 1.2f, -1.1f}, pDst[dstLen];
   IppEnum funCfg = (IppEnum)(ippAlgAuto|ippsNormB);
   int bufSize = 0;
   Ipp8u *pBuffer;

   status = ippsAutoCorrNormGetBufferSize(srcLen, dstLen, ipp32f, funCfg, &bufSize);

   if ( status != ippStsNoErr )
      return status;

   pBuffer = ippsMalloc_8u( bufSize );

   status = ippsAutoCorrNorm_32f(pSrc, srcLen, pDst, dstLen, funCfg, pBuffer);

   printf_32f("pDst", pDst, dstLen);

   ippsFree( pBuffer );
   return status;
}
```

The result is as follows:

```
pDst -> 3.3 2.0 0.6 -1.6 -0.2 0.0 0.0 0.0 0.0 0.0
```

See Also

Enumerators
Special Arguments
AutoCorrNormGetBufferSize Computes the size of the work buffer for the `ippsAutoCorrNorm` function.

CrossCorrNormGetBufferSize

Computes the size of the work buffer for the `ippsCrossCorrNorm` *function.*

Syntax

```
IppStatus ippsCrossCorrNormGetBufferSize (int src1Len, int src2Len, int dstLen, int
lowLag, IppDataType dataType, IppEnum algType, int* pBufferSize);
```

Include Files

`ipps.h`

Parameters

src1Len	Number of elements in the first source vector.
src2Len	Number of elements in the second source vector.
dstLen	Number of elements in the destination vector (length of cross-correlation).
lowLag	Lower value of the range of lags at which the correlation is computed.
dataType	Data type for cross-correlation. Possible values are `ipp32f`, `ipp32fc`, `ipp64f`, or `ipp64fc`.
algType	Bit-field mask for the algorithm type definition. Possible values are the results of composition of the `IppAlgType` and `IppsNormOp` values.
pBufferSize	Pointer to the size of the work buffer.

Description

The `ippsCrossCorrNormGetBufferSize` function computes the size in bytes of the external work buffer needed for the function that performs cross-correlation. The result is stored in the *pBufferSize* parameter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pBufferSize* is `NULL`.
`ippStsSizeErr`	Indicates an error when the length of the vector is negative, or equal to zero.
`ippStsAlgTypeErr`	Indicates an error when:

	• the result of the bitwise AND operation between the `algType` and `ippAlgMask` values differs from the `ippAlgAuto`, `ippAlgDirect`, or `ippAlgFFT` values. • the result of the bitwise AND operation between the `algType` and `ippsNormMask` values differs from the `ippsNormNone`, `ippsNormA`, or `ippsNormB` values.
`ippStsDataTypeErr`	Indicates an error when the *`dataType`* value differs from the `Ipp32f`, `Ipp32fc`, `Ipp64f`, or `Ipp64fc`.

See Also

Enumerators
CrossCorrNorm Calculates the cross-correlation of two vectors.
Special Arguments

CrossCorrNorm

Calculates the cross-correlation of two vectors.

Syntax

```
IppStatus ippsCrossCorrNorm_32f (const Ipp32f* pSrc1, int src1Len, const Ipp32f* pSrc2,
int src2Len, Ipp32f* pDst, int dstLen, int lowLag, IppEnum algType, Ipp8u* pBuffer);
```

```
IppStatus ippsCrossCorrNorm_64f (const Ipp64f* pSrc1, int src1Len, const Ipp64f* pSrc2,
int src2Len, Ipp64f* pDst, int dstLen, int lowLag, IppEnum algType, Ipp8u* pBuffer);
```

```
IppStatus ippsCrossCorrNorm_32fc (const Ipp32fc* pSrc1, int src1Len, const Ipp32fc*
pSrc2, int src2Len, Ipp32fc* pDst, int dstLen, int lowLag, IppEnum algType, Ipp8u*
pBuffer);
```

```
IppStatus ippsCrossCorrNorm_64fc (const Ipp64fc* pSrc1, int src1Len, const Ipp64fc*
pSrc2, int src2Len, Ipp64fc* pDst, int dstLen, int lowLag, IppEnum algType, Ipp8u*
pBuffer);
```

Include Files

`ipps.h`

Parameters

`pSrc1`	Pointer to the first source vector.
`src1Len`	Number of elements in the first source vector.
`pSrc2`	Pointer to the second source vector.
`src2Len`	Number of elements in the second source vector.
`pDst`	Pointer to the destination vector. This vector stores the calculated cross-correlation of the *`pSrc1`* and *`pSrc2`* vectors.
`dstLen`	Number of elements in the destination vector. This value determines the range of lags at which the cross-correlation is calculated.
`lowLag`	Cross-correlation lowest lag.

algType	Bit-field mask for the algorithm type definition. Possible values are the results of composition of the `IppAlgType` and `IppsNormOp` values.
pBuffer	Pointer to the buffer for internal calculations.

Description

These functions calculate the cross-correlation of the *pSrc1* vector and the *pSrc2* vector, and store the results in the *pDst* vector. The result vector *pDst* is calculated by the following equations:

$$pDst[n] = \sum_{i=0}^{len1-1} conj(pSrc1[i]) \cdot pSrc2[n+i+lowLag] \ ,$$

where

$$0 \le n < dstLen,$$

$$pSrc2[j] = \begin{cases} pSrc2[j], & 0 < j < len2 \\ 0, & otherwise \end{cases}$$

Before using this function, you need to compute the size of the work buffer using the `ippsCrossCorrNormGetBufferSize` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when the length of a vector is less than, or equal to zero.
`ippStsAlgTypeErr`	Indicates an error when: • the result of the bitwise AND operation between the `algType` and `ippAlgMask` differs from the `ippAlgAuto`, `ippAlgDirect`, or `ippAlgFFT` values. • the result of the bitwise AND operation between the `algType` and `ippsNormMask` differs from the `ippsNormNone`, `ippsNormA`, or `ippsNormB` values.

Example

The code example below demonstrates how to use the `ippsCrossCorrNormGetBufferSize` and `ippsCrossCorrNorm` functions.

```
IppStatus CrossCorrNormExample (void) {
   IppStatus status;
   const int src1Len=5, src2Len=7, dstLen=16;
   int lowLag = -5;
   Ipp32f pSrc1[src1Len] = {1.f,1.f,1.f,1.f,1.f}, pSrc2[src2Len] = {1.f,1.f,1.f,1.f,1.f,1.f,1.f},
pDst[dstLen];
   IppEnum funCfgNormNo = (IppEnum)(ippAlgAuto|ippsNormNone);
   IppEnum funCfgNormA = (IppEnum)(ippAlgAuto|ippsNormA);
   IppEnum funCfgNormB = (IppEnum)(ippAlgAuto|ippsNormB);
   int bufSizeNo=0, bufSizeA=0, bufSizeB=0, bufSizeMax=0;
   Ipp8u *pBuffer;

   status = ippsCrossCorrNormGetBufferSize(src1Len, src2Len, dstLen, -5, ipp32f, funCfgNormNo,
&bufSizeNo);
   if ( status != ippStsNoErr ) return status;
   status = ippsCrossCorrNormGetBufferSize(src1Len, src2Len, dstLen, -5, ipp32f, funCfgNormA,
&bufSizeA);
   if ( status != ippStsNoErr ) return status;
   status = ippsCrossCorrNormGetBufferSize(src1Len, src2Len, dstLen, -5, ipp32f, funCfgNormB,
&bufSizeB);
   if ( status != ippStsNoErr ) return status;

   bufSizeMax = IPP_MAX(bufSizeNo, IPP_MAX(bufSizeA, bufSizeB));// get max buffer size

   pBuffer = ippsMalloc_8u( bufSizeMax );

   status = ippsCrossCorrNorm_32f(pSrc1, src1Len, pSrc2, src2Len, pDst, dstLen, lowLag,
funCfgNormNo, pBuffer);
   printf_32f("pDst_NormNone", pDst, dstLen);

   status = ippsCrossCorrNorm_32f(pSrc1, src1Len, pSrc2, src2Len, pDst, dstLen, lowLag,
funCfgNormA, pBuffer);
   printf_32f("pDst_NormA", pDst, dstLen);

   status = ippsCrossCorrNorm_32f(pSrc1, src1Len, pSrc2, src2Len, pDst, dstLen, lowLag,
funCfgNormB, pBuffer);
   printf_32f("pDst_NormB", pDst, dstLen);

   ippsFree( pBuffer );
   return status;
}
```

The result is as follows:

```
pDst_NormNone -> 0.0 1.0 2.0 3.0 4.0 5.0 5.0 5.0 4.0 3.0 2.0 1.0 0.0 0.0 0.0 0.0
pDst_NormA    -> 0.0 0.2 0.4 0.6 0.8 1.0 1.0 1.0 0.8 0.6 0.4 0.2 0.0 0.0 0.0 0.0
pDst_NormB    -> 0.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0
```

See Also

Enumerators

Special Arguments

CrossCorrNormGetBufferSize Computes the size of the work buffer for the `ippsCrossCorrNorm` function.

ConvolveGetBufferSize

Computes the size of the work buffer for the `ippsConvolve` *function.*

Syntax

```
IppStatus ippsConvolveGetBufferSize (int src1Len, int src2Len, IppDataType dataType,
IppEnum algType, int* pBufferSize);
```

Include Files

`ipps.h`

Parameters

src1Len	Number of elements in the first source vector.
src2Len	Number of elements in the second source vector.
dataType	Data type for convolution. Possible values are `ipp32f` and `ipp64f`.
algType	Bit-field mask for the algorithm type definition. Possible values are listed in the `IppAlgType` enumerator.
pBufferSize	Pointer to the size of the work buffer.

Description

The `ippsConvolveGetBufferSize` function computes the size, in bytes, of the external work buffer needed for the functions that perform convolution operations. The result is stored in the *pBufferSize* parameter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pBufferSize* is `NULL`.
`ippStsSizeErr`	Indicates an error when the length of the vector is negative, or equal to zero.
`ippStsAlgTypeErr`	Indicates an error when the result of the bitwise AND operation between the `algType` and `ippAlgMask` differs from the `ippAlgAuto`, `ippAlgDirect`, or `ippAlgFFT` values.
`ippStsDataTypeErr`	Indicates an error when the *dataType* value differs from the `ipp32f` or `ipp64f`.

See Also

Enumerators
Special Arguments
Convolve Performs a finite linear convolution of two vectors.

Convolve

Performs a finite linear convolution of two vectors.

Syntax

```
IppStatus ippsConvolve_32f (const Ipp32f* pSrc1, int src1Len, const Ipp32f* pSrc2, int src2Len, Ipp32f* pDst, IppEnum algType, Ipp8u* pBuffer);
```

```
IppStatus ippsConvolve_64f (const Ipp64f* pSrc1, int src1Len, const Ipp64f* pSrc2, int src2Len, Ipp64f* pDst, IppEnum algType, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Parameters

pSrc1	Pointer to the first source vector.
src1Len	Number of elements in the first source vector.
pSrc2	Pointer to the second source vector.
src2Len	Number of elements in the second source vector.
pDst	Pointer to the destination vector. This vector stores the result of the convolution of the *pSrc1* and *pSrc2* vectors.
algType	Bit-field mask for the algorithm type definition. Possible values are listed in the `IppAlgType` enumerator.
pBuffer	Pointer to the buffer for internal calculations.

Description

These functions perform the finite linear convolution of the *pSrc1* and *pSrc2* vectors. The *src1Len* elements of the *pSrc1* vector are convolved with the *src2Len* elements of the *pSrc2* vector. The result of the convolution is stored in the *pDst* vector with the length equal to *src1Len*+*src2Len*-1. The result vector *pDst* is calculated by the following equations:

$$pDst[n] = \sum_{k=0}^{n} pSrc1[k] \cdot pSrc2[n-k] \qquad 0 \le n \le src1Len + src2Len - 1$$

where

- *pSrc1*[i]=0, if i≥*src1Len*
- *pSrc2*[j]=0, if j≥*src2Len*

Before using this function, you need to compute the size of the work buffer using the `ippsConvolveGetBufferSize` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.

ippStsSizeErr	Indicates an error when the length of a vector is less than, or equal to zero.
ippStsAlgTypeErr	Indicates an error when the result of the bitwise AND operation between algType and ippAlgMask differs from the ippAlgAuto, ippAlgDirect, or ippAlgFFT values.

Example

The code example below demonstrates how to use the ippsConvolveGetBufferSize and ippsConvolve_32f functions.

```
IppStatus ConvolveExample (void) {
   IppStatus status;
   const int src1Len = 5, src2Len = 2, dstLen = src1Len+src2Len-1;
   Ipp32f pSrc1[src1Len] = {-2.f,0.f,1.f,-1.f,3.f}, pSrc2[src2Len]={0.f,1.f}, pDst[dstLen];
   IppEnum funCfg = (IppEnum)(ippAlgAuto);
   int bufSize = 0;
   Ipp8u *pBuffer;

   status = ippsConvolveGetBufferSize(src1Len, src2Len, ipp32f, funCfg, &bufSize);

   if ( status != ippStsNoErr )
      return status;

   pBuffer = ippsMalloc_8u( bufSize );

   status = ippsConvolve_32f(pSrc1, src1Len, pSrc2, src2Len, pDst, funCfg, pBuffer);

   printf_32f("pDst", pDst, dstLen);

   ippsFree( pBuffer );
   return status;
}
```

The result is as follows:

```
pDst -> 0.0 -2.0 0.0 1.0 -1.0 3.0
```

See Also

Enumerators
Special Arguments
ConvolveGetBufferSize Computes the size of the work buffer for the ippsConvolve function.

ConvBiased

Computes the specified number of elements of the full finite linear convolution of two vectors.

Syntax

IppStatus ippsConvBiased_32f(const Ipp32f* *pSrc1*, int *src1Len*, const Ipp32f* *pSrc2*, int *src2Len*, Ipp32f* pDst, int *dstLen*, int *bias*);

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc1, *pSrc2*	Pointers to the two vectors to be convolved.
src1Len	Number of elements in the vector *pSrc1*.
src2Len	Number of elements in the vector *pSrc2*.
pDst	Pointer to the vector *pDst*. This vector stores the result of the convolution.
dstLen	Number of elements in the vector *pDst*.
bias	Parameter that specifes the starting element of the convolution.

Description

This function computes *dstLen* elements of finite linear convolution of two specified vectors *pSrc1* and *pSrc2* starting with an element that is specified by the *bias*. The result is stored in the vector *pDst*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *src1Len* or *src2Len* is less than or equal to 0.

Example

The example below shows how to call the function ippsConvBiased.

```
void func_convbiased()
{
    Ipp32f pSrc1[5] = {1.1, -2.0, 3.5, 2.2, 0.0};
    Ipp32f pSrc2[4] = {0.0, 0.2, 2.5, -1.0};
    const int len = 10;
    Ipp32f pDst[len];
    int bias = 1;

    ippsZero_32f(pDst, len);
    ippsConvBiased_32f(pSrc1, 5, &pSrc2[1], 3, pDst, len, bias);
}
```

Result:

```
        pDst -> 0.2  2.3  -4.3  9.2  5.5  0.0  0.0  0.0  0.0  0.0
```

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.

SumWindow

Sums elements in the mask applied to each element of a vector.

Syntax

```
IppStatus ippsSumWindow_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len, int maskSize);
IppStatus ippsSumWindow_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len, int maskSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements of the vector.
maskSize	Size of the mask.

Description

This function sets each element in the destination vector *pDst* as the sum of *maskSize* elements of the source vector *pSrc*. The computation is performed as follows:

$$pDst[n] = \sum_{k=n}^{maskSize} pSrc[k] \ , 0 \le n < len$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsMaskSizeErr`	Indicates an error when *maskSize* is less than or equal to 0.

FIR Filter Functions

The functions described in this section perform a finite impulse response (FIR) filtering of input data. The functions initialize different FIR filter structures, get and set the delay lines and filter coefficients (taps), and perform filtering. Intel IPP contains the functions that implement the FIR filters without the delay line - stream FIR filters.

Special set of functions allows to compute the filter coefficients for different filters.

To perform single-rate FIR filtering with the `ippsFIRSR` function, follow this scheme:

1. Call `ippsFIRSRGetSize` function to get the size of the filter specification structure and the work buffer.
2. Call `ippsFIRSRInit` function to initialize the filter specification structure.
3. Call `ippsFIRSR` function to apply the single-rate FIR filter to a source vector.

FIRMRGetSize

Computes the size of the context structure and work buffer for multi-rate FIR filtering.

Syntax

```
IppStatus ippsFIRMRGetSize(int tapsLen, int upFactor, int downFactor, IppDataType
tapsType, int* pSpecSize, int* pBufSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

tapsLen	Length of the FIR filter.
upFactor	Multi-rate up factor.
downFactor	Multi-rate down factor.
tapsType	Data type of the coefficients. Supported values are `Ipp32f` and `Ipp64f`.
pSpecSize	Pointer to the size of the FIR specification structure.
pBufSize	Pointer to the size of the temporary buffer required for FIR filtering.

Description

This function computes the following:

- Size of the internal specification structure for multi-rate FIR filtering. The structure can be shared between all threads of the application.
- Size of the work buffer for each thread.

For an example on how to use this function, refer to the example provided with the `FIRMR` function description.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.

See Also

FIRMR Performs multi-rate FIR filtering of a source vector.

FIRMRInit

Initializes the multi-rate FIR filter specification structure.

Syntax

```
IppStatus ippsFIRMRInit_<mod>(const Ipp<dataType>* pTaps, int tapsLen, int upFactor,
int upPhase, int downFactor, int downPhase, IppsFIRSpec_<dataType>* pSpec);
```

Supported values for `mod`:

32f	64f	32fc	64fc

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pTaps	Pointer to the array containing filter coefficients. The number of elements in the array is *tapsLen*.
tapsLen	Number of filter coefficients.
upFactor	Multi-rate upsampling factor.
upPhase	Phase for upsampled signal.
downFactor	Multi-rate downsampling factor.
downPhase	Phase for downsampled signal.
pSpec	Pointer to the FIR specification structure.

Description

This function initializes the multi-rate FIR filter specification structure in the external buffer. Before using this function, compute the size of the specification structure and the size of the work buffer using the `FIRMRGetSize` function.

The parameter *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the function SampleUp for more details). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The parameter *upPhase* is the parameter, which determines where a non-zero sample lies within the *upFactor*-length block of the upsampled input signal.

The parameter *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the function SampleDown for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

The*downPhase* parameter determines where non-discarded sample lies within a block of upsampled filter response.

For an example on how to use this function, refer to the example provided with the FIRMR function description.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than, or equal to zero.
ippStsFIRMRFactorErr	Indicates an error when *upFactor* or*downFactor* is less than, or equal to zero.
ippStsFIRMRPhaseErr	Indicates an error when *upPhase*/*downPhase* is negative, or greater than or equal to *upFactor*/*downFactor*.

See Also

FIRMRGetSize Computes the size of the context structure and work buffer for multi-rate FIR filtering.

FIRMR Performs multi-rate FIR filtering of a source vector.

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.

FIRMR

Performs multi-rate FIR filtering of a source vector.

Syntax

```
IppStatus ippsFIRMR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
IppsFIRSpec_32f* pSpec, const Ipp32f* pDlySrc, Ipp32f* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRMR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
IppsFIRSpec_64f* pSpec, const Ipp64f* pDlySrc, Ipp64f* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRMR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
IppsFIRSpec_32fc* pSpec, const Ipp32fc* pDlySrc, Ipp32fc* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRMR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int numIters,
IppsFIRSpec_64fc* pSpec, const Ipp64fc* pDlySrc, Ipp64fc* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRMR_16s(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
IppsFIRSpec_32f* pSpec, const Ipp16s* pDlySrc, Ipp16s* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRMR_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int numIters,
IppsFIRSpec_32fc* pSpec, const Ipp16sc* pDlySrc, Ipp16sc* pDlyDst, Ipp8u* pBuf);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
numIters	Number of elements in the destination vector.
pSpec	Pointer to the internal specification structure.
pDlySrc	Pointer to the array containing values for the source delay lines. The value can be NULL.
pDlyDst	Pointer to the array containing values for the destination delay line. The value can be NULL.
pBuf	Pointer to the work buffer.

Description

Before using this function, you need to initialize the internal constant specification structure using the ippsFIRMR_Init function.

This function filters the source vector using the multi-rate FIR filter. Filtering is performed by the following formula:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters, \quad 0 \le i < tapsLen - 1$$

where

- x(0)...x(*numIters*) is the source vector
- h(0)...h(*tapsLen*-1) are the FIR filter coefficients

To compute the y(0)...y(*tapsLen*-1) destination vector, the function uses the *pDlySrc* array of the delay line. The length of the *pDlySrc* array is *tapsLen*-1 elements.

The first *tapsLen*-1 elements of the function are:

y(0)=h(*tapsLen*-1)*d(0)+h(*tapsLen*-2)*d(1)+...+h(1)*d(*tapsLen*-2)+h(0)*x(0)

y(0)=h(*tapsLen*-1)*d(1)+h(*tapsLen*-2)*d(2)+...+h(1)*x(0)+h(0)*x(1)

y(*tapsLen*-1)=h(*tapsLen*-1)*x(0)+...+h(1)*x(*tapsLen*-2)+h(0)*x(*tapsLen*-1)

where

d(0), d(1), d(2), and d(*tapsLen*-2) are the elements of the *pDlySrc* array

The last *tapsLen*-1 elements of the source vector are copied to the non-zero *pDlyDst* buffer for the next call of the FIR filter.

The arrays *pDlySrc* and *pDlyDst* support NULL values:

- if *pDlySrc* is NULL, the function uses the delay line with zero values
- if *pDlyDst* is NULL, the function does not copy any data to the destination delay line

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pTaps* or *pSpec* is NULL.

ippStsSizeErr	Indicates an error when the *tapsLen* value is less than, or equal to zero.

Example

The code example below demonstrates how to use the FIRMRGetSize, ippsFIRMR_Init, and ippsFIRMR functions.

```
int firmr()
{
    IppsFIRSpec_32f *pSpec;
    Ipp32f pTaps[8] = { 0.125,0.125,0.125,0.125,0.125,0.125,0.125,0.125};
    int i;
    int numIters = 33;
    int tapsLen = 8;
    int upFactor = 2;
    int upPhase = 0;
    int downFactor = 3;
    int downPhase = 0;
    int specSize, bufSize;
    Ipp32f pSrc[downFactor*numIters];
    Ipp32f pDst[upFactor *numIters];
    Ipp32f pDlySrc[(tapsLen + upFactor - 1  ) / upFactor];
    Ipp32f pDlyDst[(tapsLen + upFactor - 1  ) / upFactor];
    Ipp8u* pBuf;
    IppStatus status;
    status = ippsFIRMRGetSize(tapsLen, upFactor, downFactor, ipp32f,  &specSize,  &bufSize );
    printf("ippsFIRMRGetSize / status = %s\n", ippGetStatusString(status));
    pSpec = (IppsFIRSpec_32f*)ippsMalloc_8u(specSize);
    pBuf  = ippsMalloc_8u(bufSize);
    status = ippsFIRMR_Init_32f ( pTaps, tapsLen, upFactor, upPhase, downFactor, downPhase, pSpec);
    printf("ippsFIRMR_Init_32f / status = %s\n", ippGetStatusString(status));
    if( ippStsNoErr != status){
        return -1;
    }
    for(i=0;i<downFactor*numIters;i++){
        pSrc[i] = 1;
    }
    status = ippsFIRMR_32f( pSrc, pDst, numIters, pSpec, NULL, pDlyDst, pBuf);
    printf("ippsFIRMR_32f / status = %s\n", ippGetStatusString(status));
    if( ippStsNoErr != status){
        return -1;
    }
    printf("src\n");
    for(i=0;i<numIters*downFactor;i++){
        printf("%6f ", pSrc[+i]);
    }
    printf("\ndst\n");
    for(i=0;i<numIters*upFactor;i++){
        printf("%6f ", pDst[i]);
    }
    printf("\n");
    return 0;
}
```

See Also

FIRMRGetSize Computes the size of the context structure and work buffer for multi-rate FIR filtering.

FIRMRInit Initializes the multi-rate FIR filter specification structure.

FIRSRGetSize

Computes the size of the constant structure and work buffer for single-rate FIR filtering.

Syntax

```
IppStatus ippsFIRSRGetSize(int tapsLen, IppDataType tapsType, int* pSpecSize, int*
pBufSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

tapsLen	Length of the FIR filter.
tapsType	Data type of the coefficients. Support values: `ipp32f` or `ipp64f`.
pSpecSize	Pointer to the size of the internal constant specification structure.
pBufSize	Pointer to the size of the work buffer required for FIR filtering.

Description

This function computes the following:

- Size of the internal constant specification structure for single-rate FIR filtering. The structure can be shared between all threads of the application.
- Size of the work buffer for each thread.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when the *tapsLen* value is less than, or equal to zero.
`ippStsAlgTypeErr`	Indicates an error when the specified algorithm type is not supported.

See Also

Examples of Using FIR Functions

FIRSRInit

Initializes the constant structure for single-rate FIR filtering.

Syntax

```
IppStatus ippsFIRSRInit_32f(const Ipp32f* pTaps, int tapsLen, IppAlgType algType,
IppsFIRSpec_32f* pSpec);
```

```
IppStatus ippsFIRSRInit_64f(const Ipp64f* pTaps, int tapsLen, IppAlgType algType,
IppsFIRSpec_64f* pSpec);
```

```
IppStatus ippsFIRSRInit_32fc(const Ipp32fc* pTaps, int tapsLen, IppAlgType algType,
IppsFIRSpec_32fc* pSpec);
```

```
IppStatus ippsFIRSRInit_64fc(const Ipp64fc* pTaps, int tapsLen, IppAlgType algType,
IppsFIRSpec_64fc* pSpec);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pTaps	Pointer to the filter coefficients.
tapsLen	Length of the FIR filter.
algType	Bit-field mask for the algorithm type definition. Possible values are: `ippAlgAuto`, `ippAlgDirect`, or `ippAlgFFT`.
pSpec	Pointer to the internal constant specification structure.

Description

Before using this function, you need to compute the size of the specification structure using the `ippsFIRSRGetSize` function. This function initializes the constant specification structure for single-rate FIR filtering.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when the *tapsLen* value is less than, or equal to zero.
`ippStsAlgTypeErr`	Indicates an error when the specified algorithm type is not supported.
`ippStsMismatch`	Indicates an error when the selected algorithm type for the pair of *tapsLen* and *numIters* values is not effective.

See Also

FIRSRGetSize Computes the size of the constant structure and work buffer for single-rate FIR filtering.

Examples of Using FIR Functions

FIRSR

Performs single-rate FIR filtering of a source vector.

Syntax

```
IppStatus ippsFIRSR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
IppsFIRSpec_32f* pSpec, const Ipp32f* pDlySrc, Ipp32f* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRSR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
IppsFIRSpec_64f* pSpec, const Ipp64f* pDlySrc, Ipp64f* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRSR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
IppsFIRSpec_32fc* pSpec, const Ipp32fc* pDlySrc, Ipp32fc* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRSR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int numIters,
IppsFIRSpec_64fc* pSpec, const Ipp64fc* pDlySrc, Ipp64fc* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRSR_16s(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
IppsFIRSpec_32f* pSpec, const Ipp16s* pDlySrc, Ipp16s* pDlyDst, Ipp8u* pBuf);
```

```
IppStatus ippsFIRSR_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int numIters,
IppsFIRSpec_32fc* pSpec, const Ipp16sc* pDlySrc, Ipp16sc* pDlyDst, Ipp8u* pBuf);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
numIters	Number of elements in the destination vector.
pSpec	Pointer to the internal constant specification structure.
pDlySrc	Pointer to the array containing values for the source delay lines.
pDlyDst	Pointer to the array containing values for the destination delay line.
pBuf	Pointer to the work buffer.

Description

Before using this function, you need to initialize the internal constant specification structure using the `ippsFIRSRInit` function.

This function filters the source vector using the single-rate FIR filter. Filtering is performed by the following formula:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad \begin{array}{l} 0 \le n < numIters \\ 0 \le i < tapsLen - 1 \end{array}$$

where

- `x(0)...x(`*numIters*`)` is the source vector
- `h(0)...h(`*tapsLen*`-1)` are the FIR filter coefficients

To compute the `y(0)...y(`*tapsLen*`-1)` destination vector, the function uses the *pDlySrc* array of the delay line. The length of the *pDlySrc* array is *tapsLen*-1 elements.

The first *tapsLen*-1 elements of the function are:

y(0)=h(*tapsLen*-1)*d(0)+h(*tapsLen*-2)*d(1)+...+h(1)*d(*tapsLen*-2)+h(0)*x(0)

y(0)=h(*tapsLen*-1)*d(1)+h(*tapsLen*-2)*d(2)+...+h(1)*x(0)+h(0)*x(1)

y(*tapsLen*-1)=h(*tapsLen*-1)*x(0)+...+h(1)*x(*tapsLen*-2)+h(0)*x(*tapsLen*-1)

where

d(0), d(1), d(2), and d(*tapsLen*-2) are the elements of the *pDlySrc* array

The last *tapsLen*-1 elements of the source vector are copied to the non-zero *pDlyDst* buffer for the next call of the FIR filter.

The arrays *pDlySrc* and *pDlyDst* support NULL values:

- if *pDlySrc* is NULL, the function uses the delay line with zero values
- if *pDlyDst* is NULL, the function does not copy any data to the destination delay line

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pTaps* or *pSpec* is NULL.
ippStsSizeErr	Indicates an error when the *tapsLen* value is less than, or equal to zero.
ippStsAlgTypeErr	Indicates an error when the specified algorithm type is not supported.
ippStsMismatch	Indicates an error when the selected algorithm type for the pair of *tapsLen* and *numIters* values is not effective.

See Also

FIRSRInit Initializes the constant structure for single-rate FIR filtering.
Examples of Using FIR Functions

FIRSparseInit

Initializes a sparse FIR filter structure.

Syntax

```
IppStatus ippsFIRSparseInit_32f(IppsFIRSparseState_32f** ppState, const Ipp32f*
pNZTaps, const Ipp32s* pNZTapPos, int nzTapsLen, const Ipp32f* pDlyLine, Ipp8u*
pBuffer);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pNZTaps	Pointer to the array containing the non-zero tap values. The number of elements in the array is *nzTapsLen*.
pNZTapPos	Pointer to the array containing positions of the non-zero tap values. The number of elements in the array is *nzTapsLen*.

nzTapsLen	Number of elements in the array with non-zero tap values.
pDlyLine	Pointer to the array containing the delay line values.
ppState	Double pointer to the sparse FIR state structure.
pBuffer	Pointer to the external buffer for the sparse FIR state structure.

Description

This function initializes a sparse FIR filter state structure *ppState* in the external buffer *pBuffer*. The size of this buffer must be computed previously by calling the function FIRSparseGetStateSize. The initialization function copies the values of filter coefficients from the array *pNZTaps* containing *nzTapsLen* non-zero taps and their positions from the array *pNZTapPos* into the state structure *ppState*. The array *pDlyLine* specifies the delay line values. The number of elements in this array is *pNZTapPos*[*nzTapsLen* -1]. If the pointer to the array *pDlyLine* is not NULL, the array contents are copied into the state structure *ppState*, otherwise the delay line values in the state structure are initialized to 0.

NOTE
The values of *nzTapsLen* and *pNZTapPos*[*nzTapsLen* -1] must be equal to those specified in the function FIRSparseGetStateSize.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers *ppState*, *pNZTaps*, *pNZTapPos*, or *pBuffer* is NULL.
ippStsFIRLenErr	Indicates an error if *nxTapsLen* is less than or equal to 0.
ippStsSparseErr	Indicates an error if positions of the non-zero taps are not in ascending order, or are negative or repetitive.

FIRSparseGetStateSize

Computes the size of the external buffer for the sparse FIR filter structure.

Syntax

```
IppStatus ippsFIRSparseGetStateSize_32f(int nzTapsLen, int order, int* pStateSize);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

nzTapsLen	Number of elements in the array containing the non-zero tap values.
order	Order of the sparse FIR filter.
pStateSize	Pointer to the computed value of the external buffer.

Description

This function computes the size of the external buffer for a sparse FIR filter structure that is required for the function ippsFIRSparseInit . Computation is based on the specified number of non-zero filter coefficients *nzTapsLen* and filter order *order* that is equal to the number of elements in the delay line *pNZTapPos* [*nzTapsLen* -1] (see description of the function ippsFIRSparseInit). The result value is stored in the *pStateSize*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pStateSize* pointer is NULL.
ippStsFIRLenErr	Indicates an error if *nzTapsLen* or *order* is less than or equal to 0; or *nzTapsLen* is more than *order*.

FIRSparseGetDlyLine

Retrieves the delay line contents from the sparse FIR filter state structure.

Syntax

```
IppStatus ippsFIRSparseGetDlyLine_32f(const IppsFIRSparseState_32f* pState, Ipp32f*
pDlyLine);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pState	Pointer to the sparse FIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values.

Description

This function copies the delay line values from the state structure *pState* and stores them into *pDlyLine*. The destination array *pDlyLine* contains samples in the reverse order as compared to the order of samples in the source vector.

Before calling ippsFIRSparseGetDlyLine, the corresponding filter state structure must be initialized with the FIRSparseInit function.

Return Values

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

See Also

FIRSparseInit Initializes a sparse FIR filter structure.

FIRSparseSetDlyLine

Sets the delay line contents in the sparse FIR filter state structure.

Syntax

```
IppStatus ippsFIRSparseSetDlyLine_32f(IppsFIRSparseState_32f* pState, const Ipp32f*
pDlyLine);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the FIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values.

Description

This function copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. The source array *pDlyLine* must contain samples in the reverse order as compared to the order of samples in the source vector.

Before calling `ippsFIRSparseSetDlyLine`, the corresponding filter state structure must be initialized with the `FIRSparseInit` function.

Return Values

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

See Also

FIRSparseInit Initializes a sparse FIR filter structure.

FIRSparse

Filters a source vector through a sparse FIR filter.

Syntax

```
IppStatus ippsFIRSparse_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
IppsFIRSparseState_32f* pState);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the sparse FIR filter state structure.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements that are filtered.

Description

This function applies the sparse FIR filter to the *len* elements of the source vector *pSrc*, and stores the results in *pDst*. The filter parameters - the number of non-zero taps *nzTapsLen*, their values *pNZTaps* and their positions *pNZTapPos*, and the delay line values *pDlyLine* - are specified in the sparse FIR filter structure *pState* that should be previously initialized by calling the function ippsFIRSparseInit.

In the following definition of the sparse FIR filter, the sample to be filtered is denoted *x*(n), the non-zero taps are denoted *pNZTaps*(i), their positions are denoted *pNZTapPos*(i) and the return value is *y*(n).

The return value *y*(n) is defined by the formula for a sparse FIR filter:

$$y(n) = \sum_{i=0}^{nzTapsLen-1} pNZTaps(i) \cdot x(n - pNZTapPos(i)), \quad 0 \le n < len$$

After the function has performed calculations, it updates the delay line values stored in the state.

Return Values

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

Example

The example below shows how to use the sparse FIR filter functions.

```
int buflen;
Ipp8u *buf;
int nzTapsLen = 5; //number of non-zero taps
Ipp32f nzTaps [] = {0.5, 0.4, 0.3, 0.2, 0.1};  //non-zero taps values
Ipp32s nzTapsPos[] = {0, 10, 20, 30, 40};  //non-zero tap positions
IppsFIRSparseState_32f* firState;
Ipp32f *src, *dst;

/* ........................... */
ippsFIRSparseGetStateSize_32f(nzTapsLen, nzTapsPos [nzTapsLen - 1], &buflen);
buf = ippsMalloc_8u(buflen);
ippsFIRSparseInit_32f(&firState, nzTaps, nzTapsPos, nzTapsLen, NULL, buf);

/* .... initializing src somehow .... */
ippsFIRSparse_32f(src, dst, len, firState);

/*dst[i]=src[i]*0.5 + src[i-10]*0.4 + src[i-20]*0.3 + src[i-30]*0.2 + src[i-40]*0.1 */
```

```
/* ........................*/
ippsFree(buf);
```

Examples of Using FIR Functions

The code examples below demonstrate how to use the `ippsFIRSR` function:

- Standard FIR Filtering with a Not-in-place Destination
- Standard FIR Filtering with an In-place Destination
- Stream FIR Filtering with a Not-in-place Destination
- Stream FIR Filtering with an In-place Destination
- Standard FIR Filtering with a Not-in-place Destination and Threading
- Standard FIR Filtering with an In-place Destination and Threading

Standard FIR Filtering with a Not-in-place Destination

Type of FIR Filter	Destination	Source Delay Line	Threading
standard	not-in-place	zero	none

```
    #define LEN 1024
    #define TAPS_LEN 8

    IppsFIRSpec_32f *pSpec;
    float           *src, *dst, *dly, *taps;
    Ipp8u           *buf;
    int             specSize, bufSize;
    IppStatus status;
    //get sizes of the spec structure and the work buffer
    status = ippsFIRSRGetSize (TAPS_LEN,  ipp32f ,  &specSize,  &bufSize );

    src   = ippsMalloc_32f(LEN);
    dst   = ippsMalloc_32f(LEN);
    dly   = ippsMalloc_32f(TAPS_LEN-1);
    taps  = ippsMalloc_32f(TAPS_LEN);
    pSpec = (IppsFIRSpec_32f*)ippsMalloc_8u(specSize);
    buf   = ippsMalloc_8u(bufSize);

    //initialize the spec structure
    ippsFIRSRInit_32f( taps, TAPS_LEN, ippAlgDirect, pSpec );
    //apply the FIR filter
    ippsFIRSR_32f(src, dst, LEN,  pSpec,  NULL, dly,  buf);
```

Standard FIR Filtering with an In-place Destination

Type of FIR Filter	Destination	Source Delay Line	Threading
standard	in-place	zero	none

```
    #define LEN 1024
    #define TAPS_LEN 8

    IppsFIRSpec_32f *pSpec;
    float           *src, *dst, *dly, *taps;
    Ipp8u           *buf;
    int             specSize, bufSize;

    //get sizes of the spec structure and the work buffer
```

```
ippsFIRSRGetSize(TAPS_LEN, ipp32f, &specSize, &bufSize );

src   = ippsMalloc_32f(LEN);
dst   = src;
dly   = ippsMalloc_32f(TAPS_LEN-1);
taps  = ippsMalloc_32f(TAPS_LEN);
pSpec = (IppsFIRSpec_32f*)ippsMalloc_8u(specSize);
buf   = ippsMalloc_8u(bufSize);

//initialize the spec structure
ippsFIRSRInit_32f( taps, TAPS_LEN, ippAlgDirect, pSpec );
//apply the FIR filter
ippsFIRSR_32f(src, dst, LEN,  pSpec,  NULL, dly,  buf);
```

Stream FIR Filtering with a Not-in-place Destination

Type of FIR Filter	Destination	Source Delay Line	Threading
stream	not-in-place	*src*	none

```
#define LEN 1024
#define TAPS_LEN 8

IppsFIRSpec_32f *pSpec;
float           *src, *dst,  *taps;
Ipp8u           *buf;
int             specSize, bufSize;

//get sizes of the spec structure and the work buffer
ippsFIRSRGetSize(TAPS_LEN, ipp32f, &specSize, &bufSize );

src   = ippsMalloc_32f(LEN+TAPS_LEN-1);
dst   = ippsMalloc_32f(LEN);
taps  = ippsMalloc_32f(TAPS_LEN);
pSpec = (IppsFIRSpec_32f*)ippsMalloc_8u(specSize);
buf   = ippsMalloc_8u(bufSize);

//initialize the spec structure
ippsFIRSRInit_32f( taps, TAPS_LEN, ippAlgDirect, pSpec );
//apply the FIR filter
ippsFIRSR_32f(src+TAPS_LEN-1, dst, LEN,  pSpec,  src, NULL,  buf);
```

Stream FIR Filtering with an In-place Destination

Type of FIR Filter	Destination	Source Delay Line	Threading
stream	in-place	*src*	none

```
#define LEN 1024
#define TAPS_LEN 8

IppsFIRSpec_32f *pSpec;
float           *src, *dst, *taps;
Ipp8u           *buf;
int             specSize, bufSize;

//get sizes of the spec structure and the work buffer
ippsFIRSRGetSize(TAPS_LEN, ipp32f, &specSize, &bufSize );

src   = ippsMalloc_32f(LEN+TAPS_LEN-1);
```

```
dst   = src;
taps  = ippsMalloc_32f(TAPS_LEN);
pSpec = (IppsFIRSpec_32f*)ippsMalloc_8u(specSize);
buf   = ippsMalloc_8u(bufSize);

//initialize the spec structure
ippsFIRSRInit_32f( taps, TAPS_LEN, ippAlgDirect, pSpec );
//apply the FIR filter
ippsFIRSR_32f(src+TAPS_LEN-1, dst, LEN,  pSpec,  src, NULL,  buf);
```

Standard FIR Filtering with a Not-in-place Destination and Threading

Type of FIR Filter	Destination	Source Delay Line	Threading
standard	not-in-place	zero	NTHREADS

```
#define LEN 1024
#define TAPS_LEN 8
#define DLY_LEN TAPS_LEN-1
#define NTH 4

float *src,*dst;
float *dlyOut, *taps;
unsigned char *buf;
IppsFIRSpec_32f* pSpec;
int specSize, bufSize;
int i,tlen, ttail;

//get sizes of the spec structure and the work buffer
ippsFIRSRGetSize(TAPS_LEN, ipp32f, &specSize, &bufSize );

src    = ippsMalloc_32f(LEN);
dst    = ippsMalloc_32f(LEN);
dlyOut = ippsMalloc_32f(TAPS_LEN-1);
taps   = ippsMalloc_32f(TAPS_LEN);
pSpec  = (IppsFIRSpec_32f*)ippsMalloc_8u(specSize);
buf    = ippsMalloc_8u(bufSize*NTH);
for(i=0;i<LEN;i++){
    src[i] = i;
}
for(i=0;i<TAPS_LEN;i++){
    taps[i] = 1;
}
//initialize the spec structure
ippsFIRSRInit_32f( taps, TAPS_LEN, ippAlgDirect, pSpec );
tlen  = LEN / NTH;
ttail = LEN % NTH;
for(i=0;i< NTH;i++) {//this cycle means parallel region
    Ipp32f* s   = src+i*tlen;
    Ipp32f* d   = dst+i*tlen;
    int     len = tlen+((i==(NTH - 1))?ttail:0);
    Ipp8u*  b   = buf+i*bufSize;
    if( i == 0)
        ippsFIRSR_32f(s, d, len, pSpec, NULL,           NULL  , b);
    else if (i == NTH - 1)
        ippsFIRSR_32f(s, d, len, pSpec, s-(TAPS_LEN-1), dlyOut, b);
    else
        ippsFIRSR_32f(s, d, len, pSpec, s-(TAPS_LEN-1), NULL  , b);
}
```

Standard FIR Filtering with an In-place Destination and Threading

Type of FIR Filter	Destination	Source Delay Line	Threading
standard	in-place	zero	NTHREADS

```
#define LEN 1024
#define TAPS_LEN 8
#define DLY_LEN TAPS_LEN-1
#define NTHREADS 4

float *src, *dst,  *dlyOut, *taps;
float* tdly[NTHREADS];
unsigned char *buf;
IppsFIRSpec_32f* pSpec;
int specSize, bufSize;
int i,tlen, ttail;

//get sizes of the spec structure and the work buffer
ippsFIRSRGetSize(TAPS_LEN, ipp32f, &specSize, &bufSize );

src    = ippsMalloc_32f(LEN);
dst    = ippsMalloc_32f(LEN);
dlyOut = ippsMalloc_32f(TAPS_LEN-1);
taps   = ippsMalloc_32f(TAPS_LEN);
pSpec  = (IppsFIRSpec_32f*)ippsMalloc_8u(specSize);
buf    = ippsMalloc_8u(bufSize*NTHREADS);

//initialize the spec structure
ippsFIRSRInit_32f( taps, TAPS_LEN, ippAlgDirect, pSpec );
tlen  = LEN / NTHREADS;
ttail = LEN % NTHREADS;
//tdly    = ippsMalloc_32f((TAPS_LEN-1)*(NTHREADS-1));

for(i=1;i<NTHREADS;i++){//cycle in main thread
    tdly[i]  = ippsMalloc_32f(TAPS_LEN-1);
    ippsCopy_32f(src+i*tlen-(TAPS_LEN-1), tdly[i], TAPS_LEN-1);
}
for(i=0;i< NTHREADS;i++) {//this cycle means parallel region
    Ipp32f* s   = src+i*tlen;
    Ipp32f* d   = dst+i*tlen;
    int     len = tlen+((i==NTHREADS - 1)?ttail:0);
    Ipp8u*  b   = buf+i*bufSize;
    if( i == 0)
        ippsFIRSR_32f(s, d, len, pSpec, NULL,    NULL  , b);
    else if (i == NTHREADS - 1)
        ippsFIRSR_32f(s, d, len, pSpec, tdly[i], dlyOut, b);
    else
        ippsFIRSR_32f(s, d, len, pSpec, tdly[i], NULL  , b);
}
```

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.

FIRGenGetBufferSize

Computes the size of the internal buffer required for computation of FIR coefficients.

Syntax

```
IppStatus ippsFIRGenGetBufferSize(int tapsLen, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

tapsLen	Number of taps.
pBufferSize	Pointer to the calculated buffer size (in bytes).

Description

This function computes the size of the buffer that is required for `ippsFIRGenBandpass`, `ippsFIRGenBandstop`, `ippsFIRGenHighpass`, and `ippsFIRGenLowpass` internal calculations.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsIIRGenOrderErr`	Indicates an error when the length of the coefficients array is less than 5.

See Also

FIRGenBandpass Computes bandpass FIR filter coefficients.
FIRGenBandstop Computes bandstop FIR filter coefficients.
FIRGenHighpass Computes highpass FIR filter coefficients.
FIRGenLowpass Computes lowpass FIR filter coefficients.

FIRGenLowpass

Computes lowpass FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenLowpass_64f(Ipp64f rFreq, Ipp64f* pTaps, int tapsLen, IppWinType
winType, IppBool doNormal, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

rFreq	Normalized cutoff frequency, must 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; must be equal or greater than 5.
winType	Specifies what type of window is 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	Specifies normalized or non-normalized sequence of the filter coefficients is computed. The *doNormal* must have one of the following values: • `ippTrue` for normalized sequence of coefficients • `ippFalse` for non-normalized sequence of coefficients
pBuffer	Pointer to the buffer for internal calculations. To get the size of the buffer, use the `ippsFIRGenGetBufferSize` function.

Description

This function computes *tapsLen* coefficients for lowpass FIR filter with the cutoff frequency *rFreq* by windowing the ideal infinite filter coefficients. The quality of filtering is defined by the number of 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*.

For more information about the used algorithm, see [MIT 93].

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTaps* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when the *tapsLen* is less than 5, or *rFreq* is out of range.

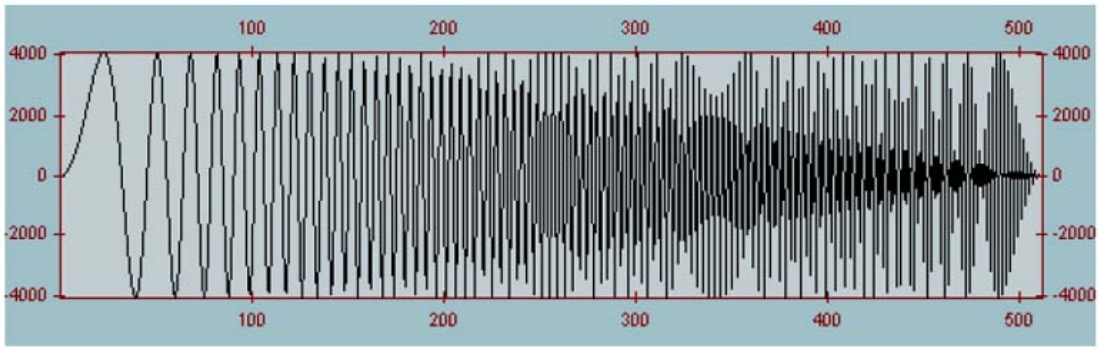
Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

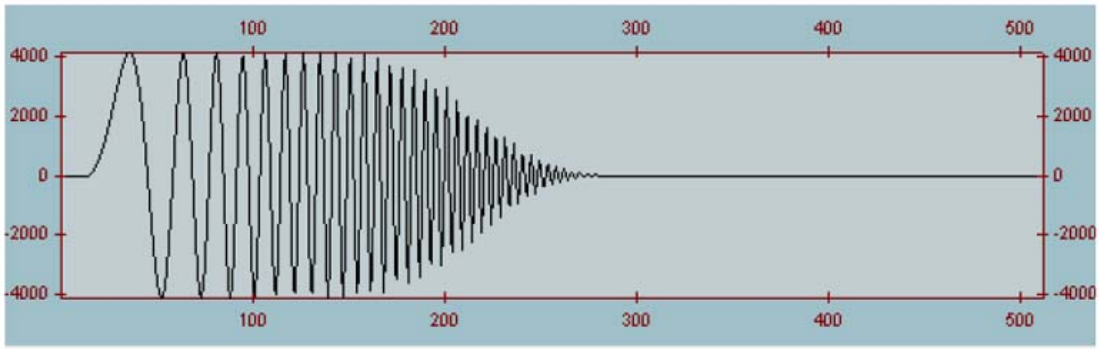
`FIRGenLowpass.c`

Result:

initial signal



filtered signal



FIRGenHighpass

Computes highpass FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenHighpass_64f(Ipp64f rFreq, Ipp64f* pTaps, int tapsLen, IppWinType
winType, IppBool doNormal, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

rFreq	Normalized cutoff frequency, must 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; must be equal or greater than 5.
winType	Specifies what type of window is 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	Specifies normalized or non-normalized sequence of the filter coefficients is 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.
pBuffer	Pointer to the buffer for internal calculations. To get the size of the buffer, use the `ippsFIRGenGetBufferSize` function.

Description

This function computes *tapsLen* coefficients for highpass FIR filter the cutoff 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.he computed coefficients are stored in the array *pTaps*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTaps* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when the *tapsLen* is less than 5, or *rFreq* is out of the range.

FIRGenBandpass

Computes bandpass FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenBandpass_64f(Ipp64f rLowFreq, Ipp64f rHighFreq, Ipp64f* pTaps, int
tapsLen, IppWinType winType, IppBool doNormal, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

rLowFreq	Normalized low cutoff frequency, must be in the range (0, 0.5) and less than *rHighFreq*.
rHighFreq	Normalized high cutoff frequency, must 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	Specifies what type of window is 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	Specifies normalized or non-normalized sequence of the filter coefficients is 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.
pBuffer	Pointer to the buffer for internal calculations. To get the size of the buffer, use the ippsFIRGenGetBufferSize function.

Description

This function computes *tapsLen* coefficients for bandpass FIR filter with the cutoff frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Return Values

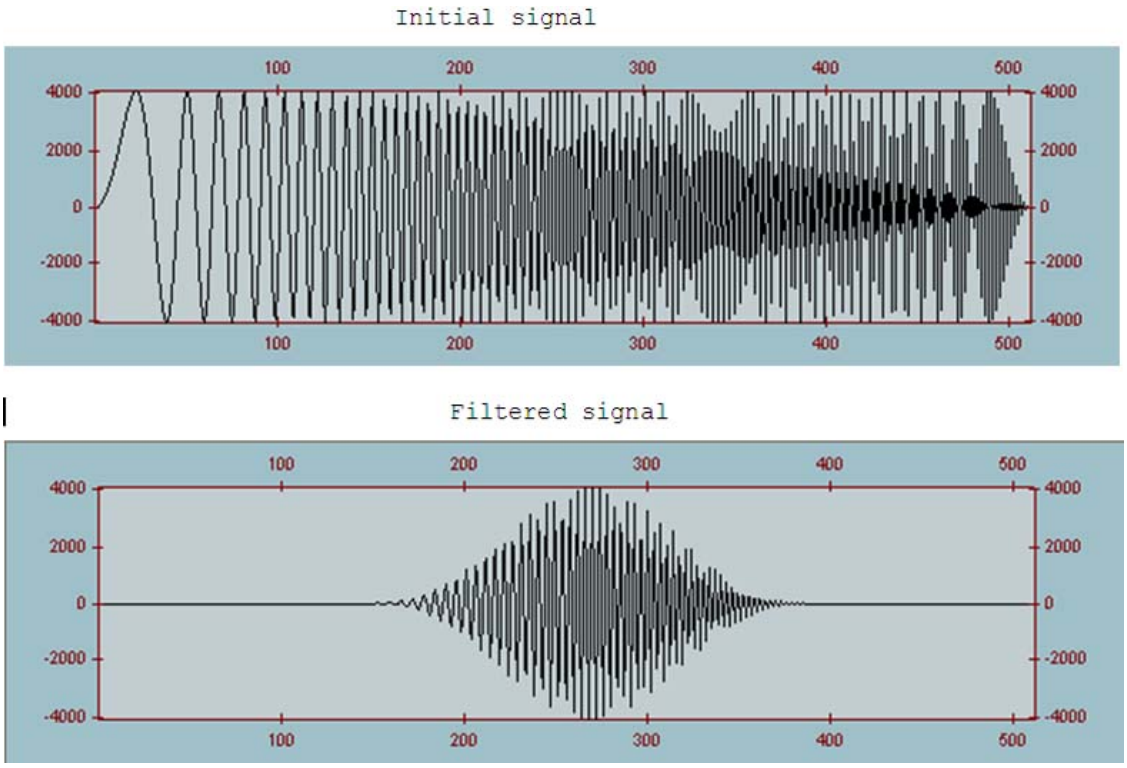
ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pTaps* pointer is *NULL*.
ippStsSizeErr	Indicates an error when the *tapsLen* is less than 5, or *rLowFreq* is greater than or equal to *rHighFreq*, or one of the frequency parameters *rLowFreq* and *rHighFreq* is out of the range.

Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

FIRGenBandpass.c

Result:



FIRGenBandstop

Computes bandstop FIR filter coefficients.

Syntax

```
IppStatus ippsFIRGenBandstop_64f(Ipp64f rLowFreq, Ipp64f rHighFreq, Ipp64f* pTaps, int
tapsLen, IppWinType winType, IppBool doNormal, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

rLowFreq	Normalized low cutoff frequency, must be in the range (0, 0.5) and less than *rHighFreq*.
rHighFreq	Normalized high cutoff frequency, must 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, must be equal or greater than 5.
winType	Specifies what type of window is 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	Specifies normalized or non-normalized sequence of the filter coefficients is 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.
pBuffer	Pointer to the buffer for internal calculations. To get the size of the buffer, use the `ippsFIRGenGetBufferSize` function.

Description

This function computes *tapsLen* coefficients for bandstop FIR filter with the cutoff frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see Windowing Functions. The computed coefficients are stored in the array *pTaps*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTaps* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when the *tapsLen* is less than 5, or *rLowFreq* is greater than or equal to *rHighFreq*, or one of the frequency parameters *rLowFreq* and *rHighFreq* is out of the range.

Single-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a single-rate FIR least mean squares (LMS) filter
- get and set the delay line values
- get the filter coefficients (taps) values
- perform filtering

FIRLMSGetTaps

Retrieves the tap values from the FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSGetTaps_32f(const IppsFIRLMSState_32f* pState, Ipp32f* pOutTaps);
```

```
IppStatus ippsFIRLMSGetTaps32f_16s(const IppsFIRLMSState32f_16s* pState, Ipp32f*
pOutTaps);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the FIR LMS filter state structure.
pOutTaps	Pointer to the array holding copies of the taps.

Description

This function copies the taps from the state structure *pState* to the *tapsLen*-length array *pOutTaps*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSGetDlyLine

Retrieves the delay line contents from the FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState, Ipp32f* pDlyLine,
int* pDlyLineIndex);
```

```
IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s* pState, Ipp16s*
pDlyLine, int* pDlyLineIndex);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the FIR LMS filter state structure.
pDlyLine	Pointer to the *tapsLen* -length array holding the delay line values.
pDlyLineIndex	Pointer to the array to store the current delay line index copied from the filter state structure.

Description

This function copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pDlyLine* and *pDlyLineIndex*, respectively.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

FIRLMSSetDlyLine

Sets the delay line contents in the FIR LMS filter.

Syntax

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState, Ipp32f* pDlyLine,
int* pDlyLineIndex);
```

```
IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s* pState, Ipp16s*
pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIRLMSSetDlyLine_32f(IppsFIRLMSState_32f* pState, const Ipp32f* pDlyLine,
int dlyLineIndex);
```

```
IppStatus ippsFIRLMSSetDlyLine32f_16s(IppsFIRLMSState32f_16s* pState, const Ipp16s*
pDlyLine, int dlyLineIndex);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the FIR LMS filter state structure.
pDlyLine	Pointer to the *tapsLen*-length array holding the delay line values.
pDlyLineIndex	Pointer to the index of the delay line.
dlyLineIndex	Initial index of the delay line to be stored in the filter state structure *pState*.

Description

This function copies the delay line values from *pDlyLine*, and the current delay line index from *dlyLineIndex*, and stores them into the state structure *pState*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

FIRLMSGetStateSize

Computes the size of the external buffer for the FIR least mean squares (LMS) filter structure.

Syntax

```
IppStatus ippsFIRLMSGetStateSize32f_16s(int tapsLen, int dlyIndex, int* pBufferSize);
```

```
IppStatus ippsFIRLMSGetStateSize_32f(int tapsLen, int dlyIndex, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

tapsLen	Number of elements in the array containing tap values.
dlyIndex	Current index of the delay line.
pBufferSize	Pointer to the computed buffer size value.

Description

This function computes the size of the external buffer for the FIR LMS filter state structure and stores the result in *pBufferSize*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pBufferSize* is NULL.
ippStsFIRLenErr	Indicates an error when *tapsLen* is less than, or equal to zero.

See Also

FIRLMS Filters a vector through the FIR least mean squares (LMS) filter.

FIRLMSInit

Initializes the adaptive FIR least mean squares (LMS) filter state structure.

Syntax

```
IppStatus ippsFIRLMSInit32f_16s(IppsFIRLMSState32f_16s** ppState, const Ipp32f* pTaps,
int tapsLen, const Ipp16s* pDlyLine, int dlyIndex, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRLMSInit_32f(IppsFIRLMSState_32f** ppState, const Ipp32f* pTaps, int
tapsLen, const Ipp32f* pDlyLine, int dlyIndex, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

ppState	Double pointer to the state structure.
pTaps	Pointer to the array of tap values.
tapsLen	Number of elements in the array containing tap values.
pDlyLine	Pointer to the array containing delay line values. The number of elements in the array is 2*`tapsLen`.
dlyIndex	Current index of the delay line.

pBuffer	Pointer to the external buffer for the FIR LMS state structure.

Description

This function initializes the single-rate FIR LMS filter state structure. The `ippsFIRLMSInit` function copies the taps from the *pTaps* array of *tapsLen* length into the state structure *ppTaps*. The *pDlyLine* array of size 2**tapsLen* specifies the delay line values. The current index of the delay line is defined by *dlyIndex*. If the pointer to *pDlyLine* or *pTaps* is `NULL`, the corresponding value of the state structure is initialized to zero.

To compute the size of the buffer required for the FIR LMS state structure, use the `ippsFIRLMSGetStateSize` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when either *ppState* or *pBuffer* is `NULL`.
`ippStsFIRLenErr`	Indicates an error when *tapsLen* is less than, or equal to zero.

See Also

FIRLMS Filters a vector through the FIR least mean squares (LMS) filter.

FIRLMSGetStateSize Computes the size of the external buffer for the FIR least mean squares (LMS) filter structure.

FIRLMS

Filters a vector through the FIR least mean squares (LMS) filter.

Syntax

```
IppStatus ippsFIRLMS_32f(const Ipp32f* pSrc, const Ipp32f* pRef, Ipp32f* pDst, int len,
float mu, IppsFIRLMSState_32f* pState);
```

```
IppStatus ippsFIRLMS32f_16s(const Ipp16s* pSrc, const Ipp16s* pRef, Ipp16s* pDst, int
len, float mu, IppsFIRLMSState32f_16s* pState);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the FIR LMS filter state structure.
pSrc	Pointer to the source vector .
pRef	Pointer to the reference signal
pDst	Pointer to the output signal
len	Number of elements in the vector.
mu	Adaptation step.

Description

Before calling this function, compute the size of the buffer required for the *pState* structure using FIRLMSGetStateSize and initialize the structure using FIRLMSInit.

This function filters a source vector *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 element of the source vector *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 $h_{n+1}(i) = h_n(i) + 2 * mu * errVal * x(n-i)$,

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.

Return Values

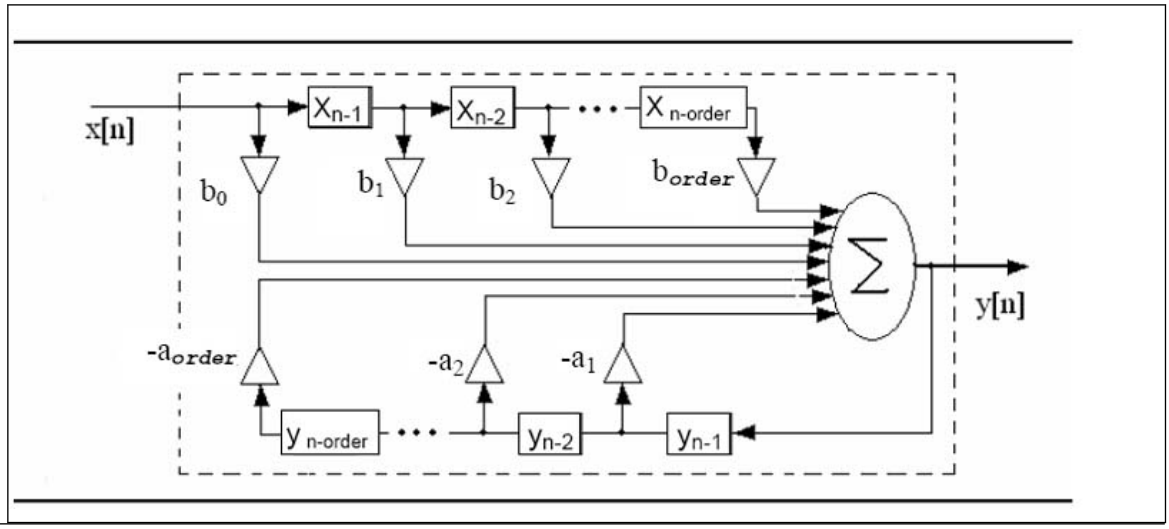
ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than, or equal to 0.
ippStsContextMatchErr	Indicates an error when the state identifier is incorrect.

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.

The figure below shows the structure of an arbitrary order IIR filter.

Structure of an Arbitrary Order Filter



Here `x[n]` is a sample of the input signal, `y[n]` is a sample of the output signal, *`order`* is the filter order, and b_0, b_1, . . ., b_{order}, a_1, . . ., a_{order} are the reduced filter coefficients.

The output signal is computed by the following formula:

$$y[n] = \sum_{k=0}^{order} b_k \cdot x(n-k) - \sum_{k=1}^{order} a_k \cdot y(n-k)$$

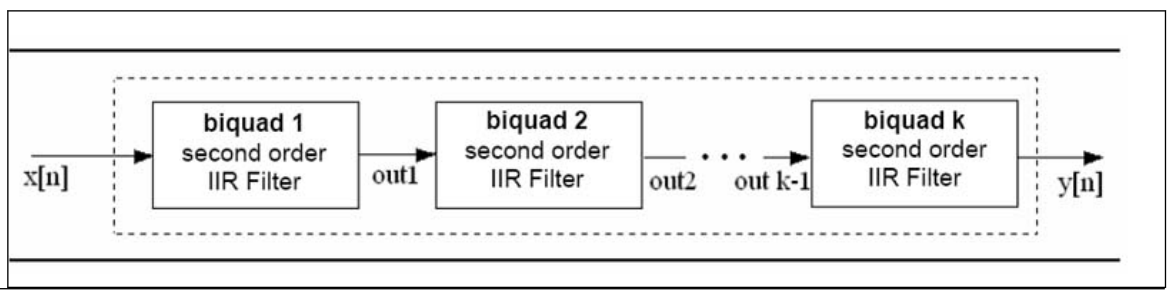
.

Reduced coefficients are calculated as $a_k = A_k/A_0$ and $b_k = B_k/A_0$

where A_0, A_1,...A_{order}, B_0, B_1,...B_{order} are initial filter coefficients (taps).

A biquad IIR filter is a cascade of second-order filters. The figure below illustrates the structure of the biquad filter with **k** cascades of second-order filters.

Structure of a BiQuad IIR Filter



By default, all Intel IPP IIR filter functions that do not have the `_DF1_` suffix in a name, use the direct form 2 (DF2) delay line. The difference between the direct form 1 (DF1) and DF2 representations of the delay line is that DF1 contains *delayed* values of the source and destination vectors, while DF2 is two times shorter and contains pre-calculated values based on the following code [Opp75]:

```
for( i = 0; i < order; i++ ){
    pDly[i] = 0;
    for( n = order - i; n > 0; n-- ){
        pDly[i] += pTaps[n+i] * pSrc[len-n]; /* b- coefficients */
```

```
    }
}
for( i = 0; i < order; i++ ){
    for( n = order - i; n > 0; n-- ){
        pDly[i] -= pTaps[order+n+i] * pDst[len-n]; /* a- coefficients */
    }
}
```

There is no way to transform DF2 back to DF1. Therefore, if you need DF1 output, copy the corresponding last order values of the source vector and last order values of the destination vector to DF1 buffer. Please note that the IIRSetDlyLine and IIRGetDlyLine functions get/return the delay line values also in DF2 form.

To initialize and use an IIR filter, follow this general scheme:

1. Call `ippsIIRInit` to initialize the filter as an arbitrary order IIR filter in the external buffer, or `ippsIIRInit_BiQuad` to initialize the filter as a cascade of biquads in the external buffer. Size of the buffer can be computed by calling the functions `ippsIIRGetStateSize` or `ippsIIRGetStateSize_BiQuad`, respectively.
2. Call `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.

IIRInit

Initializes an arbitrary IIR filter state.

Syntax

Case 1: Operation on integer samples

```
IppStatus ippsIIRInit32f_16s(IppsIIRState32f_16s** ppState, const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine, Ipp8u* pBuf);
IppStatus ippsIIRInit64f_16s(IppsIIRState64f_16s** ppState, const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuf);
IppStatus ippsIIRInit64f_32s(IppsIIRState64f_32s** ppState, const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuf);

IppStatus ippsIIRInit32fc_16sc(IppsIIRState32fc_16sc** ppState, const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u* pBuf);
IppStatus ippsIIRInit64fc_16sc(IppsIIRState64fc_16sc** ppState, const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuf);
IppStatus ippsIIRInit64fc_32sc(IppsIIRState64fc_32sc** ppState, const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuf);
```

Case 2: Operation on floating point samples

```
IppStatus ippsIIRInit_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine, Ipp8u* pBuf);
IppStatus ippsIIRInit64f_32f(IppsIIRState64f_32f** ppState, const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuf);
IppStatus ippsIIRInit_64f(IppsIIRState_64f** ppState, const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuf);

IppStatus ippsIIRInit_32fc(IppsIIRState_32fc** ppState, const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u* pBuf);
IppStatus ippsIIRInit64fc_32fc(IppsIIRState64fc_32fc** ppState, const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit_64fc(IppsIIRState_64fc** ppState, const Ipp64fc* pTaps, int
order, const Ipp64fc* pDlyLine, Ipp8u* pBuf);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pTaps	Pointer to the array containing the taps. The number of elements in the array is `2*(order+1)`.
order	Order of the IIR filter.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order*.
ppState	Pointer to the pointer to the arbitrary IIR state structure to be created.
pBuf	Pointer to the external buffer.

Description

This function initializes an arbitrary IIR filter state in the external buffer. The size of this buffer must be computed previously by calling the function `IIRGetStateSize`. The initialization functions copy the taps from the array *pTaps* into the state structure *pState*. The *order*-length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The `2*( order + 1)`-length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

$A_0 \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsDivByZeroErr`	Indicates an error when A_0 is equal to 0.
`ippStsIIROrderErr`	Indicates an error when *order* is less than or equal to 0.

IIRInit_BiQuad

Initializes an IIR filter state.

Syntax

Case 1: Operation on integer samples

```
IppStatus ippsIIRInit32f_BiQuad_16s(IppsIIRState32f_16s** ppState, const Ipp32f* pTaps,
int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit64f_BiQuad_16s(IppsIIRState64f_16s** ppState, const Ipp64f* pTaps,
int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit64f_BiQuad_32s(IppsIIRState64f_32s** ppState, const Ipp64f* pTaps,
int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit32fc_BiQuad_16sc(IppsIIRState32fc_16sc** ppState, const Ipp32fc*
pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit64fc_BiQuad_16sc(IppsIIRState64fc_16sc** ppState, const Ipp64fc*
pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit64fc_BiQuad_32sc(IppsIIRState64fc_32sc** ppState, const Ipp64fc*
pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit64f_BiQuad_DF1_32s(IppsIIRState64f_32s** ppState, const Ipp64f*
pTaps, int numBq, const Ipp32s* pDlyLine, Ipp8u* pBuf);
```

Case 2: Operation on floating point samples

```
IppStatus ippsIIRInit_BiQuad_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps, int
numBq, const Ipp32f* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit64f_BiQuad_32f(IppsIIRState64f_32f** ppState, const Ipp64f* pTaps,
int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit_BiQuad_64f(IppsIIRState_64f** ppState, const Ipp64f* pTaps, int
numBq, const Ipp64f* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit_BiQuad_32fc(IppsIIRState_32fc** ppState, const Ipp32fc* pTaps,
int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit64fc_BiQuad_32fc(IppsIIRState64fc_32fc** ppState, const Ipp64fc*
pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit_BiQuad_64fc(IppsIIRState_64fc** ppState, const Ipp64fc* pTaps,
int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRInit_BiQuad_DF1_32f(IppsIIRState_32f** pState, const Ipp32f* pTaps,
int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuf);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pTaps	Pointer to the array containing the taps. The number of elements in the array is 6**numBq*.
numBq	Number of cascades of biquads.

pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is 2**numBq*.
ppState	Pointer to the pointer to the biquad IIR state structure.
pBuf	Pointer to the external buffer.
pState	Pointer to the state structure.

Description

This function initializes a biquad (BQ) IIR filter state in the external buffer. The size of this buffer must be computed previously by calling the corresponding function `ippsIIRGetStateSize_BiQuad`. The initialization function copies the taps from the array *pTaps* into the state structure *ppState*. The array *pDlyLine* specifies the delay line values. The number of elements in the array *pDlyLine* is 4**numBq* for the function flavor `ippsIIRInit_BiQuad_DF1`, and 2* *numBq* for all other flavors.

If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

The function flavor `ippsIIRInit_BiQuad_DF1` operates with the delay line values that are arranged in the array as follows:

$x_{0,-2}$, $x_{0,-1}$, $B_{0,2}$, $y_{0,-2}$, $y_{0,-1}$, $x_{1,-2}$, $x_{1,-1}$, $y_{1,-2}$, $y_{1,-1}$, ... $x_{numBq-1,-2}$, $x_{numBq-1,-1}$, $y_{numBq-1,-2}$, $y_{numBq-1,-1}$.

A biquad IIR filter is defined by a cascade of biquads. The number of cascades of biquads is specified by the *numBq* value. The 6**numBq* -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}$, $B_{0,1}$, $B_{0,2}$, $A_{0,0}$, $A_{0,1}$, $A_{0,2}$; $B_{1,0}$, $B_{1,1}$, $B_{1,2}$, $A_{1,0}$, $A_{1,1}$, $A_{1,2}$; . . . $A_{numBq-1,2}$

$A_{n,0} \neq 0$, $B_{n,0} \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsDivByZeroErr`	Indicates an error when A_0, $A_{n,0}$ or $B_{n,0}$ is equal to 0.
`ippStsIIROrderErr`	Indicates an error when *numBq* is less than or equal to 0.

IIRGetStateSize

Computes the length of the external buffer for the arbitrary IIR filter state structure.

Syntax

```
IppStatus ippsIIRGetStateSize32f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32s(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize32fc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32sc(int order, int* pBufferSize);
```

```
IppStatus ippsIIRGetStateSize_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64fc(int order, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

order	Order of the IIR filter.
pBufferSize	Pointer to the computed buffer size value.

Description

This function computes the size of the external buffer for an arbitrary IIR filter state, and stores the result in *pBufferSize*.

To compute a size of the buffer, the filter order parameter *order* must be specified.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pBufferSize* pointer is NULL.
ippStsIIROrderErr	Indicates an error when *order* is less than or equal to 0.

IIRGetStateSize_BiQuad

Computes the length of the external buffer for the biquad IIR filter state structure.

Syntax

```
IppStatus ippsIIRGetStateSize32f_BiQuad_16s(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_16s(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32s(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize32fc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32sc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64f(int numBq, int* pBufferSize);
```

```
IppStatus ippsIIRGetStateSize_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_DF1_32s(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_DF1_32f(int numBq, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

numBq	Number of cascades of biquads.
pBufferSize	Pointer to the computed buffer size value.

Description

This function computes the size of the external buffer for a corresponding biquad IIR filter state, and stores the result in *pBufferSize*.

To compute a size of the buffer, the number of cascades of biquads *numBq* must be specified.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pBufferSize* pointer is NULL.
ippStsIIROrderErr	Indicates an error when *numBq* is less than or equal to 0.

IIRGetDlyLine

Retrieves the delay line contents from the IIR filter state.

Syntax

```
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 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 ippsIIRGetDlyLine_32f(const IppsIIRState_32f* pState, Ipp32f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32f(const IppsIIRState64f_32f* pState, Ipp64f*
pDlyLine);
IppStatus ippsIIRGetDlyLine_64f(const IppsIIRState_64f* pState, Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine_32fc(const IppsIIRState_32fc* pState, Ipp32fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32fc(const IppsIIRState64fc_32fc* pState, Ipp64fc*
pDlyLine);
IppStatus ippsIIRGetDlyLine_64fc(const IppsIIRState_64fc* pState, Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_DF1_32s(const IppsIIRState64f_32s* pState, Ipp32s*
pDlyLine);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the IIR filter state structure.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters.

Description

This function copies the delay line values from the corresponding state structure *pState* and stores them into the *pDlyLine* array. If the pointer is `NULL`, then the delay line values in the state structure are initialized to zero.

The corresponding filter state must be initialized beforehand by one of the initialization functions.

Return Values

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

IIRSetDlyLine

Sets the delay line contents in an IIR filter state.

Syntax

```
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 ippsIIRSetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState, const Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState, const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState, const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine_32f(IppsIIRState_32f* pState, const Ipp32f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32f(IppsIIRState64f_32f* pState, const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine_64f(IppsIIRState_64f* pState, const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine_32fc(IppsIIRState_32fc* pState, const Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState, const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine_64fc(IppsIIRState_64fc* pState, const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_DF1_32s(IppsIIRState64f_32s* pState, const Ipp32s* pDlyLine);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the IIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is *order* for arbitrary filters and 2**numBq* for BQ filters. If the pointer is `NULL`, then the delay line values in the state structure are initialized to zero.

Description

This function copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. If the pointer is `NULL`, then the delay line values in the state structure are initialized to zero.

The filter state must be initialized beforehand by one of the initialization functions.

Return Values

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

IIR

Filters a source vector through an IIR filter.

Syntax

Case 1: Not-in-place operation on integer samples

```
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 ippsIIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
IppsIIRState32fc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsIIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
IppsIIRState64fc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsIIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int len,
IppsIIRState64fc_32sc* pState, int scaleFactor);
```

Case 2: Not-in-place operation on floating point samples

```
IppStatus ippsIIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, IppsIIRState_32f*
pState);
```

```
IppStatus ippsIIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, IppsIIRState_64f*
pState);
```

```
IppStatus ippsIIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
IppsIIRState64f_32f* pState);
```

```
IppStatus ippsIIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len, IppsIIRState_32fc*
pState);
```

```
IppStatus ippsIIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len, IppsIIRState_64fc*
pState);
```

```
IppStatus ippsIIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
IppsIIRState64fc_32fc* pState);
```

Case 3: In-place operation on integer samples

```
IppStatus ippsIIR32f_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState32f_16s* pState,
int scaleFactor);
```

```
IppStatus ippsIIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int len, IppsIIRState32fc_16sc*
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 ippsIIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int len, IppsIIRState64fc_16sc*
pState, int scaleFactor);
```

```
IppStatus ippsIIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int len, IppsIIRState64fc_32sc*
pState, int scaleFactor);
```

Case 4: In-place operation on floating point samples

```
IppStatus ippsIIR_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState_32f* pState);
```

```
IppStatus ippsIIR_64f_I(Ipp64f* pSrcDst, int len, IppsIIRState_64f* pState);
```

```
IppStatus ippsIIR64f_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState64f_32f* pState);
IppStatus ippsIIR_32fc_I(Ipp32fc* pSrcDst, int len, IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc_I(Ipp64fc* pSrcDst, int len, IppsIIRState_64fc* pState);
IppStatus ippsIIR64fc_32fc_I(Ipp32fc* pSrcDst, int len, IppsIIRState64fc_32fc* pState);
```

Case 4: Operation with specified number of vector

```
IppStatus ippsIIR_32f_P(const Ipp32f** ppSrc, Ipp32f** ppDst, int len, int nChannels,
IppsIIRState_32f** ppState);
IppStatus ippsIIR64f_32s_PSfs(const Ipp32s** ppSrc, Ipp32s** ppDst, int len, int
nChannels, IppsIIRState64f_32s** ppState, int* pScaleFactor);
IppStatus ippsIIR_32f_IP(Ipp32f** ppSrcDst, int len, int nChannels, IppsIIRState_32f**
ppState);
IppStatus ippsIIR64f_32s_IPSfs(Ipp32s** ppSrcDst, int len, int nChannels,
IppsIIRState64f_32s** ppState, int* pScaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the IIR filter state structure.
ppState	Pointer to the array of the pointers to the IIR filter state structures.
pSrc	Pointer to the source vector.
ppSrc	Pointer to the array of pointers to the source vectors.
pDst	Pointer to the destination vector.
ppDst	Pointer to the array of pointers to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operations.
ppSrcDst	Pointer to the array of pointers to the source and destination vectors for the in-place operations.
len	Number of elements of the vector to be filtered.
nChannels	Number of vectors to be filtered.
scaleFactor	Scale factor, refer to Integer Scaling.
pScaleFactor	Pointer to the scale factor.

Description

This function filters *len* elements of the source vector *pSrc* or *pSrcDst* through an IIR filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

The filter state must be initialized before calling the function `ippsIIR`. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

Function flavors described in the **Case 4** filter simultaneously the *nChannels* source vectors. Each vector must have the *len* elements and is filtered with its own state structure. These state structures must be initialized beforehand.

Example demonstrates how to use the function `ippsIIR` to filter a sample. The function `ippsConvert_64f32s_Sfs` converts floating-point taps into integer data type before calling `ippsIIRInitAlloc_32s`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less or equal to 0.
`ippStsChannelErr`	Indicates an error when *nChannels* is less or equal to 0.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`IIR.c`

IIRSparseInit

Initializes a sparse IIR filter structure.

Syntax

```
IppStatus ippsIIRSparseInit_32f(IppsIIRSparseState_32f** ppState, const Ipp32f*
pNZTaps, const Ipp32s* pNZTapPos, int nzTapsLen1, int nzTapsLen2, const Ipp32f*
pDlyLine, Ipp8u* pBuf);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pNZTaps	Pointer to the array containing the non-zero tap values.
pNZTapPos	Pointer to the array containing positions of the non-zero tap values. The number of elements in the array is *nzTapsLen*.
nzTapsLen1, *nzTapsLen2*	Pointer to the destination vector.
pDlyLine	Pointer to the array containing the delay line values.
ppState	Double pointer to the sparse IIR state structure.
pBuf	Pointer to the external buffer for the sparse IIR state structure.

Description

This function initializes a sparse IIR filter state structure *ppState* in the external buffer *pBuf*. The size of this buffer must be computed previously by calling the function ippsIIRSparseGetStateSize.

The (*nzTapsLen1* + *nzTapsLen2*)-length array *pNZTaps* specifies the non-zero taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{nzTapsLen1-1}, A_0, A_1, \ldots, A_{nzTapsLen2-1}$.

The (*nzTapsLen1* + *nzTapsLen2*)-length array *pNZTapPos* specifies the non-zero tap positions arranged in the array as follows:

$BP_0, BP_1, \ldots, BP_{nzTapsLen1-1}, AP_0, AP_1, \ldots, AP_{nzTapsLen2-1}, AP_0 \neq 0$

The initialization function copies the values of filter coefficients from the array *pNZTaps* containing non-zero taps and their positions from the array *pNZTapPos* into the state structure *ppState*. The array *pDlyLine* contains the delay line values.The number of elements in this array is *pNZTapPos*[*nzTapsLen1*-1] + *pNZTapPos*[*nzTapsLen1* + *nzTapsLen2*- 1]. If the pointer to the array *pDlyLine* is not NULL, the array contents is copied into the state structure *ppState*, otherwise the delay line values in the state structure are initialized to 0.

NOTE
The values of *nzTapsLen1*, *nzTapsLen2*, *pNZTapPos*[*nzTapsLen* -1], and *pNZTapPos*[*nzTapsLen1* + *nzTapsLen2*- 1] must be equal to those specified in the function ippsIIRSparseGetStateSize.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsIIROrderErr	Indicates an error if *nzTapsLen1* is less than or equal to 0, or *nzTapsLen2* is less than 0.
ippStsSparseErr	Indicates an error if positions of the non-zero taps are not in ascending order, or are negative or repetitive; or *pNZTapPos*[*nzTapsLen1*] is equal to 0.

IIRSparseGetStateSize

Computes the size of the external buffer for the sparse IIR filter structure.

Syntax

```
IppStatus ippsIIRSparseGetStateSize_32f(int nzTapsLen1, int nzTapsLen2, int order1, int order2, int* pStateSize);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

nzTapsLen1, *nzTapsLen2*	Number of elements in the array containing the non-zero tap values.
order1, *order2*	Order of the sparse IIR filter.
pStateSize	Pointer to the computed value of the external buffer.

Description

This function computes the size in bytes of the external buffer for a sparse IIR filter state that is required for the function `ippsIIRSparseInit`. The computations are based on the specified number of non-zero filter coefficients *nzTapsLen1*, *nzTapsLen2* and filter orders *order1*, *order2*. *order1* = *pNZTapPos*[*nzTapsLen1* -1], *order2* = *pNZTapPos*[*nzTapsLen1* + *nzTapsLen2*- 1] (see description of the function `ippsIIRSparseInit` for more details). The result value is stored in the *pStateSize*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pStateSize* pointer is NULL.
ippStsIIROrderErr	Indicates an error if *nzTapsLen1* is less than or equal to 0, or *nzTapsLen2* is less than 0.
ippStsSparseErr	Indicates an error if *order1* or *order2* is less than 0.

IIRSparse

Filters a source vector through a sparse IIR filter.

Syntax

```
IppStatus ippsIIRSparse_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
IppsIIRSparseState_32f* pState);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pState	Pointer to the sparse IIR filter state structure.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements that will be filtered.

Description

This function applies the sparse IIR filter to the *len* elements of the source vector *pSrc*, and stores the results in *pDst*. The filter parameters - the number of non-zero taps *nzTapsLen1*, *nzTapsLen2*, their values *pNZTaps* and their positions *pNZTapPos*, and the delay line values *pDlyLine* - are specified in the sparse IIR filter structure *pState* that should be previously initialized the function `ippsIIRSparseInit`.

In the following definition of the sparse IIR filter, the sample to be filtered is denoted `x(n)`, the non-zero taps are denoted B_i and A_i, their positions are denoted BP_i and AP_i.

The non-zero taps are arranged in the array as follows:

$B_0, B_1, \ldots, B_{nzTapsLen1-1}, A_0, A_1, \ldots, A_{nzTapsLen2-1}$.

The non-zero tap positions are arranged in the array as follows:

$BP_0, BP_1, \ldots, BP_{nzTapsLen1-1}, AP_0, AP_1, \ldots, AP_{nzTapsLen2-1}, AP_0 \neq 0$

The return value is `y(n)` is defined by the formula for a sparse IIR filter:

$$y(n) = \sum_{k=0}^{nzTapsLen1-1} B_k \cdot x(n - BP_k) + \sum_{k=1}^{nzTapsLen2-1} A_k \cdot y(n - AP_k) \quad 0 \le n < len$$

After the function has performed calculations, it updates the delay line values stored in the filter state structure.

Return Values

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

Example

The example below shows how to use the sparse IIR filter functions.

```
int buflen;  Ipp8u *buf;  int nzTapsLen1 = 5;    //number of non-zero taps in the FIR part of the
formula  int nzTapsLen2 = 3;    //number of non-zero taps in the IIR part of the formula  Ipp32f
nzTaps [] = {0.5, 0.4, 0.3, 0.2, 0.1, 0.8, 0.7,
0.6};                                                //non-zero taps values (FIR+IIR)  Ipp32s
nzTapsPos[] = {0, 10, 20, 30, 40, 1, 5, 15};                                          //non-zero
tap positions (FIR+IIR)  IppsIIRSparseState_32f* iirState;  Ipp32f *src, *dst; /
* ............................................ */  ippsIIRSparseGetStateSize_32f(nzTapsLen1,
nzTapsLen2, nzTapsPos [nzTapsLen1 - 1],                                  nzTapsPos [nzTapsLen1 +
nzTapsLen2 - 1], &buflen);  buf = ippsMalloc_8u(buflen);  ippsIIRSparseInit_32f(&iirState, nzTaps,
nzTapsPos, nzTapsLen1, nzTapsLen2,                        NULL, buf);  /* . . . . initializing
src somehow . . . . */ ippsIIRSparse_32f(src, dst, len, iirState); /* dst[i] = src[i] * 0.5 +
src[i-10] * 0.4 +  src[i-20] * 0.3 +  src[i-30] * 0.2            +  src[i-40] * 0.1 + dst[i-1] *
0.8 + dst[i-5] * 0.7 + dst[i-15] * 0.6  */ /* ..................................................
*/ ippsFree(buf);
```

IIRGenGetBufferSize

Computes the size of the buffer required for `ippsIIRGenLowpass` *and* `ippsIIRGenHighpass` *internal calculations.*

Syntax

```
IppStatus ippsIIRGenGetBufferSize(int order, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

order	Order of the filter [1, 12].
pBufferSize	Pointer to the calculated buffer size (in bytes).

Description

This function computes the size of the buffer that is required for `ippsIIRGenLowpass`/`ippsIIRGenHighpass` internal calculations.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsIIRGenOrderErr`	Indicates an error when the *order* value is less than 1, or greater than 12.

See Also

IIRGenLowpass, IIRGenHighpass Computes lowpass and highpass IIR filter coefficients.

IIRGenLowpass, IIRGenHighpass

Computes lowpass and highpass IIR filter coefficients.

Syntax

```
IppStatus ippsIIRGenLowpass_64f(Ipp64f rFreq, Ipp64f ripple, int order, Ipp64f* pTaps,
IppsIIRFilterType filterType, Ipp8u* pBuffer);
```

```
IppStatus ippsIIRGenHighpass_64f(Ipp64f rFreq, Ipp64f ripple, int order, Ipp64f* pTaps,
IppsIIRFilterType filterType, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

rFreq	Cutoff frequency, should be in the range (0, 0.5).
ripple	Posible ripple in pass band for `ippChebyshev1` type of filter.
order	Order of the filter [1, 12]
pTaps	Pointer to the array where computed tap values are stored.
filterType	Type of the IIR filter, possible values: `ippButterworth`, `ippChebyshev1`.

pBuffer	Pointer to the buffer for internal calculations. To get the size of the buffer, use the `ippsIIRGenGetBufferSize` function.

Description

These functions computes coefficients for lowpass or highpass IIR filters, respectively, with the cutoff frequency *rFreq*. The parameter *filterType* specifies the type of the filter. The computed coefficients are stored in the array *pTaps*. Its length must be at least 2*(*order* + 1) and the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

Application Notes

Butterworth filters are characterized by a magnitude response that is at most flat in the passband and monotonic overall. Butterworth filters sacrifice rolloff steepness for monotonicity in the passband. Unless the smoothness of the Butterworth filter is needed, *Chebyshev1 filter* can generally provide steeper rolloff characteristics with a lower filter order. Chebyshev1 type filters are equiripple in the passband and monotonic in the stopband. For `ippButterworth` filter cutoff frequency is the frequency where the magnitude response of the filter is $2^{-1/2}$. For `ippChebyshev1` filter cutoff frequency is the frequency at which the magnitude response of the filter is (-*ripple*) dB. For the functions `ippsIIRGenLowpass` and `ippsIIRGenHighpass`, the normalized cutoff frequency *rFreq* must be a number between 0 and 0.5, where 0.5 corresponds to the Nyquist frequency, π radians per sample. The correspondence between MATLAB's *Wn* and Intel IPP *rFreq* is very simple: *Wn = 2*rFreq*.

Examples:

1) For data sampled at 1000 Hz, create a 9th-order highpass Butterworth filter with cutoff frequency at 300 Hz.

Intel IPP:

```
Ipp64f pTaps[2*(9+1)];
```

```
status = ippsIIRGenHighpass( 300.0/1000.0, 0, 9, pTaps, ippButterworth );
```

MATLAB:

```
[b,a] = butter(9,300/500,'high');
```

2) For data sampled at 1000 Hz, create a 9th-order lowpass Chebyshev1 filter with ripple in the passband of 0.5 dB and a cutoff frequency at 300 Hz.

Intel IPP:

```
Ipp64f pTaps[2*(9+1)];
```

```
status = ippsIIRGenLowpass( 300.0/1000.0, 0.5, 9, pTaps, ippChebyshev1 );
```

MATLAB:

```
[b,a] = cheby1(9, 0.5, 300/500);
```

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pTaps* pointer is `NULL`.
`ippStsIIRGenOrderErr`	Indicates an error when the *order* is less than 1 or greater than 12.
`ippStsFilterFrequencyErr`	Indicates an error when the *rFreq* is out of the range.

See Also

IIRGenGetBufferSize Computes the size of the buffer required for `ippsIIRGenLowpass` and `ippsIIRGenHighpass` internal calculations.

IIRIIR Filter Functions

The functions described in this section initialize an infinite impulse response (IIR) filter and perform a zero-phase digital filtering of input data in both forward and backward directions. The formulas below explain why the filtered signal has zero-phase distortion. Consider the following case in the frequency domain: if `x(n)` is the input sequence and `h(n)` is the IIR filter's impulse response, then the result of the forward filter pass is:

$$Y_1\left(e^{i\phi}\right) = X\left(e^{i\phi}\right) * H\left(e^{i\phi}\right)$$

where

- $X(e^{i\varphi})$ is the Fourier transform of `x(n)`
- $H(e^{i\varphi})$ is the Fourier transform of `h(n)`
- $Y_1(e^{i\varphi})$ is the Fourier transform of the forward filter pass

Backward filtering corresponds to filtering of time-reversed signal. Time reversal corresponds to replacing φ with $-\varphi$ in the frequency domain, so the result of time reversal is:

$$Y_1\left(e^{-i\phi}\right) = X\left(e^{-i\phi}\right) * H\left(e^{-i\phi}\right)$$

When the filter is applied for the second time, the above formula is multiplied by the Fourier transform of the filter's impulse response function $H(e^{i\varphi})$:

$$Y_1\left(e^{-i\phi}\right) = X\left(e^{-i\phi}\right) * H\left(e^{-i\phi}\right) * H\left(e^{i\phi}\right)$$

The final time reversal in the frequency domain results in:

$$Y_1\left(e^{-i\phi}\right) = X\left(e^{i\phi}\right) * H\left(e^{i\phi}\right) * H\left(e^{-i\phi}\right) = X\left(e^{i\phi}\right) * \left|H\left(e^{i\phi}\right)\right|^2$$

You can see from the resulting equation that:

- The filtered signal has zero-phase distortion (as the filtering was done with $|H(e^{i\varphi})|^2$, which is purely real-valued)
- The filter transfer function has the squared magnitude of the original filter transfer function
- The filter order is double the order of the initialized IIR filter

To initialize and use an IIRIIR filter, follow this general scheme:

1. Call `ippsIIRIIRInit` to initialize the IIRIIR filter in the external buffer. To compute the size of the buffer, use the `ippsIIRIIRGetStateSize` function.
2. Call `ippsIIRIIR` to filter a vector.
3. Call `ippsIIRIIRGetDlyLine` and `ippsIIRIIRSetDlyLine` to get and set the delay line values in the IIRIIR state structure.

IIRIIRGetStateSize

Computes the length of the external buffer for the IIRIIR filter state structure.

Syntax

```
IppStatus ippsIIRIIRGetStateSize_32f(int order, int* pBufferSize);
IppStatus ippsIIRIIRGetStateSize64f_32f(int order, int* pBufferSize);
IppStatus ippsIIRIIRGetStateSize_64f(int order, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

order	Order of the IIRIIR filter.
pBufferSize	Pointer to the computed buffer size value.

Description

This function computes the size of the external buffer for the IIRIIR filter state structure, and stores the result in *pBufferSize*.

Use this function before using `IIRIIRInit`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pBufferSize* pointer is `NULL`.
`ippStsIIROrderErr`	Indicates an error when *order* is less than or equal to zero.

See Also

IIRIIRInit Initializes the IIRIIR filter state structure.

IIRIIRInit

Initializes the IIRIIR filter state structure.

Syntax

```
IppStatus ippsIIRIIRInit_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps, int
order, const Ipp32f* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRIIRInit64f_32f(IppsIIRState64f_32f** ppState, const Ipp64f* pTaps, int
order, const Ipp64f* pDlyLine, Ipp8u* pBuf);
```

```
IppStatus ippsIIRIIRInit_64f(IppsIIRState_64f** ppState, const Ipp64f* pTaps, int
order, const Ipp64f* pDlyLine, Ipp8u* pBuf);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pTaps	Pointer to the array containing the taps. The number of elements in the array is `2*(order+1)`.
order	Order of the IIRIIR filter.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order*. If `IIRIIRInit` is called with *pDlyLine*==`NULL`, then it automatically forms the delay line that

	minimizes the start-up and ending transients. The line is formed by matching the initial conditions to remove the DC offset at the beginning and the end of the input vector.
ppState	Pointer to the pointer to the arbitrary IIRIIR state structure to be created.
pBuf	Pointer to the external buffer.

Description

This function initializes the arbitrary IIRIIR filter state structure in the external buffer. Before using the `IIRIIRInit` function, compute the size of the external buffer by calling the `IIRIIRGetStateSize` function. The initialization functions copy the taps from the *pTaps* array into the *pState* structure . The *order*-length array *pDlyLine* specifies the delay line values. If the *pDlyLine* pointer to the array is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to values that minimize the start-up and ending transients. These values are obtained by matching the initial conditions to remove the DC offset at the beginning and the end of the input vector.

The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The `2*( order + 1)`-length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \ldots, B_{order}, A_0, A_1, \ldots, A_{order}$

$A_0 \neq 0$

If the state structure is not created, the initialization function returns an error status.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsDivByZeroErr`	Indicates an error when A_0 is equal to zero.
`ippStsIIROrderErr`	Indicates an error when *order* is less than or equal to zero.

See Also

IIRIIRGetStateSize Computes the length of the external buffer for the IIRIIR filter state structure.

IIRIIRGetDlyLine

Retrieves the delay line contents from the IIIRIR filter state structure.

Syntax

```
IppStatus ippsIIRIIRGetDlyLine_32f(const IppsIIRState_32f* pState, Ipp32f* pDlyLine);
```

```
IppStatus ippsIIRIIRGetDlyLine64f_32f(const IppsIIRState64f_32f* pState, Ipp64f*
pDlyLine);
```

```
IppStatus ippsIIRIIRGetDlyLine_64f(const IppsIIRState_64f* pState, Ipp64f* pDlyLine);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the IIRIIR filter state structure.
pDlyLine	Pointer to the array containing the delay line values. The number of elements in the array is *order*.

Description

This function copies the delay line values from the corresponding *pState* structure and stores them into the *pDlyLine* array.

The corresponding filter state structure must be initialized beforehand by the initialization function.

Return Values

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

IIRIIRSetDlyLine

Sets the delay line contents in the IIRIIR filter state structure.

Syntax

```
IppStatus ippsIIRIIRSetDlyLine_32f(IppsIIRState_32f* pState, const Ipp32f* pDlyLine);
IppStatus ippsIIRIIRSetDlyLine64f_32f(IppsIIRState64f_32f* pState, const Ipp64f*
pDlyLine);
IppStatus ippsIIRIIRSetDlyLine_64f(IppsIIRState_64f* pState, const Ipp64f* pDlyLine);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pState	Pointer to the IIRIIR filter state structure.
pDlyLine	Pointer to the array holding the delay line values. The number of elements in the array is *order*. If the pointer is NULL, then the delay line values in the state structure are formed internally to minimize the start-up and ending transients. These values are obtained by matching the initial conditions to remove the DC offset at the beginning and the end of the input vector.

Description

This function copies the delay line values from *pDlyLine* and stores them into the *pState* structure . If the pointer is NULL, then the delay line values in the state structure are initialized to values that minimize the start-up and ending transients. These values are obtained by matching the initial conditions to remove the DC offset at the beginning and the end of the input vector.

The filter state must be initialized beforehand by the initialization function.

Return Values

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

IIRIIR

Filters a source vector through an IIRIIR filter.

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsIIRIIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, IppsIIRState_32f*
pState);
```

```
IppStatus ippsIIRIIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, IppsIIRState_64f*
pState);
```

```
IppStatus ippsIIRIIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
IppsIIRState64f_32f* pState);
```

Case 2: In-place operation

```
IppStatus ippsIIRIIR_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState_32f* pState);
```

```
IppStatus ippsIIRIIR_64f_I(Ipp64f* pSrcDst, int len, IppsIIRState_64f* pState);
```

```
IppStatus ippsIIRIIR64f_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState64f_32f* pState);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pState	Pointer to the IIRIIR filter state structure.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for the in-place operations.
len	Number of elements of the vector to be filtered.

Description

This function filters *len* elements of the source vector *pSrc* or *pSrcDst* through an IIRIIR filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*.

Before calling the ippsIIRIIR function, initialize the filter state structure by using the IIRIIRInit function and specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* value.

Return Values

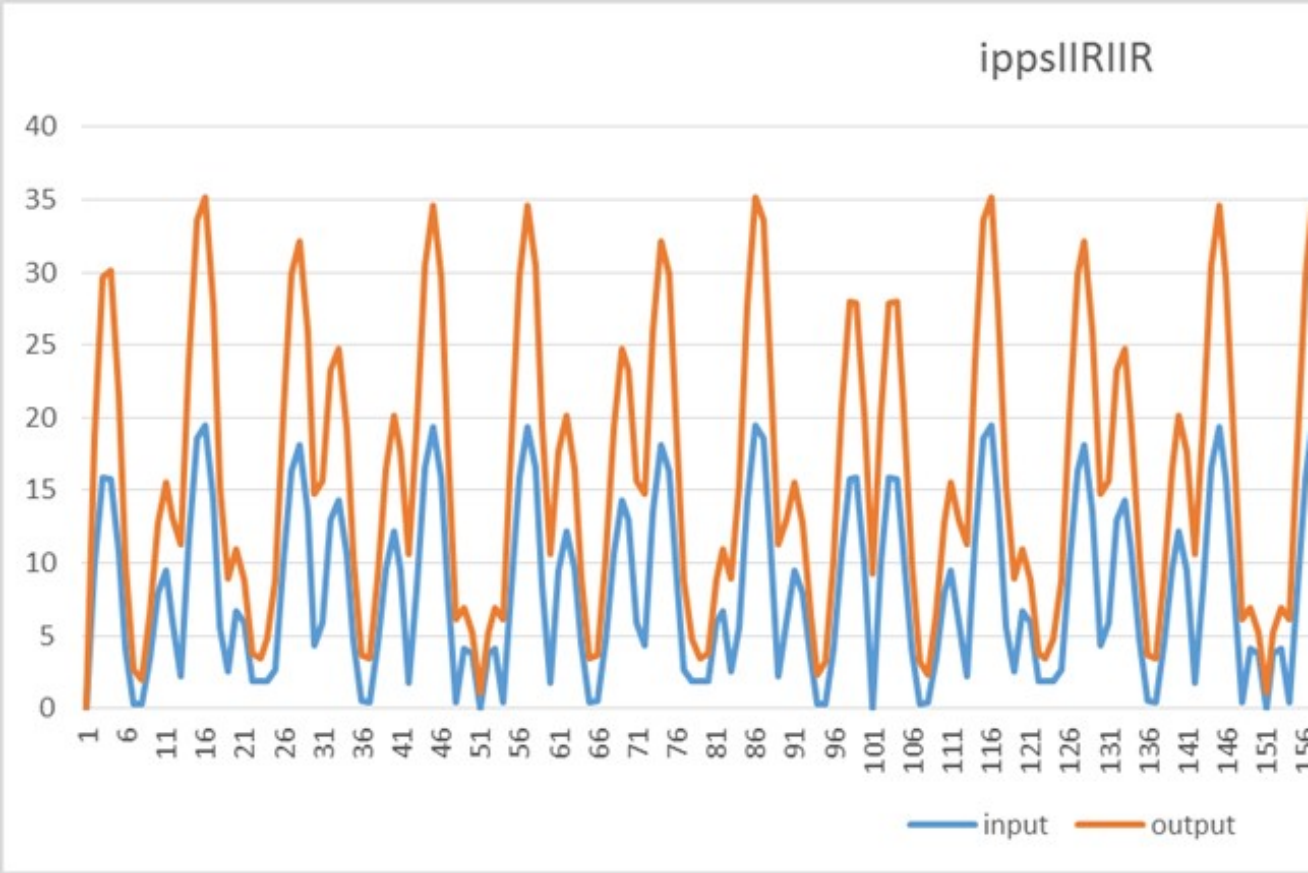
`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error when `length of the vectors < 3*(IIR order)`.
`ippStsContextMatchErr`	Indicates an error when the state identifier is incorrect.

Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`IIRIIR.c`

The figure below shows the result of the example, where the `X`-axis is index of the input/output vector/signal to see that there is no phase distortion, `Y`-axis - amplitude of input/output signals.



Median Filter Functions

Median filters are nonlinear rank-order filters based on replacing each element of the source vector 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 filter implementation in Intel IPP is that elements outside the source vector, which are needed to determine the median value for "border" elements, are located in a delay line. If the delay line is absent, then they are set to be equal to the corresponding edge element of the source vector.

FilterMedianGetBufferSize

Computes the size of the work buffer for the `ippsFilterMedian` *function.*

Syntax

```
IppStatus ippsFilterMedianGetBufferSize (int maskSize, IppDataType dataType, int*
pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

maskSize	Size of the median mask.
dataType	Data type of the source and destination vectors. Possible values are `ipp8u`, `ipp16s`, `ipp32s`, `ipp32f`, or `ipp64f`.
pBufferSize	Pointer to the computed size of the external work buffer, in bytes.

Description

The `ippsFilterMedianGetBufferSize` function computes the size, in bytes, of the external work buffer needed for the `ippsFilterMedian` function. The result is stored in the *pBufferSize* parameter.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pBufferSize* is `NULL`.
`ippStsMaskSizeErr`	Indicates an error when *maskSize* is less than, or equal to zero.
`ippStsDataTypeErr`	Indicates an error when *dataType* has an illegal value.
`ippStsEvenMedianMaskSize`	Indicates a warning when *maskSize* has an even value.

See Also

FilterMedianMODIFIED API. Computes median values for each source vector element.

FilterMedian

MODIFIED API. Computes median values for each source vector element.

Syntax

```
IppStatus ippsFilterMedian_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int maskSize,
const Ipp8u* pDlySrc, Ipp8u* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, int maskSize,
const Ipp16s* pDlySrc, Ipp16s* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, int maskSize,
const Ipp32s* pDlySrc, Ipp32s* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, int maskSize,
const Ipp32f* pDlySrc, Ipp32f* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, int maskSize,
const Ipp64f* pDlySrc, Ipp64f* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_8u_I(Ipp8u* pSrcDst, int len, int maskSize, const Ipp8u*
pDlySrc, Ipp8u* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_16s_I(Ipp16s* pSrcDst, int len, int maskSize, const Ipp16s*
pDlySrc, Ipp16s* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_32s_I(Ipp32s* pSrcDst, int len, int maskSize, const Ipp32s*
pDlySrc, Ipp32s* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_32f_I(Ipp32f* pSrcDst, int len, int maskSize, const Ipp32f*
pDlySrc, Ipp32f* pDlyDst, Ipp8u* pBuffer);
```

```
IppStatus ippsFilterMedian_64f_I(Ipp64f* pSrcDst, int len, int maskSize, const Ipp64f*
pDlySrc, Ipp64f* pDlyDst, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrcDst	Pointer to the source and destination vector (for the in-place operation).
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vector.
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.
pDlySrc	Pointer to the array containing values for the source delay lines.
pDlyDst	Pointer to the array containing values for the destination delay lines.
pBuffer	Pointer to the work buffer. To compute the size of the buffer, use the FilterMedianGetBufferSize function.

Description

This function computes median values for each element of the source vector *pSrc* or *pSrcDst*, and stores the result in *pDst* or *pSrcDst*, respectively.

NOTE
The values for non-existent elements are stored in *pDlySrc* (if it is not NULL). The last (*maskSize*-1) elements of vectors are stored in *pDlyDst* (if it is not NULL). For example, if *maskSize* is equal to 3, then:

```
pDst[0] = median(pDlySrc[0], pDlySrc[1], pSrc[0]);
pDst[1] = median(pDlySrc[1], pSrc[0], pSrc[1]);
pDst[2] = median(pSrc[0], pSrc[1], pSrc[2]);
...
pDlyDst[0] = pSrc[len-2];
pDlyDst[1] = pSrc[len-1]
```

If *pDlySrc* is NULL, the value of a non-existent element is equal to the value of the first vector element:

```
pDst[0] = median(pSrc[0], pSrc[0], pSrc[0]);
pDst[1] = median(pSrc[0], pSrc[0], pSrc[1]);
pDst[2] = median(pSrc[0], pSrc[1], pSrc[2]);
...
```

If *pDlyDst* is NULL, the operation of storing the last (*maskSize*-1) elements of vectors is not performed.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the*pSrcDst*, *pSrc*, *pDst*, *pBuffer* pointers is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsEvenMedianMaskSize	Indicates a warning when the median mask length is even.

Example

The example below illustrates using ippsFilterMedian_16s_I for single-rate filtering.

```
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 1 2 4 5 4 5 7
Matlab* Analog:
    >> x = [1 2 127 4 5 0 7 8]; medfilt1(x)
```

Polyphase Resampling Functions

The Intel® IPP functions described in this section build, apply, and free Kaizer-windowed polyphase filters for data resampling. Functions with the `Fixed` suffix are intended for fixed rational resampling factor and can provide faster speed. Functions without the suffix build universal resampling filter with linear interpolation of filter coefficients and enable a variable factor.

For general description of the polyphase resampling algorithm, see *"Multirate Digital Signal Processing"* by R. Crochire and L. Rabiner, [Cro83].

ResamplePolyphaseGetSize, ResamplePolyphaseFixedGetSize

Get the size of the polyphase resampling structure.

Syntax

```
IppStatus ippsResamplePolyphaseGetSize_16s(Ipp32f window, int nStep, int* pSize,
IppHintAlgorithm hint);
```

```
IppStatus ippsResamplePolyphaseGetSize_32f(Ipp32f window, int nStep, int* pSize,
IppHintAlgorithm hint);
```

```
IppStatus ippsResamplePolyphaseFixedGetSize_16s(int inRate, int outRate, int len, int*
pSize, int* pLen, int* pHeight, IppHintAlgorithm hint);
```

```
IppStatus ippsResamplePolyphaseFixedGetSize_32f(int inRate, int outRate, int len, int*
pSize, int* pLen, int* pHeight, IppHintAlgorithm hint);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

window	The size of the ideal lowpass filter window.
nStep	The discretization step for filter coefficients.
inRate	The input rate for fixed factor resampling.
outRate	The output rate for fixed factor resampling.
len	The filter length for fixed factor resampling.
pSize	The pointer to the variable that contains the size of the polyphase resampling structure.
pLen	The pointer to the variable that contains the real filter length.
pHeight	The pointer to the variable that contains the number of filters.
hint	Suggests using specific code (must be equal to `ippAlgHintFast`). Possible values for the *hint* parameter are given in Hint Arguments.

Description

These functions determine the size required for the fixed rate polyphase resampling structure and associated storage, the filter length, and the number of filters in the filter bank. The returned length of the filter is equal to min{l ≥*len*, l%4}, the filter length for zero phase is greater by 1. These values can be used for export and import of fixed polyphase resampling filter.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	For ippsResamplePolyphaseGetSize function, indicates an error when *inRate*, *outRate* , *nStep* or *len* is less than or equal to 0. For ippsResamplePolyphaseFixedGetSize function, indicates an error when *inRate*, *outRate*, *nStep* or *len* is less than or equal to 0.
ippStsBadArgErr	Indicates an error when *window* is less than 2/*nStep*.

ResamplePolyphaseInit, ResamplePolyphaseFixedInit

Initialize the structure for polyphase resampling with calculating the filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseInit_16s( Ipp32f window, int nStep, Ipp32f rollf, Ipp32f
alpha, IppsResamplingPolyphase_16s* pSpec, IppHintAlgorithm hint);
```

```
IppStatus ippsResamplePolyphaseInit_32f( Ipp32f window, int nStep, Ipp32f rollf, Ipp32f
alpha, IppsResamplingPolyphase_32f* pSpec, IppHintAlgorithm hint);
```

```
IppStatus ippsResamplePolyphaseFixedInit_16s( int inRate, int outRate, int len, Ipp32f
rollf, Ipp32f alpha, IppsResamplingPolyphaseFixed_16s* pSpec, IppHintAlgorithm hint);
```

```
IppStatus ippsResamplePolyphaseFixedInit_32f( int inRate, int outRate, int len, Ipp32f
rollf, Ipp32f alpha, IppsResamplingPolyphaseFixed_32f* pSpec, IppHintAlgorithm hint);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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 fixed factor resampling.
outRate	The output rate for fixed factor resampling.

len	The filter length for fixed factor resampling.
pSpec	The pointer to the resampling state structure.
hint	Suggests using specific code (must be equal to `ippAlgHintFast`). The possible values for the parameter *hint* are listed in Hint Arguments.

Description

The function `ippsResamplePolyphaseInit` initializes structures for data resampling using the ideal lowpass filter. The function `ippsResamplePolyphaseInit` applies the Kaiser window with alpha parameter and window width to the lowpass filter. This means that the values of the ideal lowpass filtering function are calculated for all *i* values such that $|i/nStep| \le window$.

Use the *pSpec* structure to resample input samples with the `ippsResample` function with arbitrary resampling factor. In this case, filter coefficients for each output sample are calculated using linear interpolation between two nearest values. The size of the filter depends on the resampling factor.

The function `ippsResamplePolyphaseFixedInit` initializes structures for data resampling with the factor equal to *inRate*/*outRate*. If you denote the number of filters created in the `IppsResamplingPolyphaseStructure` structure for input and output frequencies by *fnum*, then

fnum = *outRate*/*GCD* (*inRate*, *outRate*)

where

GCD (a, b) is the greatest common divisor of *a* and *b*. For example, if *inRate* = 8000 and *outRate* = 11025, then the number of filters will be *fnum* = 11025/*GCD* (8000, 11025) = 441.

Functions with the `Fixed` suffix pre-calculate filter coefficients for each phase and store them in the data structure for better performance. Use these functions when the ratio *inRate*/*outRate* is rational only. These functions can be considerably faster but may require large data structures for some input and output rates.

Before calling these functions, you need to allocate memory for the resampling state structure. To calculate the memory size, filter length, and the number of filters, use the `ippsResamplePolyphaseGetSize` or `ippsResamplePolyphaseFixedGetSize` functions.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	For `ippsResamplePolyphaseInit` function, indicates an error when *inRate*, *outRate* , *nStep* or *len* is less than or equal to 0. For `ippsResamplePolyphaseFixedInit` function, 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*.

ResamplePolyphaseSetFixedFilter

Sets polyphase resampling filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseSetFixedFilter_16s(const Ipp16s* pSrc, int step, int
height, IppsResamplingPolyphaseFixed_16s* pSpec);
```

```
IppStatus ippsResamplePolyphaseSetFixedFilter_32f(const Ipp32f* pSrc, int step, int
height, IppsResamplingPolyphaseFixed_32f* pSpec);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	The pointer to the input vector of filter coefficients.
step	The row step in *pSrc* vector.
height	The number of filters (the number of rows in *pSrc* vector).
pSpec	The pointer to the resampling state structure.

Description

This function imports pre-calculated filter coefficients into the polyphase resampling structure. If the *step* value is less than the filter length, trailing filter coefficients are zeroed.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.
ippStsBadArgErr	Indicates that *height* is greater than the number of filters in *pSpec* structure.

ResamplePolyphaseGetFixedFilter

Gets polyphase resampling filter coefficients.

Syntax

```
IppStatus ippsResamplePolyphaseGetFixedFilter_16s(Ipp16s* pDst, int step, int height,
const IppsResamplingPolyphaseFixed_16s* pSpec);
```

```
IppStatus ippsResamplePolyphaseGetFixedFilter_32f(Ipp32f* pDst, int step, int height,
const IppsResamplingPolyphaseFixed_32f* pSpec);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pDst	The pointer to the output vector of filter coefficients.

step	The row step in *pSrc* vector.
height	The number of filters (the number of rows in *pSrc* vector).
pSpec	The pointer to the resampling state structure.

Description

This function exports filter coefficients from the polyphase resampling structure. If the *step* value is less than the filter length, only first *step* coefficients are exported.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *step* or *height* is less than or equal to 0.
ippStsBadArgErr	Indicates an error when *height* is greater than the number of filters in *pSpec* structure.

Example

The code example below demonstrates export and import of the Polyphase Resampling Filter Bank.

```
int inRate=16000; // input frequency
 int outRate=8000; // output frequency
 int history; // half of filter length
 char fname[]="filter.flt\0";
 // coefficient file name
 {
 int size,len,height;
 FILE *file; short *pFilter;
 IppsresamplingPolyphaseFixed_16s *state;
 history=(int)(64.0f*0.5*IPP_MAX(1.0,1.0/(double)outRate/(double)inRate))+1;
 ippsResamplePolyphaseFixedGetSize_16s(inRate, outRate, 2*(history-1), &size, &len, &height,
ippAlgHintFast);
 state = (IppsResamlingPolyphaseFixed_16s*)ippsMalloc_8u(size);
 ippsResamplePolyphaseFixedInit_16s(inRate,outRate,2*(history-1), 0.95f, 9.0f, state,
ippAlgHintFast);
 pFilter=ippsMalloc_16s(len*height);
ippsResamplePolyphaseGetFixedFilter_16s(pFilter,len,height,state);
 file=fopen(fname,"wb"); fwrite(&size,sizeof(int),1,file);
 fwrite(&len,sizeof(int),1,file);
 fwrite(&height,sizeof(int),1,file);
 fwrite(pFilter,sizeof(short),len*height,file);
 fclose(file); ippsFree(pFilter);
 ippsFree (state);
 }
 {
 int size,len,height; FILE *file;
 short *pFilter;
 IppsresamplingPolyphaseFixed_16s *state;
 history=(int)(64.0f*0.5*IPP_MAX(1.0,1.0/(double)outRate/(double)inRate))+1;
 file=fopen(fname,"rb");
 fread(&size,sizeof(int),1,file);
 fread(&len,sizeof(int),1,file);
 fread(&height,sizeof(int),1,file);
 pFilter=ippsMalloc_16s(len*height);
 fread(pFilter,sizeof(short),len*height,file);
```

```
 fclose(file);
 state=(IppsresamplingPolyphaseFixed_16s*)ippsMalloc_8u(size);
 ippsResamplePolyphaseFixedInit_16s(inRate,outRate,2*(history-1), 0.95f, 9.0f, state,
ippAlgHintFast);
 ippsResamplePolyphaseSetFixedFilter_16s((const Ipp16s*)pFilter,len,height,
(IppsresamplingPolyphaseFixed_16s*)state);
 ippsFree(pFilter);
 // use of polyphase filter
 …
 ippsFree(state);
}
```

ResamplePolyphase, ResamplePolyphaseFixed

Resample input data using polyphase filters.

Syntax

```
IppStatus ippsResamplePolyphase_16s(const Ipp16s* pSrc, int len, Ipp16s* pDst, Ipp64f
factor, Ipp32f norm, Ipp64f* pTime, int* pOutlen, const IppsResamplingPolyphase_16s*
pSpec);
```

```
IppStatus ippsResamplePolyphase_32f(const Ipp32f* pSrc, int len, Ipp32f* pDst, Ipp64f
factor, Ipp32f norm, Ipp64f* pTime, int* pOutlen, const IppsResamplingPolyphase_32f*
pSpec);
```

```
IppStatus ippsResamplePolyphaseFixed_16s(const Ipp16s* pSrc, int len, Ipp16s* pDst,
Ipp32f norm, Ipp64f* pTime, int* pOutlen, const IppsResamplingPolyphaseFixed_16s*
pSpec);
```

```
IppStatus ippsResamplePolyphaseFixed_32f(const Ipp32f* pSrc, int len, Ipp32f* pDst,
Ipp32f norm, Ipp64f* pTime, int* pOutlen, const IppsResamplingPolyphaseFixed_32f*
pSpec);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	The pointer to the input vector.
pDst	The pointer to the output vector.
len	The number of input vector elements to resample.
norm	The norm factor for output samples.
factor	The resampling factor.
pTime	The pointer to the start time of resampling (in input vector elements). Keeps the input sample number and the phase for the first output sample from the next input data portion.
pOutlen	The number of calculated output vector elements.
pSpec	The pointer to the resampling state structure.

Description

These functions convert data from the input vector changing their frequency and compute all output samples that can be correctly calculated for the given input and the filter length. For the `ippsResamplePolyphase` function, the ratio of output and input frequencies is defined by the *factor* argument. For the `ippsResamplePolyphaseFixed` function, this ratio is defined during creation of the resampling structure. The value for *pTime*[0] defines the time value for which the first output sample is calculated.

Input vector with indices less than *pTime[0]* contains the history data of filters. The history length is equal to *flen*/2 for `ippsResamplePolyphaseFixed` function , and [1/2*window**max(1, 1/*factor*)]+1 for `ippsResamplePolyphase` function. Here *flen* is the filter length and *window* is the size of the ideal lowpass filter window. The input vector must contain the same number of elements with indices 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* * min (1, *factor*) before saturation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSpec*, *pSrc*, *pDst*, *pTime* or *pOutlen* pointer is *NULL*.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
ippStsBadArgErr	Indicates an error when *factor* is less than or equal to 0.

Example

The code example below demonstrates 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;
    int size,len,height;
    IppsResamplingPolyphaseFixed_16s *state;
    ippsResamplePolyphaseFixedGetSize_16s(inRate,outRate,2*(history-1),&size,&len,
&height,ippAlgHintAccurate);
    state=(IppsResamlingPolyphaseFixed_16s*)ippsMalloc_8u(size);
    ippsResamplePolyphaseFixedInit_16s(inRate,outRate,2*(history-1),0.95f,9.0f,state,
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(inBuf,lastread-history-(int)time,
                              outBuf,0.98f,&time,&outLen,state);
       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(inBuf,lastread-(int)time,
                        outBuf,0.98f,&time,&outLen,state);
   fwrite(outBuf,outLen,sizeof(short),outfd);
   outCount+=outLen;
   printf("%d inputs resampled to %d outputs\n",inCount,outCount);
   ippsFree(outBuf);
   ippsFree(inBuf);
   ippsFree (state);
}
```

Transform Functions

7

This chapter describes the Intel® IPP functions that perform Fourier and discrete cosine transforms (DCT), as well as Hartley, Hilbert, Walsh-Hadamard and wavelet transforms of signals.

Fourier Transform Functions

The functions described in this section perform the fast Fourier transform (FFT), the discrete Fourier transform (DFT) of signal samples. It also includes variations of the basic functions to support different application requirements.

Special Arguments

This section describes the flag and hint arguments used by the Fourier transform functions.

The Fourier transform functions require you to specify the *flag* and *hint* arguments.

The *flag* argument specifies the result normalization method. The following table lists the possible values 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.

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.

Packed Formats

This section describes the main packed formats Perm, Pack, and CCS used by the Fourier transform functions.

Pack Format

Perm Format

The Perm format stores the values in the order in which the Fourier transform algorithms use them. This is the most natural way of storing values for the Fourier transform algorithms. The Perm format is an arbitrary permutation of the Pack format. An important characteristic of the Perm format is that the real and imaginary parts of a given sample need not be adjacent.

NOTE
For input signal of odd length the perm and pack format are identical.

CCS Format

The CCS format stores the values of the first half of the output complex signal resulted from the forward Fourier transform.

Arrangement of Forward Fourier Transform Results in Packed Formats - Even Length

Index	0	1	2	3	. . .	N-2	N-1	N	N+1
Pack	R_0	R_1	I_1	R_2	. . .	$I_{(N-1)/2}$	$R_{N/2}$		
Perm	R_0	$R_{N/2}$	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	R_0	0	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

Forward Fourier transform Result Representation in Packed Formats - Odd Length

Index	0	1	2	3	. . .	N-2	N-1	N
Pack	R_0	R_1	I_1	R_2	. . .	$R_{(N-1)/2}$	$I_{(N-1)/2}$	
Perm	R_0	R_1	I_1	R_2	. . .	$R_{(N-1)/2}$	$I_{(N-1)/2}$	
CCS	R_0	0	R_1	I_1	. . .	$I_{(N-1)/2-1}$	$R_{(N-1)/2}$	$I_{(N-1)/2}$

Format Conversion Functions

The following functions ippsConjPack, ippsConjPerm, and ippsConjCcs convert data from the packed formats to a usual complex data format using the FFT symmetry property for transforming real data. The output data is complex, the output array length is defined by the number of complex elements in the output vector. Note that the output array size is two times as big as the input array size. The data stored in CCS format require a bigger array than the other formats. Even and odd length arrays have some specific features discussed for each function separately.

ConjPack

Converts the data in Pack *format to complex data format.*

Syntax

```
IppStatus ippsConjPack_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPack_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPack_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPack_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

lenDst	Number of elements in the vector.

Description

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*.

The table below 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.

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

Return Values

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

Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

`ConjPack.c`

ConjPerm

Converts the data in `Perm` *format to complex data format.*

Syntax

```
IppStatus ippsConjPerm_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPerm_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPerm_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

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

Description

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*.

The following table 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.

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

Return Values

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

ConjCcs

Converts the data in `CCS` *format to complex data format.*

Syntax

```
IppStatus ippsConjCcs_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjCcs_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjCcs_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

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*.

The following table 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.

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

Return Values

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

Functions for Packed Data Multiplication

The functions described in this section perform the element-wise complex multiplication of vectors stored in Pack or Perm formats. These functions are used with the function ippsFFTFwd and ippsFFTInv to perform fast convolution on real signals.

The standard vector multiplication function ippsMul can not be used to multiply Pack or Perm format vectors because:

- Two real samples are stored in Pack format.
- The Perm format might not pair the real parts of a signal with their corresponding imaginary parts.

> **NOTE**
> The vectors stored in CCS format can be multiplied using the standard function for complex data multiplication.

MulPack

Multiply the elements of two vectors stored in `Pack` *format.*

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsMulPack_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
IppStatus ippsMulPack_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, int len);
```

Case 2: In-place operation

```
IppStatus ippsMulPack_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMulPack_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc1, *pSrc2*	Pointers to the vectors whose elements are to be multiplied together.
pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc1*[n] * *pSrc2*[n].
pSrc	Pointer to the vector whose elements are to be multiplied by the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
len	Number of elements in the vector.

Description

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 `ippsMulPack` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The functions multiply the packed data according to their packed format. The data in Pack packed format include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

Return Values

`ippStsNoErr`	Indicates no error

ippStsNullPtrErr	Indicates an error when the *pSrcDst*, *pDst*, *pSrc1*, *pSrc2*, or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

MulPack_32f.c

MulPerm

Multiply the elements of two vectors stored in Perm *format.*

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsMulPerm_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
```

```
IppStatus ippsMulPerm_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, int len);
```

Case 2: In-place operation

```
IppStatus ippsMulPerm_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsMulPerm_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc1, *pSrc2*	Pointers to the vectors whose elements are to be multiplied together.
pDst	Pointer to the destination vector which stores the result of the multiplication *pSrc1*[n] * *pSrc2*[n].
pSrc	Pointer to the vector whose elements are to be multiplied by the elements of *pSrcDst* in-place.
pSrcDst	Pointer to the source and destination vector (for the in-place operation).
len	Number of elements in the vector.

Description

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 ippsMulPerm multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The function multiplies the packed data according to their packed format. The data in Perm packed formats include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pSrcDst*, *pDst*, *pSrc1*, *pSrc2*, or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

MulPackConj

Multiplies elements of a vector by the elements of a complex conjugate vector stored in Pack *format.*

Syntax

```
IppStatus ippsMulPackConj_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMulPackConj_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

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

Description

This function multiplies the elements of a source vector *pSrc* by elements of the vector that is complex conjugate to the source vector *pSrcDst* and stores the results in *pSrcDst*. The function performs only in-place operations on data stored in Pack format.

Return Values

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

Fast Fourier Transform Functions

The functions described in this section compute the forward and inverse fast Fourier transform of real and complex signals. The FFT is similar to the discrete Fourier transform (DFT) but is significantly faster. The length of the vector transformed by the FFT must be a power of 2.

To use the FFT functions, initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms. The amount of prior calculations is thus reduced and the overall performance increased.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method.

To initialize the FFT specification structure, use the `ippsFFTInit_R` and `ippsFFTInit_C` functions. Before using these functions, you need to compute the size of the specification structure using `ippsFFTGetSize_R` and `ippsFFTGetSize_C`, respectively.

The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in `Perm`, `Pack`, or `CCS` format.

You can speed up the FFT by using an external buffer. The use of external buffer can improve performance by avoiding allocation and deallocation of internal buffers and storing data in cache. The size of the external buffer is returned by the `ippsFFTInit_R` and `ippsFFTInit_C` functions.

If the external buffer is not specified (correspondent parameter is set to `NULL`), then the FFT function itself allocates the memory needed for operation.

FFTInit_R, FFTInit_C

Initializes the FFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsFFTInit_R_32f(IppsFFTSpec_R_32f** ppFFTSpec, int order, int flag,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

```
IppStatus ippsFFTInit_R_64f(IppsFFTSpec_R_64f** ppFFTSpec, int order, int flag,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

Case 2: Operation on complex signal

```
IppStatus ippsFFTInit_C_32f(IppsFFTSpec_C_32f** ppFFTSpec, int order, int flag,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

```
IppStatus ippsFFTInit_C_64f(IppsFFTSpec_C_64f** ppFFTSpec, int order, int flag,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

```
IppStatus ippsFFTInit_C_32fc(IppsFFTSpec_C_32fc** ppFFTSpec, int order, int flag,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

```
IppStatus ippsFFTInit_C_64fc(IppsFFTSpec_C_64fc** ppFFTSpec, int order, int flag,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

order	FFT order. The input signal length is $N = 2^{order}$.

flag	Specifies the result normalization method. The values for the *flag* argument are described in the section Flag and Hint Arguments.
hint	This parameter is deprecated. Set the value to `ippAlgHintNone`.
ppFFTSpec	Double pointer to the FFT specification structure to be created.
pSpec	Pointer to the area for the FFT specification structure.
pSpecBuffer	Pointer to the work buffer.

Description

These functions initialize the FFT specification structure *ppFFTSpec* with the following parameters:

- the transform *order*. This parameter defines the transform length. Input and output signals are arrays of 2 *order* length.
- the normalization *flag*
- the specific code *hint*

Before calling these functions, you need to compute the size of the specification structure and the work buffer (if it is required) using the `ippsFFTGetSize_R` and `ippsFFTGetSize_C` functions.

If `pSpecBufferSize` returned by the `ippsFFTGetSize` function is equal to zero, the parameter *pSpecBuffer* can be `NULL`.

The suffix after the function name indicates the flavors of the FFT functions: `ippsFFTInit_C` is for complex flavors and `ippsFFTInit_R` is for real flavors.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsFftOrderErr`	Indicates an error when the *order* value is incorrect.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.

See Also

FFTGetSize_R, FFTGetSize_C Computes sizes of the FFT specification structure and required working buffers.

FFTGetSize_R, FFTGetSize_C

Computes sizes of the FFT specification structure and required working buffers.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsFFTGetSize_R_32f(int order, int flag, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_R_64f(int order, int flag, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

Case 2: Operation on complex signal

```
IppStatus ippsFFTGetSize_C_32f(int order, int flag, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_64f(int order, int flag, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_32fc(int order, int flag, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_64fc(int order, int flag, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

order	FFT order. The input signal length is $N = 2^{order}$.
flag	Specifies the result normalization method. The values for the *flag* argument are described in Flag and Hint Arguments.
hint	This parameter is deprecated. Set the value to `ippAlgHintNone`.
pSpecSize	Pointer to the FFT specification structure size value.
pSpecBufferSize	Pointer to the buffer size value for FFT initialization function.
pBufferSize	Pointer to the size value of the FFT external work buffer.

Description

These functions compute the following:

- the size of the FFT specification structure. Computed value stored in *pSpecSize*.
- the work buffer size for the FFT structure initialization functions `ippsFFTInit_R` and `ippsFFTInit_C`. Computed value is stored in *pSpecBufferSize*.
- the size of the FFT work buffer for the different flavors of `ippsFFTFwd` and `ippsFFTInv`. Computed value is stored in *pBufferSize*.

The suffix after the function name indicates the flavors of the FFT functions: `ippsFFTGetSize_C` is for complex flavors and `ippsFFTGetSize_R` is for real flavors.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsFftOrderErr`	Indicates an error when the *order* value is incorrect.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.

See Also

Special Arguments

FFTInit_R, FFTInit_C Initializes the FFT specification structure for real and complex signals.

FFTFwd_CToC

Computes the forward fast Fourier transform (FFT) of a complex signal.

Syntax

Case 1: Not-in-place operation on real data type

```
IppStatus ippsFFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
pDstRe, Ipp32f* pDstIm, const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
pDstRe, Ipp64f* pDstIm, const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 2: Not-in-place operation on complex data type

```
IppStatus ippsFFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
```

Case 3: In-place operation on real data type.

```
IppStatus ippsFFTFwd_CToC_32f_I(Ipp32f* pSrcDstRe, Ipp32f* pSrcDstIm, const
IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64f_I(Ipp64f* pSrcDstRe, Ipp64f* pSrcDstIm, const
IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 4: In-place operation on complex data type.

```
IppStatus ippsFFTFwd_CToC_32fc_I(Ipp32fc* pSrcDst, const IppsFFTSpec_C_32fc* pFFTSpec,
Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64fc_I(Ipp64fc* pSrcDst, const IppsFFTSpec_C_64fc* pFFTSpec,
Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pSrcDst	Pointer to the input and output array containing complex values (for the in-place operation).
pSrcDstRe	Pointer to the input and output array containing real parts of the signal (for the in-place operation).
pSrcDstIm	Pointer to the input and output array containing imaginary parts of the signal (for the in-place operation).

pBuffer	Pointer to the external work buffer.

Description

This function computes the forward FFT of a complex signal according to the following *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions, you need to initialize the FFT specification structure using the `ippsFFTInit_C` function.

The functions using the complex data type, for example with the `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*. Their in-place flavors use the complex array *pSrcDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, for example, with the `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively. Their in-place flavors use separate real and imaginary arrays *pSrcDstRe* and *pSrcDstIm*, respectively.

To avoid memory allocation within the functions, you can use this function with the external work buffer *pBuffer*. Once the work buffer is allocated, it can be used for all following calls of the functions computing FFT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of the external buffer must be previously computed by the function `ippsFFTGetBufSize_C` or `ippsFFTGetSize_C`.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

The length of the FFT must be a power of 2.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers with exception of *pBuffer* is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.

Example

The code example below demonstrates how to use the `ippsFFTGetSize`, `ippsFFTInit`, and `ippsFFTFwd_CToC` functions.

```
void ippsFFT_32fc_example()
{
    Ipp32fc Src[32] = {
        {0.0, 1.0}, {2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0},
        {0.0, 1.0}, {2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0},
        {0.0, 1.0}, {2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0},
        {0.0, 1.0}, {2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0},
        {0.0, 1.0}, {2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0},
        {0.0, 1.0}, {2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0},
        {0.0, 1.0}, {2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0},
        {0.0, 1.0}, {2.0, 3.0}, {4.0, 5.0}, {6.0, 7.0}
    };
    Ipp32fc Dst[32];

    int FFTOrder = 5;
```

```
    IppsFFTSpec_C_32fc *pSpec = 0;

    Ipp8u *pMemSpec = 0;
    Ipp8u *pMemInit = 0;
    Ipp8u *pMemBuffer = 0;

    int sizeSpec = 0;
    int sizeInit = 0;
    int sizeBuffer = 0;

    int flag = IPP_FFT_NODIV_BY_ANY;

    /// get sizes for required buffers
    ippsFFTGetSize_C_32fc(FFTOrder, flag, ippAlgHintNone, &sizeSpec, &sizeInit, &sizeBuffer);

    /// allocate memory for required buffers
    pMemSpec = (Ipp8u*) ippMalloc(sizeSpec);

    if (sizeInit > 0)
    {
        pMemInit = (Ipp8u*) ippMalloc(sizeInit);
    }

    if (sizeBuffer > 0)
    {
        pMemBuffer = (Ipp8u*) ippMalloc(sizeBuffer);
    }

    /// initialize FFT specification structure
    ippsFFTInit_C_32fc(&pSpec, FFTOrder, flag, ippAlgHintNone, pMemSpec, pMemInit);

    /// free initialization buffer
    if (sizeInit > 0)
    {
        ippFree(pMemInit);
    }

    /// perform forward FFT
    ippsFFTFwd_CToC_32fc(Src, Dst, pSpec, pMemBuffer);

    /// ...

    /// free buffers
    if (sizeBuffer > 0)
    {
        ippFree(pMemBuffer);
    }

    ippFree(pMemSpec);
}
```

Result:

```
Dst ->      {  96.0, 128.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 },
           {   0.0,   0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 },
            { -64.0,   0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 },
            {   0.0,   0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 },
            { -32.0, -32.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 },
```

```
            {   0.0,   0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 },
            {   0.0, -64.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 },
            {   0.0,   0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0 }
```

See Also

Integer Scaling

FFTInit_R, FFTInit_C Initializes the FFT specification structure for real and complex signals.

FFTGetSize_R, FFTGetSize_C Computes sizes of the FFT specification structure and required working buffers.

FFTInv_CToC

Computes the inverse fast Fourier transform (FFT) of a complex signal.

Syntax

Case 1: Not-in-place operation on real data type

```
IppStatus ippsFFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
pDstRe, Ipp32f* pDstIm, const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
pDstRe, Ipp64f* pDstIm, const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 2: Not-in-place operation on complex data type

```
IppStatus ippsFFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
```

Case 3: In-place operation on real data type

```
IppStatus ippsFFTInv_CToC_32f_I(Ipp32f* pSrcDstRe, Ipp32f* pSrcDstIm, const
IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64f_I(Ipp64f* pSrcDstRe, Ipp64f* pSrcDstIm, const
IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 4: In-place operation on complex data type

```
IppStatus ippsFFTInv_CToC_32fc_I(Ipp32fc* pSrcDst, const IppsFFTSpec_C_32fc* pFFTSpec,
Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64fc_I(Ipp64fc* pSrcDst, const IppsFFTSpec_C_64fc* pFFTSpec,
Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.

pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pSrcDst	Pointer to the input and output array containing complex values (for the in-place operation).
pSrcDstRe	Pointer to the input and output array containing real parts of the signal(for the in-place operation).
pSrcDstIm	Pointer to the input and output array containing imaginary parts of the signal(for the in-place operation).
pBuffer	Pointer to the external work buffer.

Description

This function computes the inverse FFT of a complex signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The FFT specification structure must be initialized by the ippsFFTInit_C function beforehand.

The function flavors using the complex data type, for example with the 32fc suffixes, process the input complex array *pSrc* and store the result in *pDst*. Their in-place flavors use the complex array *pSrcDst*.

The function flavors using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, for example with the 32f suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively. Their in-place flavors uses separate real and imaginary arrays *pSrcDstRe* and *pSrcDstIm*, respectively.

The function may be used with the external work buffer *pBuffer* to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing FFT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of the external buffer must be previously computed by the ippsFFTGetSize_C function.

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

The length of the FFT must be a power of 2.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers with exception of *pBuffer* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

FFTFwd_RToPack, FFTFwd_RToPerm, FFTFwd_RToCCS

Computes the forward or inverse fast Fourier transform (FFT) of a real signal.

Syntax

Case 1: Not-in-place operation, result in `Pack` Format

```
IppStatus ippsFFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 2: In-place operation, result in `Pack` Format.

```
IppStatus ippsFFTFwd_RToPack_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f* pFFTSpec,
Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPack_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f* pFFTSpec,
Ipp8u* pBuffer);
```

Case 3: Not-in-place operation, result in `Perm` Format

```
IppStatus ippsFFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 4: In-place operation, result in `Perm` Format.

```
IppStatus ippsFFTFwd_RToPerm_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f* pFFTSpec,
Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f* pFFTSpec,
Ipp8u* pBuffer);
```

Case 5: Not-in-place operation, result in `CCS` Format

```
IppStatus ippsFFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 6: In-place operation, result in `CCS` Format.

```
IppStatus ippsFFTFwd_RToCCS_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f* pFFTSpec,
Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f* pFFTSpec,
Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array.
pDst	Pointer to the output array containing packed complex values.
pSrcDst	Pointer to the input and output arrays for the in-place operation.

pBuffer	Pointer to the external work buffer.

Description

These functions compute the forward FFT of a real signal and store the result in Pack, Perm, or CCS packed formats respectively. The transform is performed in accordance with the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure must be initialized by the corresponding flavors of ippsFFTInit_R. The length of the FFT must be a power of 2.

The function may be used with the external work buffer *pBuffer*to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing FFT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of the external buffer must be previously computed by the ippsFFTGetSize_R function.

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

ippsFFTFwd_RToPack. This function computes the forward FFT and stores the result in Pack format.

ippsFFTFwd_RToPerm. This function computes the forward FFT and stores the result in Perm format.

ippsFFTFwd_RToCCS. This function computes the forward FFT and stores the result in CCS format.

Tables "Arrangement of Forward Fourier Transform Results in Packed Formats - Even Length" and "Forward Fourier transform Result Representation in Packed Formats - Odd Length" show how the output results are arranged in the packed formats.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers with exception of *pBuffer* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

FFTFwd_RToCCS.c

FFTInv_PackToR, FFTInv_PermToR, FFTInv_CCSToR

Computes the inverse fast Fourier transform (FFT) of a real signal.

Syntax

Case 1: Not-in-place operation on input data in Pack format

```
IppStatus ippsFFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 2: In-place operation on input data in `Pack` format

```
IppStatus ippsFFTInv_PackToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f* pFFTSpec,
Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PackToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f* pFFTSpec,
Ipp8u* pBuffer);
```

Case 3: Not-in-place operation on input data in `Perm` format

```
IppStatus ippsFFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 4: In-place operation on input data in `Perm` format

```
IppStatus ippsFFTInv_PermToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f* pFFTSpec,
Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f* pFFTSpec,
Ipp8u* pBuffer);
```

Case 5 Not-in-place operation on input data in `CCS` format

```
IppStatus ippsFFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

Case 6: In-place operation on input data in `CCS` format

```
IppStatus ippsFFTInv_CCSToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f* pFFTSpec,
Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f* pFFTSpec,
Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pFFTSpec	Pointer to the FFT specification structure.
pSrc	Pointer to the input array.
pDst	Pointer to the output array containing real values.
pSrcDst	Pointer to the input and output for the in-place operation.
pBuffer	Pointer to the external work buffer.

Description

The function may be used with the external work buffer *pBuffer* to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing FFT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of the external buffer must be previously computed by the function ippsFFTGetSize_R.

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

ippsFFTInv_PackToR. This function computes the inverse FFT of input data in Pack format.

ippsFFTInv_PermToR. This function computes the inverse FFT of input data in Perm format.

ippsFFTInv_CCSToR. This function computes the inverse FFT of input data in CCS format.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers with exception of *pBuffer* is *NULL*.
ippStsContextMatchErr	Indicates an error when the specification identifier *pFFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.

Discrete Fourier Transform Functions

The functions described in this section compute the forward and inverse discrete Fourier transform of real and complex signals. The DFT is less efficient than the fast Fourier transform, however the length of the vector transformed by the DFT can be arbitrary.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method. The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in Pack, Perm, or CCS formats.

To use the DFT functions, you should initialize the specification structure which contains such data as tables of twiddle factors. Use the ippsDFTInit_R and ippsDFTInit_C functions to initialize the specification structure both for forward and inverse transforms. Before using these functions, compute the size of the DFT specification structure using the ippsDFTGetSize_R or ippsDFTGetSize_C functions and allocate memory for the structure beforehand.

You can speed up the DFT by using an external buffer. The use of external buffer can improve performance by avoiding allocation and deallocation of internal buffers and storing data in cache. The size of the external buffer is returned by the ippsDFTInit_R and ippsDFTInit_C functions.

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.

DFTInit_R, DFTInit_C

Initializes the DFT specification structure for real and complex signals.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsDFTInit_R_32f(int length, int flag, IppHintAlgorithm hint,
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pMemInit);
```

```
IppStatus ippsDFTInit_R_64f(int length, int flag, IppHintAlgorithm hint,
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pMemInit);
```

Case 2: Operation on complex signal

```
IppStatus ippsDFTInit_C_32fc(int length, int flag, IppHintAlgorithm hint,
IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pMemInit);
```

```
IppStatus ippsDFTInit_C_32f(int length, int flag, IppHintAlgorithm hint,
IppsDFTSpec_C_32f* pDFTSpec, Ipp8u* pMemInit);
```

```
IppStatus ippsDFTInit_C_64fc(int length, int flag, IppHintAlgorithm hint,
IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pMemInit);
```

```
IppStatus ippsDFTInit_C_64f(int length, int flag, IppHintAlgorithm hint,
IppsDFTSpec_C_64f* pDFTSpec, Ipp8u* pMemInit);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

length	Length of the DFT transform.
flag	Specifies the result normalization method. The values for the *flag* argument are described in the section Flag and Hint Arguments.
hint	This parameter is deprecated. Set the value to `ippAlgHintNone`.
pDFTSpec	Pointer to the DFT specification structure to be initialized.
pMemInit	Pointer to the temporary work buffer.

Description

These functions 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.

Before calling these functions the memory must be allocated for the DFT specification structure and the temporary work buffer (if it is required). The size of the DFT specification structure and the work buffer must be computed by the functions `ippsDFTGetSize_R` or `ippsDFTGetSize_C`.

If the work buffer is not used, the parameter *pMemInit* can be `NULL`. If the work buffer is used, the parameter *pMemInit* cannot be `NULL`. After initialization is done, the temporary work buffer can be freed.

ippsDFTInit_R function initializes the real DFT specification structure.

ippsDFTInit_C function initializes the complex DFT specification structure.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsFftFlagErr	Indicates an error when the *flag* value is incorrect.
ippStsFftOrderErr	Indicates an error when the memory needed to calculate the *length* value of the DFT transform exceeds the limit.
ippStsSizeErr	Indicates an error when *length* is less than, or equal to 0.

DFTGetSize_R, DFTGetSize_C

Computes sizes of the DFT work buffer and required working buffers.

Syntax

Case 1: Operation on real signal

```
IppStatus ippsDFTGetSize_R_32f(int length, int flag, IppHintAlgorithm hint, int*
pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

```
IppStatus ippsDFTGetSize_R_64f(int length, int flag, IppHintAlgorithm hint, int*
pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

Case 2: Operation on complex signal

```
IppStatus ippsDFTGetSize_C_32fc(int length, int flag, IppHintAlgorithm hint, int*
pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

```
IppStatus ippsDFTGetSize_C_32f(int length, int flag, IppHintAlgorithm hint, int*
pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

```
IppStatus ippsDFTGetSize_C_64fc(int length, int flag, IppHintAlgorithm hint, int*
pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

```
IppStatus ippsDFTGetSize_C_64f(int length, int flag, IppHintAlgorithm hint, int*
pSizeSpec, int* pSizeInit, int* pSizeBuf);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

length	Length of the DFT transform.
flag	Specifies the result normalization method. The values for the *flag* argument are described in the section Flag and Hint Arguments.
hint	This parameter is deprecated. Set the value to ippAlgHintNone.

pSizeSpec	Pointer to the DFT specification structure size value.
pSizeInit	Pointer to the buffer size value for DFT initialization functions.
pSizeBuf	Pointer to the size value of the DFT external work buffer.

Description

These functions compute the size of DFT specification structure, the work buffer size for the DFT structure initialization functions `ippsDFTInit_R` and `ippsDFTInit_C`, and size of the DFT work buffer for different flavors of `ippsDFTFwd` and `ippsDFTInv`. Their values in bytes are stored in *pSpecSize*, *pSizeInit*, and *pSizeBuf* respectively.

ippsDFTGetSize_R function is used for real flavors of the DFT functions.

ippsDFTGetSize_C function is used for complex flavors of the DFT functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is `NULL`.
ippStsFftFlagErr	Indicates an error when the *flag* value is incorrect.
ippStsFftOrderErr	Indicates an error when the memory needed to calculate the *length* value of the DFT transform exceeds the limit.
ippStsSizeErr	Indicates an error when *length* is less than, or equal to 0.

DFTFwd_CToC

Computes the forward discrete Fourier transform of a complex signal.

Syntax

Case 1: Operation on real data type

```
IppStatus ippsDFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f* pDFTSpec, Ipp8u* pBuffer);
```

Case 2: Operation on complex data type

```
IppStatus ippsDFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Flavors declared in `ipps.h`:

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing complex values.
pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pBuffer	Pointer to the work buffer.

Description

These functions compute the forward DFT according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

The functions operating on the complex data type process the input complex array *pSrc* and store the result in *pDst*.

The functions operating on the real data type (processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts) store the result separately in *pDstRe* and *pDstIm*, respectively.

The function can be used with the external work buffer *pBuffer*2) Data vectors for these functions must be aligned to an appropriate number of bytes that is determined by the SIMD width that is supported by the customer's platform - it is reccomended to use ippMalloc function for such purpose as it guarantees such alignment.

Required buffer size must be computed by the corresponding function `ippsDFTGet BufSize_C` prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

> **NOTE**
> Data vectors for these functions must be aligned to an appropriate number of bytes that is determined by the SIMD width that is supported by the customer's platform - use `ippMalloc` function for such alignment.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where *k* is the index of elements in the frequency domain, *n* is the index of elements in the time domain, *N* is the input signal *len*, and *A* is a multiplier defined by *flag*. Also, *x(n)* is *pSrc[n]* and *X(k)* is *pDst[k]*.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or

Optimization Notice
effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers with exception of *pBuffer* is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.

DFTInv_CToC

Computes the inverse discrete Fourier transform of a complex signal.

Syntax

Case 1: Operation on real data type

```
IppStatus ippsDFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f* pDFTSpec, Ipp8u* pBuffer);
```

Case 2: Operation on complex data type

```
IppStatus ippsDFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Flavors declared in `ipps.h`:

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing complex values.

pDst	Pointer to the output array containing complex values.
pSrcRe	Pointer to the input array containing real parts of the signal.
pSrcIm	Pointer to the input array containing imaginary parts of the signal.
pDstRe	Pointer to the output array containing real parts of the signal.
pDstIm	Pointer to the output array containing imaginary parts of the signal.
pBuffer	Pointer to the work buffer.

Description

These functions compute the inverse DFT according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, for example 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, for example with `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

The function can be used with the external work buffer *pBuffer* to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

Required buffer size must be computed by the corresponding function `ippsDFTGet BufSize_C` prior to using DFT computation functions.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers with exception of *pBuffer* is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
`ippStsMemAllocErr`	Indicates an error when no memory is allocated.
`ippStsFftFlagErr`	Indicates an error when the *flag* value is incorrect.

DFTFwd_RToPack, DFTFwd_RToPerm, DFTFwd_RToCCS

Computes the forward discrete Fourier transform of a real signal.

Syntax

Case 1: Result in `Pack` format

```
IppStatus ippsDFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

supported value for <*len*>: integer in the range [2, 64].

Case 2: Result in `Perm` format

```
IppStatus ippsDFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

Case 3: Result in `CCS` format

```
IppStatus ippsDFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Flavors declared in `ipps.h`:

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing real values .
pDst	Pointer to the output array containing packed complex values.
pBuffer	Pointer to the work buffer.

Description

These functions compute the forward DFT of a real signal. The result of the forward transform (that is 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. Tables show how the output results are arranged in the packed formats.

These functions compute the forward DFT according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

These functions can be used with the external work buffer *pBuffer* to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

ippsDFTFwd_RToPack. These functions compute the forward DFT and stores the result in Pack format.

ippsDFTFwd_RToPerm. These functions compute the forward DFT and stores the result in Perm format.

ippsDFTFwd_RToCCS. These functions compute the forward DFT and stores the result in CCS format.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers with exception of *pBuffer* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsFftFlagErr	Indicates an error when the *flag* value is incorrect.

DFTInv_PackToR, DFTInv_PermToR, DFTInv_CCSToR

Computes the inverse discrete Fourier transform of a real signal.

Syntax

Case 1: Input data in Pack format

```
IppStatus ippsDFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

Case 2: Input data in Perm format

```
IppStatus ippsDFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

Case 3: Input data in CCS format

```
IppStatus ippsDFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

Include Files

`16s_Sfs` flavors: `ipps.h`

Domain Dependencies

`16s_Sfs`:

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pDFTSpec	Pointer to the DFT specification structure.
pSrc	Pointer to the input array containing packed complex values.
pDst	Pointer to the output array containing real values.
pBuffer	Pointer to the work buffer.

Description

These functions compute the inverse DFT of a real signal. The input data (that is in the frequency-domain) are represented in several possible packed formats: Pack, Perm, or CCS. Tables show how the input data can be represented in the packed formats.

The function can be used with the external work buffer *pBuffer* to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

ippsDFTInv_PackToR. This function computes the inverse DFT for input data in `Pack` format.

ippsDFTInv_PermToR. This function computes the inverse DFT for input data in `Perm` format.

ippsDFTInv_CCSToR. This function computes the inverse DFT for input data in `CCS` format.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers with exception of *pBuffer* is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pDFTSpec* is incorrect.
ippStsMemAllocErr	Indicates an error when no memory is allocated.
ippStsFftFlagErr	Indicates an error when the *flag* value is incorrect.

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, for example the dual-tone multi frequency signal detection, work faster with such functions.

Goertz

Computes the discrete Fourier transform for a given frequency for a single signal.

Syntax

```
IppStatus ippsGoertz_32f(const Ipp32f* pSrc, int len, Ipp32fc* pVal, Ipp32f rFreq);
IppStatus ippsGoertz_64f(const Ipp64f* pSrc, int len, Ipp64fc* pVal, Ipp64f rFreq);
IppStatus ippsGoertz_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pVal, Ipp32f rFreq);
IppStatus ippsGoertz_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pVal, Ipp64f rFreq);
IppStatus ippsGoertz_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16sc* pVal, Ipp32f rFreq,
int scaleFactor);
IppStatus ippsGoertz_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pVal, Ipp32f
rFreq, int scaleFactor);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the input data vector.
len	Number of elements in the vector.
pVal	Pointer to the output DFT value.

rFreq	Single relative frequency value [0, 1.0).
scaleFactor	Scale factor, refer to Integer Scaling.

Description

This function computes a DFT for an input *len*-length vector *pSrc* for a given frequency *rFreq*, and stores the result in *pVal*.

ippsGoertzQ15. This function operates with relative frequency in Q15 format. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 1.0).

The functionality of the Goertzel algorithm can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where *k*/*N* is the normalized *rFreq* value for which the DFT is computed.

Return Values

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

Example

The example below illustrates the use of Goertzel functions for selecting the magnitudes of a given frequency when computing DFTs.

```
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)
```

Discrete Cosine Transform Functions

This section describes the functions that compute the discrete cosine transform (DCT) of a signal. DCT functions used in the Intel IPP signal processing data-domain implement the modified computation algorithm proposed in [Rao90].

DCTFwdInit

Initializes the forward discrete cosine transform structure.

Syntax

```
IppStatus ippsDCTFwdInit_32f(IppsDCTFwdSpec_32f** ppDCTSpec, int len, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

```
IppStatus ippsDCTFwdInit_64f(IppsDCTFwdSpec_64f** ppDCTSpec, int len, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

ppDCTSpec	Double pointer to the forward DCT specification structure to be created.
len	Number of samples in the DCT.
hint	This parameter is deprecated. Set the value to `ippAlgHintNone`.
pSpec	Pointer to the area for the DCT specification structure.
pSpecBuffer	Pointer to the additional work buffer, can be `NULL`.

Description

This function initializes the forward DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

Before calling this function the memory must be allocated for the DCT specification structure and the work buffer (if it is required). The size of the DCT specification structure and the work buffer must be computed by the function `ippsDCTFwdGetSize` beforehand.

If the work buffer is not used, the parameter *pSpecBuffer* can be `NULL`. If the working buffer is used, the parameter *pSpecBuffer* must not be `NULL`.

Return Values

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

DCTInvInit

Initializes the inverse discrete cosine transform structure.

Syntax

```
IppStatus ippsDCTInvInit_32f(IppsDCTInvSpec_32f** ppDCTSpec, int len, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

```
IppStatus ippsDCTInvInit_64f(IppsDCTInvSpec_64f** ppDCTSpec, int len, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

ppDCTSpec	Double pointer to the inverse DCT specification structure to be created.
len	Number of samples in the DCT.
hint	This parameter is deprecated. Set the value to `ippAlgHintNone`.
pSpec	Pointer to the area for the DCT specification structure.
pSpecBuffer	Pointer to the work buffer, can be `NULL`.

Description

This function initializes in the buffer *pSpec* the inverse DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

Before calling this function the memory must be allocated for the DCT specification structure and the work buffer (if it is required). The size of the DFT specification structure and the work buffer must be computed by the function `ippsDCTInvGetSize`.

If the work buffer is not used, the parameter *pSpecBuffer* can be `NULL`. If the working buffer is used, the parameter *pSpecBuffer* must not be `NULL`.

Return Values

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

DCTFwdGetSize

Computes the size of all buffers required for the forward DCT.

Syntax

```
IppStatus ippsDCTFwdGetSize_32f(int len, IppHintAlgorithm hint, int* pSpecSize, int*
pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsDCTFwdGetSize_64f(int len, IppHintAlgorithm hint, int* pSpecSize, int*
pSpecBufferSize, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

len	Number of samples in the DCT.
hint	This parameter is deprecated. Set the value to `ippAlgHintNone`.
pSpecSize	Pointer to the size of the forward DCT specification structure.
pSpecBufferSize	Pointer to the size of the work buffer for the initialization function.
pBufferSize	Pointer to the size of the forward DCT work buffer.

Description

This function computes the size *pSpecSize* for the forward DCT structure with the following parameters: the transform *len*, and the specific code *hint*. Additionally the function computes the size *pSpecBufferSize* of the work buffer for the initialization function `ippsDCTFwdInit` , and the size *pBufferSize* of the work buffer for the function `ippsDCTFwd` .

The function `ippsDCTFwdGetSize` should be called prior to them.

Return Values

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

DCTInvGetSize

Computes the size of all buffers required for the inverse DCT.

Syntax

```
IppStatus ippsDCTInvGetSize_32f(int len, IppHintAlgorithm hint, int* pSpecSize, int*
pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsDCTInvGetSize_64f(int len, IppHintAlgorithm hint, int* pSpecSize, int*
pSpecBufferSize, int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

len	Number of samples in the DCT.
hint	This parameter is deprecated. Set the value to `ippAlgHintNone`.
pSpecSize	Pointer to the size of the forward DCT specification structure.
pSpecBufferSize	Pointer to the size of the work buffer for the initialization function.
pBufferSize	Pointer to the size of the forward DCT work buffer.

Description

This function computes in bytes the size *pSpecSize* of the external buffer for the inverse DCT structure with the following parameters: the transform *len*, and the specific code *hint*. Additionally the function computes the size *pSpecBufferSize* of the work buffer for the initialization function `ippsDCTInvInit` and the size *pBufferSize* of the work buffer for the function `ippsDCTInv`.

The function `ippsDCTInvGetSize` must be called prior to them.

Return Values

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

DCTFwd

Computes the forward discrete cosine transform of a signal.

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, const IppsDCTFwdSpec_32f*
pDCTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDCTFwd_64f(const Ipp64f* pSrc, Ipp64f* pDst, const IppsDCTFwdSpec_64f*
pDCTSpec, Ipp8u* pBuffer);
```

Case 2: In-place operation

```
IppStatus ippsDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsDCTFwdSpec_32f* pDCTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsDCTFwd_64f_I(Ipp64f* pSrcDst, const IppsDCTFwdSpec_64f* pDCTSpec, Ipp8u*
pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: ippcore.lib, ippvm.lib

Parameters

pDCTSpec	Pointer to the forward DCT specification structure.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
pBuffer	Pointer to the external work buffer.

Description

This function computes the forward discrete cosine transform (DCT) of the source signal *pSrc* (*pSrcDst* for in-place operations) in accordance with the specification structure *pDCTSpec* that must be initialized by calling ippsDCTFwdInit beforehand. The result in stored in the *pDst* (*pSrcDst* for in-place operations).

If *len* is a power of 2, the function use an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *len*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT, *N = len*,

$$C(k) = \frac{1}{\sqrt{N}} \text{ for } k = 0, \quad C(k) = \frac{\sqrt{2}}{\sqrt{N}} \text{ for } k > 0;$$

x(n) is *pSrc*[n] and *y*(k) is *pDst*[k].

The forward DCT is defined by the formula:

$$y(k) = C(k) \sum_{n=0}^{N-1} x(n) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

The function may be used with the external work buffer *pBuffer* to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DCT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of this buffer must be computed previously using ippsDCTFwdGetSize.

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the *pDCTSpec*, *pSrc*, *pDst*, *pSrcDst* pointers is NULL.
ippStsContextMatchErr	Indicates an error if the specification identifier *pDCTSpec* is incorrect.

`ippStsMemAllocErr`	Indicates an error if memory allocation fails.

DCTInv

Computes the inverse discrete cosine transform of a signal.

Syntax

Case 1: Not-in-place operation

```
IppStatus ippsDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, const IppsDCTInvSpec_32f*
pDCTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDCTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, const IppsDCTInvSpec_64f*
pDCTSpec, Ipp8u* pBuffer);
```

Case 2: In-place operation

```
IppStatus ippsDCTInv_32f_I(Ipp32f* pSrcDst, const IppsDCTInvSpec_32f* pDCTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsDCTInv_64f_I(Ipp64f* pSrcDst, const IppsDCTInvSpec_64f* pDCTSpec, Ipp8u*
pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pDCTSpec	Pointer to the inverse DCT specification structure.
pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
pSrcDst	Pointer to the source and destination vector for in-place operations.
pBuffer	Pointer to the external work buffer.

Description

This function computes the inverse discrete cosine transform (DCT) of the source signal *pSrc* (*pSrcDst* for in-place operations) in accordance with the specification structure *pDCTSpec* that must be initialized by calling `ippsDCTInvInit` beforehand. The result in stored in the *pDst* (*pSrcDst* for in-place operations).

If *len* is a power of 2, the functions use an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *len*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT, $N = len$,

$$C(k) = \frac{1}{\sqrt{N}} \text{ for } k = 0, \quad C(k) = \frac{\sqrt{2}}{\sqrt{N}} \text{ for } k > 0;$$

`x(n)` is `pDst[n]` and `y(k)` is `pSrc[k]`.

The inverse DCT is defined by the formula:

$$x(n) = \sum_{k=0}^{N-1} C(k)y(k) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

The function may be used with the external work buffer `pBuffer` to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DCT. As internal allocation of memory is too expensive operation and depends on operating system and/or runtime libraries used - the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of this buffer must be computed previously using `ippsDCTInvGetSize`.

If the external buffer is not specified (`pBuffer` is set to `NULL`), then the function itself allocates the memory needed for operation.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the `pDCTSpec`, `pSrc`, `pDst`, `pSrcDst` pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the specification identifier `pDCTSpec` is incorrect.
`ippStsMemAllocErr`	Indicates an error if memory allocation fails.

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.

HilbertGetSize

Computes the size of the Hilbert transform structure and temporary work buffer.

Syntax

```
IppStatus ippsHilbertGetSize_32f32fc(int length, IppHintAlgorithm hint, int* pSpecSize,
int* pBufferSize);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

length	Number of samples in the Hilbert transform.
hint	Option to select the algorithmic implementation of the transform function (DFT). The values for the *hint* argument are described in Flag and Hint Arguments.
pSpecSize	Pointer to the size, in bytes, of the Hilbert context structure.
pBufferSize	Pointer to the size, in bytes, of the work buffer.

Description

This function computes the size of the Hilbert specification structure and temporary work buffer for the `ippsHilbert` function.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *length* is less than 1.

See Also

HilbertMODIFIED API. Computes an analytic signal using the Hilbert transform.

HilbertInit

Initializes the Hilbert transform structure.

Syntax

```
IppStatus ippsHilbertInit_32f32fc(int length, IppHintAlgorithm hint, IppsHilbertSpec*
pSpec, Ipp8u* pBuffer);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

length	Number of samples in the Hilbert transform.
hint	Option to select the algorithmic implementation of the transform function (DFT). The values for the *hint* argument are described in Flag and Hint Arguments.
pSpec	Pointer to the Hilbert context structure.
pBuffer	Pointer to the work buffer.

Description

This function 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 using the Hilbert transform function `ippsHilbert`.

Return Values

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

See Also

HilbertMODIFIED API. Computes an analytic signal using the Hilbert transform.

Hilbert

MODIFIED API. Computes an analytic signal using the Hilbert transform.

Syntax

```
IppStatus ippsHilbert_32f32fc(const Ipp32f* pSrc, Ipp32fc* pDst, IppsHilbertSpec*
pSpec, Ipp8u* pBuffer);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSpec	Pointer to the Hilbert specification structure.
pSrc	Pointer to the vector containing original real data.
pDst	Pointer to the output array containing complex data.
pBuffer	Pointer to the work buffer.

Description

The ippsHilbert function computes a complex analytic signal *pDst*, which contains the original real signal *pSrc* as its real part and computed Hilbert transform as its imaginary part. The Hilbert transform is performed according to the *pSpec* specification parameters: the number of samples *len*, and the specific code *hint*. The input data is zero-padded or truncated to the size of *len* as appropriate.

Before using this function, you need to compute the size of the work buffer and specification structure using the HilbertGetSize function and initialize the structure using HilbertInit.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error when the specification identifier *pSpec* is incorrect.

Example

The example below shows how to initialize the specification structure and use the function `ippsHilbert_32f32fc`.

```
IppStatus hilbert( )
{
   Ipp32f x[10];
   Ipp32fc y[10];
   int n;
   IppStatus status;
   IppsHilbertSpec* pSpec;
   Ipp8u* pBuffer;
   int sizeSpec, sizeBuf;

   status = ippsHilbertGetSize_32f32fc(10, ippAlgHintNone, &sizeSpec, &sizeBuf);

   pSpec = (IppsHilbertSpec*)ippMalloc(sizeSpec);
   pBuffer = (Ipp8u*)ippMalloc(sizeBuf);

   status = ippsHilbertInit_32f32fc(10, ippAlgHintNone, pSpec, pBuffer);

   for (n = 0; n < 10; n ++) {
      x[n] = (Ipp32f)cos(IPP_2PI * n * 2 / 9);
   }

   status = ippsHilbert_32f32fc(x, y, pSpec, pBuffer);

   ippsMagnitude_32fc((Ipp32fc*)y, x, 5);

   ippFree(pSpec);
   ippFree(pBuffer);

   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)
```

See Also

HilbertGetSize Computes the size of the Hilbert transform structure and temporary work buffer.
HilbertInit Initializes the Hilbert transform structure.

Wavelet Transform Functions

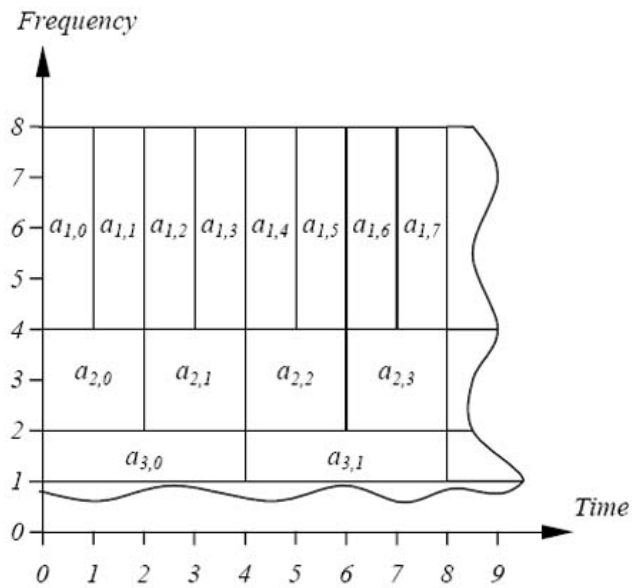
This section describes the wavelet transform functions implemented in Intel IPP.

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 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 Intel IPP and referred to as the discrete wavelet transform (DWT).

Wavelet Decomposition Coefficients in Time-Frequency Domain



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 shows the diagram of the forward DWT which allows to switch to time-frequency representation shown in Figure above. The diagram includes three levels of decomposition. Figure shows the corresponding procedure of signal reconstruction based on the elementary one-level inverse transform.

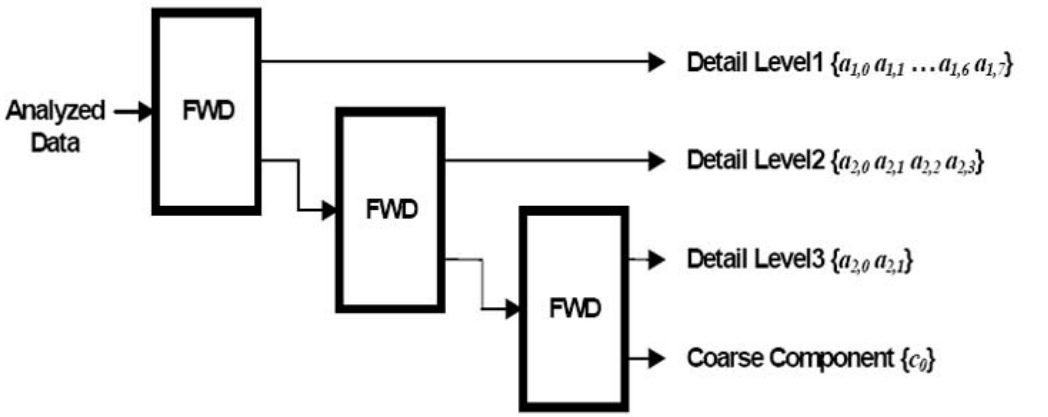
The implementation of discrete multi-scale transforms is based on the use of interpolation and decimation filters with the resampling factor 2. The basis of the multi-scale signal decomposition and reconstruction functions uniquely defines the filter parameters. The Intel IPP multi-scale transform functions use filters with finite impulse response.

The Primitives contains two sets of functions.

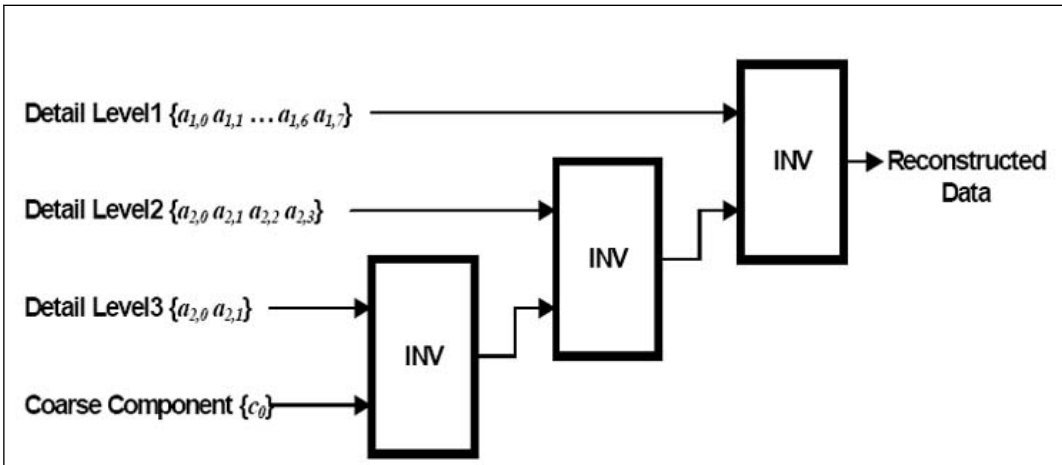
- Transforms designed for fixed filter banks. These transforms yield the highest performance.

- 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.

Three-Level Discrete Wavelet Decomposition



Three-Level Discrete Wavelet Reconstruction



Transforms for Fixed Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for fixed filter banks.

WTHaarFwd, WTHaarInv

Performs forward or inverse single-level discrete wavelet Haar transforms.

Syntax

Case 1: Forward transform

```
IppStatus ippsWTHaarFwd_32f(const Ipp32f* pSrc, int len, Ipp32f* pDstLow, Ipp32f*
pDstHigh);
```

```
IppStatus ippsWTHaarFwd_64f(const Ipp64f* pSrc, int len, Ipp64f* pDstLow, Ipp64f*
pDstHigh);

IppStatus ippsWTHaarFwd_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pDstLow, Ipp16s*
pDstHigh, int scaleFactor);
```

Case 2: Inverse transform

```
IppStatus ippsWTHaarInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, Ipp32f*
pDst, int len);

IppStatus ippsWTHaarInv_64f(const Ipp64f* pSrcLow, const Ipp64f* pSrcHigh, Ipp64f*
pDst, int len);

IppStatus ippsWTHaarInv_16s_Sfs(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh, Ipp16s*
pDst, int len, int scaleFactor);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pSrc	Pointer to the source vector for forward transform.
len	Number of elements in the vector.
pDstLow	Pointer to the array with the coarse “low frequency” components of the output for forward transform.
pDstHigh	Pointer to the array with the detail “high frequency” components of the output for forward transform.
pSrcLow	Pointer to the array with the coarse “low frequency” components of the input for inverse transform.
pSrcHigh	Pointer to the array with the detail “high frequency” components of the input for inverse transform.
pDst	Pointer to the array with the output signal for inverse transform.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

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 a 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 [Strang96] 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 = *len*. 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= *len*. 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 [Bris94].

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. Thise function performs the forward single-level discrete Haar transform of a *len*-length signal *pSrc* and stores the decomposed coarse "low-frequency" components in *pDstLow*, and the detail "high-frequency" components in *pDstHigh*.

ippsWTHaarInv. This function 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 *len*-length vector *pDst*.

For more information on wavelet transforms see [Strang96] and [Bris94].

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when the *pDst* or *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than 4 for the function ippsWinBlackmanOpt and less than 3 for all other functions of the family.

Example

The example below illustrates the use of the function ippsWTHaarFwd_32f.

```
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
```

Transforms for User Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for user filter banks.

WTFwdGetSize, WTInvGetSize

Compute the size of the wavelet transform state structures.

Syntax

IppStatus ippsWTFwdGetSize(IppDataType *srcType*, int *lenLow*, int *offsLow*, int *lenHigh*, int *offsHigh*, int* *pStateSize*);

IppStatus ippsWTInvGetSize(IppDataType *dstType*, int *lenLow*, int *offsLow*, int *lenHigh*, int *offsHigh*, int* *pStateSize*);

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

srcType, *dstType*	Data type of the transformed vector.
lenLow	Length of the low-pass filter.

offsLow	Input delay of the low-pass filter.
lenHigh	Length of the high-pass filter.
offsHigh	Input delay of the high-pass filter.
pStateSize	Pointer to the size of the `ippsWTFwd` or `ippsWTInv` state structure, in bytes.

Description

The `ippsWTFwd` and `ippsWTInv` functions compute the size of the `ippsWTFwd` and `ippsWTInv` state structures, in bytes, for the `ippsWTFwdInit` and `ippsWTInvInit` functions, respectively.

For an example on how to use these functions, refer to Wavelet Transforms Example.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *lenLow* or *lenHigh* is less than, or equal to zero.
`ippStsWtOffsetErr`	Indicates an error when the filter delay *offsLow* or *offsHigh* is less than -1.

See Also

WTFwdInit, WTInvInit Initialize the wavelet transform state structures.
WTFwd Computes the forward wavelet transform.
WTInv Computes the inverse wavelet transform.
Wavelet Transforms Example

WTFwdInit, WTInvInit

Initialize the wavelet transform state structures.

Syntax

Case 1: Forward transform

```
IppStatus ippsWTFwdInit_32f(IppsWTFwdState_32f* pState, const Ipp32f* pTapsLow, int
lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTFwdInit_8u32f(IppsWTFwdState_8u32f* pState, const Ipp32f* pTapsLow, int
lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTFwdInit_16s32f(IppsWTFwdState_16s32f* pState, const Ipp32f* pTapsLow,
int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTFwdInit_16u32f(IppsWTFwdState_16u32f* pState, const Ipp32f* pTapsLow,
int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

Case 2: Inverse transform

```
IppStatus ippsWTInvInit_32f(IppsWTInvState_32f* pState, const Ipp32f* pTapsLow, int
lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInit_32f8u(IppsWTInvState_32f8u* pState, const Ipp32f* pTapsLow, int
lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInit_32f16s(IppsWTInvState_32f16s* pState, const Ipp32f* pTapsLow,
int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInit_32f16u(IppsWTInvState_32f16u* pState, const Ipp32f* pTapsLow,
int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pState	Pointer to the initialized forward or inverse wavelet transform state structure.
pTapsLow	Pointer to the vector of low-pass filter taps.
lenLow	Number of taps in the low-pass filter.
offsLow	Input delay (offset) of the low-pass filter.
pTapsHigh	Pointer to the vector of high-pass filter taps.
lenHigh	Number of taps in the high-pass filter.
offsHigh	Input delay (offset) of the high-pass filter.

Description

The ippsWTFwdInit and ippsWTInvInit functions initialize the forward and inverse wavelet transform state structures, respectively, with the following parameters: the low-pass and high-pass filter taps *pTapsLow* and *pTapsHigh*, lengths *lenLow* and *lenHigh*, input additional delays *offsLow* and *offsHigh*.

Application Notes

These functions initialize the wavelet state structure and return the*pState* pointer to it. 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. You can also specify an additional delay *offsLow* and *offsHigh* for each filter. With the adjustable values of delays you can synchronize:

- Group of delays for high-pass and low-pass filters
- Delays between data of different levels in multilevel decomposition and reconstruction algorithms

For more information about using these parameters, see descriptions of the ippsWTFwd and ippsWTInv functions. The minimum allowed value of the additional delay for the forward transform is -1. For the inverse transform the delay values must be greater than, or equal to 0. See descriptions of the ippsWTFwd and ippsWTInv functions 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 initialization stage. To set a non-zero signal prehistory, call the function ippsWTFwdSetDlyLine. When processed an entire limited data set, data extrapolation may be performed both towards the start and the end of the data vector. For

that, the source data and their initial extrapolation are used to form the delay line, the rest of the signal is subdivided into the main block and the signal end. The signal end data and their extrapolation are used to form the last block.

For an example on how to use these functions, refer to Wavelet Transforms Example.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when any of the specified pointers 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.

See Also

WTFwd Computes the forward wavelet transform.
Wavelet Transforms Example

WTFwd

Computes the forward wavelet transform.

Syntax

```
IppStatus ippsWTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDstLow, Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_32f* 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);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrc	Pointer to the vector which holds the input signal for decomposition.
pDstLow	Pointer to the vector which holds output coarse “low frequency” components.
pDstHigh	Pointer to the vector which holds output detail “high frequency” components.
dstLen	Number of elements in the vectors *pDstHigh* and *pDstLow*.
pState	Pointer to the state structure.

Description

This function computes the forward wavelet transform. The function transforms the (2*dstLen)-length source data block pSrc into "low frequency" components pDstLow and "high frequency" components pDstHigh. The transform parameters are specified in the state structure pState.

Before using this function, you need to compute the size of the state structure and work buffer using the WTFwdGetSize_ WTInvGetSize function, and initialize the structure using WTFwdInit WTInvInit.

For an example on how to use this function, refer to Wavelet Transforms Example.

Application Notes

These functions perform the one-level forward discrete multi-scale transform. An equivalent transform diagram is shown in the Figure below. 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, that is the length of input components. The length of the input block must be double the size of each component.

Filter group delays synchronization. Some applications may require synchronization of highpass and lowpass filter time responses. A typical example of this synchronization is synchronizing symmetrical filters of different length.

Below follows an example of bi orthogonal set of spline filters of respective length of 6 and 2:

```
static const float decLow[6] = {     -6.25000000e-002f,      6.25000000e-002f,
5.00000000e-001f,      5.00000000e-001f,      6.25000000e-002f,      -6.25000000e-002f };
```

```
static const float decHigh[2] = {     -5.00000000e-001f,      5.00000000e-001f };
```

In this case the lowpass filter gives a delay two samples longer than the highpass filter, which is exactly what the difference between additional initialization function delays should be. The following values must be selected to ensure minimum common signal delay, offsLow=-1, offsHigh=-1 + 2 = 1. In this case the group times of filter delays are balanced by additional delays. The total delay time is equal to the lowpass filter group delay which has the value of two samples in the decomposition stage in the original signal time frame.

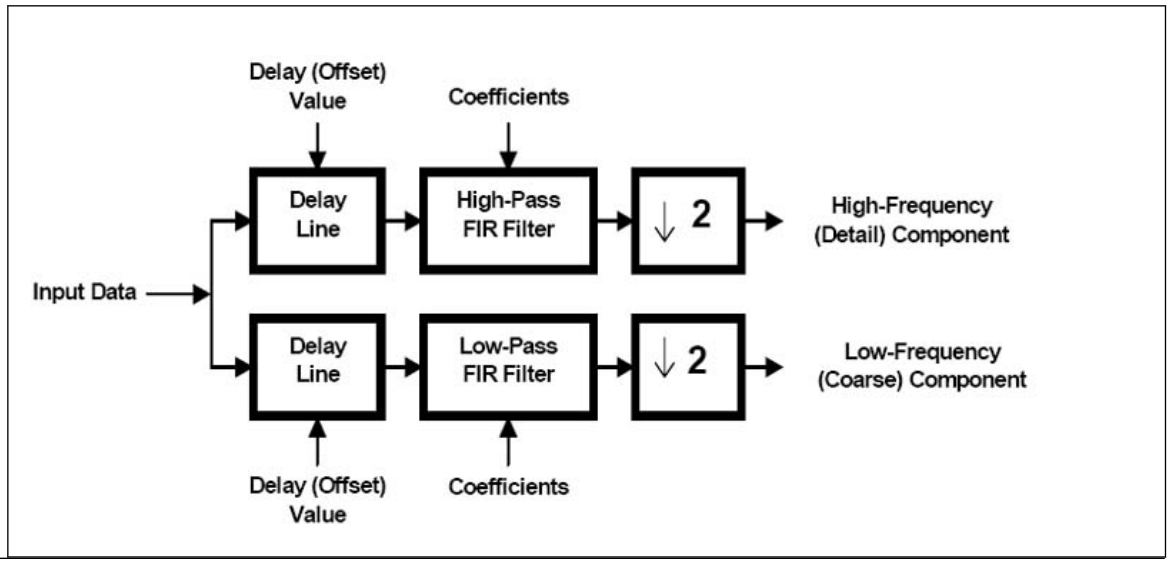
> **NOTE**
> Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, forward transform additional delays must be uniformly even for faultless signal reconstruction.

Multilevel decomposition algorithm. The implementation of multilevel decomposition algorithms may require synchronization of signal delays across components of different levels.

This is illustrated in the example of the three-level decomposition shown in Figure . 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.

One Level Forward Transform



Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error when the state identifier *`pState`* is incorrect.
`ippStsSizeErr`	Indicates an error when *`dstLen`* or *`srcLen`* is less than or equal to 0.

See Also

WTFwdGetSize, WTInvGetSize Compute the size of the wavelet transform state structures.
WTFwdInit, WTInvInit Initialize the wavelet transform state structures.
WTInv Computes the inverse wavelet transform.
Wavelet Transforms Example

WTFwdSetDlyLine, WTFwdGetDlyLine

Sets and gets the delay lines of the forward wavelet transform.

Syntax

```
IppStatus ippsWTFwdSetDlyLine_32f(IppsWTFwdState_32f* pState, const Ipp32f* pDlyLow,
const Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTFwdSetDlyLine_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_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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the state structure.
pDlyLow	Pointer to the vector which holds the delay lines for "low frequency" components.
pDlyHigh	Pointer to the vector which holds the delay lines for "high frequency" components.

Description

These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

`ippsWTFwdSetDlyLine`. This function sets the delay line values of the forward WT state.

`ippsWTFwdGetDlyLine`. This function gets the delay line values of the forward WT state.

Application Notes

These functions are designed to shape the signal prehistory, save and reconstruct delay lines. Delay lines are implemented separately for highpass and lowpass filters, which gives the option of getting independent signal prehistories for each filter.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the initial signal fed into the forward transform functions, i.e., delay line vectors must be made up of a succession of the signal prehistory counts in the same time frame as the initial signal.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the "low frequency" component filter:

dlyLowLen = lenLow + offsLow - 1,

where *lenLow* and *offsLow* are respectively the length and additional delay of the "low frequency" component filter.

The following expression defines the length of the delay line vector of the "high frequency" component filter:
dlyHighLen = lenHigh + offsHigh - 1,

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the "high frequency" component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are specified by the function ippsWTFwdInit.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDlyLow* or *pDlyHigh* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

WTInv

Computes the inverse wavelet transform.

Syntax

```
IppStatus ippsWTInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int srcLen,
Ipp32f* pDst, IppsWTInvState_32f* pState);
```

```
IppStatus ippsWTInv_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);
```

Include Files

ipps.h

Domain Dependencies

Headers: ippcore.h, ippvm.h

Libraries: ippcore.lib, ippvm.lib

Parameters

pSrcLow	Pointer to the vector which holds input coarse “low frequency” components.
pSrcHigh	Pointer to the vector which holds detail “high frequency” components.
srcLen	Number of elements in the vectors *pSrcHigh* and *pSrcLow*.
pDst	Pointer to the vector which holds the output reconstructed signal.
pState	Pointer to the state structure.

Description

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*.

Before using this function, you need to compute the size of the state structure and work buffer using the WTFwdGetSize_ WTInvGetSize function, and initialize the structure using WTFwdInit_ WTInvInit.

For an example on how to use this function, refer to Wavelet Transforms Example.

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. The Figure below shows an equivalent transform algorithm. Two interpolation filters are used for signal reconstruction; their coefficients are set at the initialization stage. The inverse transform implementation, similar to forward transform implementation, contains additional delay lines needed to synchronize the group time of filter delays and delays across different levels of data reconstruction.

Input and output data block lengths. These functions are designed to reconstruct the blocks of the even length signal. The signal component length must be the input data. The length of the output block of the reconstructed signal must be double the length of each of the components.

Filter group delay synchronization. In this example consider a biorthogonal set of spline filters of length 2 and 6:

```
static const float recLow[2] =
{
     1.00000000e+000f,
     1.00000000e+000f
};
static const float recHigh[6] =
{
    -1.25000000e-001f,
    -1.25000000e-001f,
     1.00000000e+000f,
    -1.00000000e+000f,
     1.25000000e-001f,
     1.25000000e-001f
};
```

This set of filters corresponds to the set of filters considered in a similar section of the description of the forward transform function `ippsWTFwd`.

Unlike the case described above, this time the high-pass 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 high-pass filter group delay, which at the decomposition stage is equal to two samples in the original signal time frame.

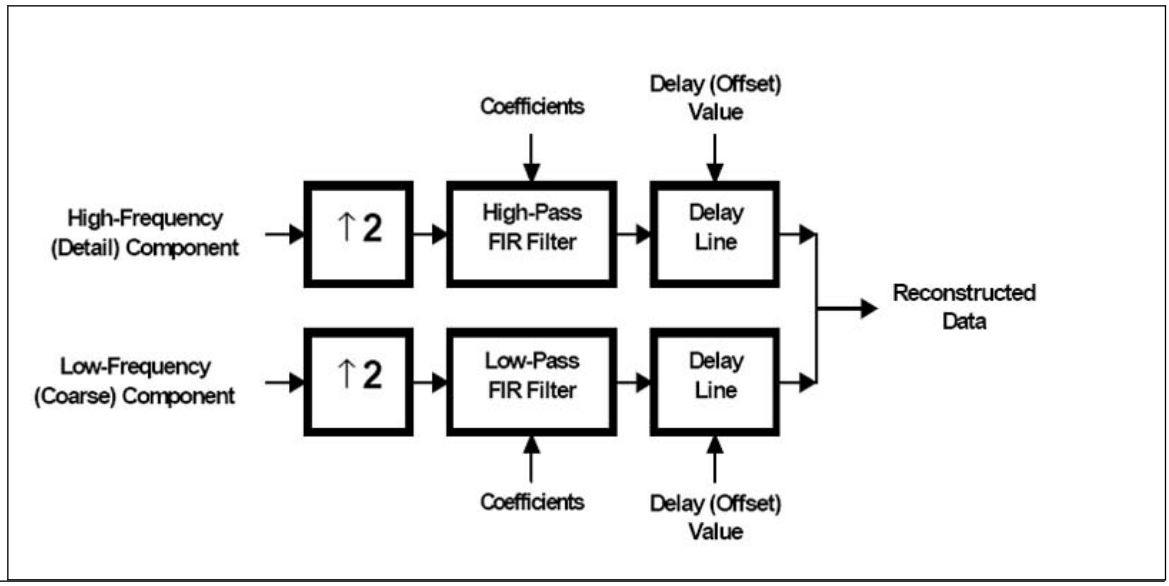
Total delay of one level of decomposition and reconstruction is equal to 4 samples, considering the decomposition stage delay.

> **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 . The scheme corresponds to the decomposition scheme described in the section of the description of the forward transform function `ippsWTFwd`. Therefore, for the inverse transform the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. The lowest level filter delays are set to *offsLow3* = 2, *offsHigh3* = 0. The total delay at this stage of reconstruction is equal to two samples. In order to ensure an equivalent delay of the "detail" part in the middle level, the delay must be increased. Respective values of additional delays for the second level are equal to *offsLow2* = 2, *offsHigh2* = *offsHigh3* + 2*2 = 4. A greater value of high frequency component delay needs to be selected for the last level of reconstruction, *offsLow1* = -1, *offsHigh1* = *offsHigh2* + 2*4 = 12.

The total delay for three levels of reconstruction is equal to 12 samples. The total delay of the three-level decomposition and reconstruction cycle is equal to 24 samples.

One Level Inverse Wavelet Transform



Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsStateMatchErr`	Indicates an error when the state identifier *`pState`* is incorrect.
`ippStsSizeErr`	Indicates an error when *`dstLen`* or *`srcLen`* is less than, or equal to 0.

See Also

WTFwdGetSize, WTInvGetSize Compute the size of the wavelet transform state structures.
WTFwdInit, WTInvInit Initialize the wavelet transform state structures.
WTFwd Computes the forward wavelet transform.
Wavelet Transforms Example

WTInvSetDlyLine, WTInvGetDlyLine

Sets and gets the delay lines of the inverse wavelet transform.

Syntax

```
IppStatus ippsWTInvSetDlyLine_32f(IppsWTInvState_32f* pState, const Ipp32f* pDlyLow,
const Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTInvSetDlyLine_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_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);
```

Include Files

`ipps.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`

Libraries: `ippcore.lib`, `ippvm.lib`

Parameters

pState	Pointer to the state structure.
pDlyLow	Pointer to the vector which holds delay lines for “low frequency” components.
pDlyHigh	Pointer to the vector which holds delay lines for “high frequency” components.

Description

These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and store them into the state structure *pState*.

ippsWTInvSetDlyLine. This function sets the delay line values of the inverse WT state.

ippsWTInvGetDlyLine. This function gets the delay line values of the inverse WT state.

Application Notes

These functions set and read delay lines of inverse multiscale transformation. The functions receive or return filter low and high frequency component delay line vectors. The functions may be used to shape previous history of each of the components. Installation functions and read functions together ensure that delay lines from each filter are saved and reconstructed.

Delay line data format. Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the low and high frequency components at the input of the inverse transform functions. Thus, delay line vectors must be made up of a succession of signal prehistory counts in the same time frame as the input components.

Delay line lengths. The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter in terms of the C language (integer division by two is used here for simplicity):

```
dlyLowLen = (lenLow + offsLow - 1) / 2,
```

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the "high frequency" component filter in terms of the C language:

dlyHighLen = (*lenHigh* + *offsHigh* - 1) / 2,

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the "high frequency" component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are specified by the function ippsWTInvInit.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pDlyLow* or *pDlyHigh* is NULL.
ippStsStateMatchErr	Indicates an error when the state identifier *pState* is incorrect.

Wavelet Transforms Example

The delay line paradigm is well-known interface solution for functions that require some pre-history in the streaming processing. In such application the use of the Intel IPP wavelet transform functions is similar to the use of the FIR, IIR, or multi-rate filters. (See also the discussion on the synchronization of low-pass and high-pass filter delays in this chapter.) But very often the wavelet transforms are used to process entire non-streaming data by extending with borders that are suitable for filter bank type that are used in transforms.

The following code example demonstrates how to implement this approach using the Intel IPP functions. It performs forward and inverse wavelet transforms of a short vector containing 12 elements. It uses Daubechies filter bank of the order 2 (that allows the perfect reconstruction) and periodical data extension by wrapping.

It is also may be useful as an illustration of how to fill delay line, if you need non-zero pre-history of signal in streaming applications.

Example

```
//   Filter bank for Daubechies, order 2
static const int fwdFltLenL = 4;
static const int fwdFltLenH = 4;
static const Ipp32f pFwdFltL[4] =
{    -1.294095225509215e-001f,     2.241438680418574e-001f,     8.365163037374690e-001f,
4.829629131446903e-001f  };
static const Ipp32f pFwdFltH[4] =
{    -4.829629131446903e-001f,     8.365163037374690e-001f,    -2.241438680418574e-001f,
-1.294095225509215e-001f  };
static const int invFltLenL = 4;
static const int invFltLenH = 4;
static const Ipp32f pInvFltL[4] =
{     4.829629131446903e-001f,     8.365163037374690e-001f,     2.241438680418574e-001f,
-1.294095225509215e-001f  };
static const Ipp32f pInvFltH[4] =
{    -1.294095225509215e-001f,    -2.241438680418574e-001f,     8.365163037374690e-001f,
-4.829629131446903e-001f  };
// minimal values
static const int fwdFltOffsL = -1;
static const int fwdFltOffsH = -1;
// minimal values, that corresponds to perfect reconstruction
static const int invFltOffsL = 0;
static const int invFltOffsH = 0;

void func_wavelet()
{
   IppStatus status=ippStsNoErr;
```

```
   Ipp32f pSrc[] = {1, -10, 324, 48, -483, 4, 7, -5532, 34, 8889, -57, 54};
   Ipp32f pDst[12];
   Ipp32f pLow[6];
   Ipp32f pHigh[6];
   IppsWTFwdState_32f* pFwdState;
   IppsWTInvState_32f* pInvState;
   int i, szState;

   printf("original:\n");
   for(i = 0; i < 12; i++)
      printf("%.0f; ", pSrc[i]);
   printf("\n");

   // Forward transform
   ippsWTFwdGetSize( ipp32f, fwdFltLenL, fwdFltOffsL, fwdFltLenH, fwdFltOffsH, &szState );
   pFwdState = (IppsWTFwdState_32f*)ippMalloc( szState );
   ippsWTFwdInit_32f( pFwdState, pFwdFltL, fwdFltLenL, fwdFltOffsL, pFwdFltH, fwdFltLenH,
fwdFltOffsH);
   // We substitute wrapping extension in "the beginning of stream"
   // Here should be the same pointers for this offsets,
   // but in the general case it may be different
   ippsWTFwdSetDlyLine_32f( pFwdState, &pSrc[10], &pSrc[10] );
   ippsWTFwd_32f( pSrc, pLow, pHigh, 6, pFwdState );

   printf("approx:\n");
   for(i = 0; i < 6; i++)
      printf("%.4f; ", pLow[i]);
   printf("\n");
   printf("details:\n");
   for(i = 0; i < 6; i++)
      printf("%.4f; ", pHigh[i]);
   printf("\n");

   // Inverse transform
   ippsWTInvGetSize( ipp32f, invFltLenL, invFltOffsL, invFltLenH, invFltOffsH, &szState );
   pInvState = (IppsWTInvState_32f*)ippMalloc( szState );
   ippsWTInvInit_32f( pInvState, pInvFltL, invFltLenL, invFltOffsL, pInvFltH, invFltLenH,
invFltOffsH );
   // For this particular case (non-shifted reconstruction)
   // here is first data itself,
   // that we need to place to delay line
   // [(invFltLenL + invFltOffsL - 1) / 2] elements for l. filtering
   // [(invFltLenH + invFltOffsH - 1) / 2] elements for h. filtering
   ippsWTInvSetDlyLine_32f( pInvState, pLow, pHigh);
   ippsWTInv_32f( &pLow[1], &pHigh[1], 5, pDst, pInvState );
   // Here are the substitution of the wrapping extension
   // at the "end of stream" and calculation of last samples of reconstruction
   // We do not use additional buffer and do not copy any data externally,
   // just substitute beginning of input data itself to simulate wrapping
   ippsWTInv_32f( pLow, pHigh, 1, &pDst[10], pInvState );

   printf("reconstruction:\n");
   for(i = 0; i < 12; i++)
      printf("%.0f; ", pDst[i]);
   printf("\n");

   ippFree(pFwdState);
   ippFree(pInvState);
}
```

After compiling and running it gives the following console output:

```
original:
1; -10; 324; 48; -483; 4; 7; -5532; 34; 8889; -57; 54;
approx:
19.1612; 58.5288; 87.8536; 487.5375; -5766.9277; 7432.4497;
details:
0.9387; 249.9611; -458.6568; 2739.2146; -3025.5576; -2070.5762;
reconstruction:
1; -10; 324; 48; -483; 4; 7; -5532; 34; 8889; -57; 54;
```

The program prints on console the original data, approximation, and details components after forward transform and perfect reconstruction of original data after inverse transform.

String Functions

8

This chapter describes the Intel® IPP functions that perform operations with a text. First part describes the functions for simple string manupulation. Second part contains functions that perform more sophisticated matching operation using patterns of the regular expressions.

String Manipulation

This section describes the Intel IPP functions that perform operations with strings. Intel IPP string functions for do not consider zero as the end of the string, but require that the length of the string (number of elements) be specified explicitly. Overlapping of the strings is not supported (for not in-place operations). Intel IPP string functions operate with two data types, `Ipp8u` and `Ipp16u`.

Find, FindRev

Looks for the first occurrence of the substring matching the specified string.

Syntax

```
IppStatus ippsFind_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind, int lenFind, int* pIndex);
```

```
IppStatus ippsFind_Z_8u(const Ipp8u* pSrcZ, const Ipp8u* pFindZ, int* pIndex);
```

```
IppStatus ippsFindRev_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind, int lenFind, int* pIndex);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source string.
pSrcZ	Pointer to the zero-ended source string.
len	Number of elements in the source string.
pFind	Pointer to the reference string.
pFindZ	Pointer to the zero-ended reference string.
lenFind	Number of elements in the reference string.
pIndex	Pointer to the result index.

Description

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 flavor ippsFind_Z operates with the zero-ended source and reference strings. The function ippsFindRev searches the source string in the reverse direction. The search is case-sensitive.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* or *lenFind* is negative.

Example

The code example below shows how to use the function ippsFind_8u.

```
 Ipp8u string[] = "abracadabra";
*/                 -----------
                   0123456789a
 */
 Ipp8u substring[] = "abra";
 Ipp8u any_of [] = "ftr";
 int index;
 ippsFind_8u( string, sizeof (string) - 1, substring, sizeof (substring) - 1, &index );
 printf ( "ippsFind_8u returned index = %d.\n", index );
 ippsFindC_Z_8u( string, " c ", &index );
 printf ( "ippsFind_Z_8u returned index = %d.\n", index );
 ippsFindRevCAny_8u( string, sizeof (string) - 1, any_of , sizeof ( any_of ) - 1, &index );
 printf ( "ippsFindRevCAny_8u returned index = %d.\n", index );

 Output:
 ippsFind_8u returned index = 0.
 ippsFind_Z_8u returned index = 4.
 ippsFindRevCAny_8u returned index = 9.
```

FindC, FindRevC

Looks for the first occurrence of the specified element within the source string.

Syntax

```
IppStatus ippsFindC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind, int* pIndex);
IppStatus ippsFindC_Z_8u(const Ipp8u* pSrcZ, Ipp8u valFind, int* pIndex);
IppStatus ippsFindRevC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind, int* pIndex);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source string.
pSrcZ	Pointer to the source zero-ended string.
len	Number of elements in the source string.
valFind	Value of the specified element.
pIndex	Pointer to the result index.

Description

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 flavor `ippsFindC_Z` operates with the zero-ended source string. The function `ippsFindRevC` searches the source string in the reverse direction. The search is case-sensitive.

Code example shows how to use the function `ippsFindC_Z_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len* is negative.

FindCAny, FindRevCAny

Looks for the first occurrence of any element of the specified array within the source string.

Syntax

```
IppStatus ippsFindCAny_8u(const Ipp8u* pSrc, int len, const Ipp8u* pAnyOf, int
lenAnyOf, int* pIndex);
```

```
IppStatus ippsFindRevCAny_8u(const Ipp8u* pSrc, int len, const Ipp8u* pAnyOf, int
lenAnyOf, int* pIndex);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source string.
len	Number of elements in the source string.
pAnyOf	Pointer to the array containing reference elements.
lenAnyOf	Number of elements in the array.

pIndex	Pointer to the result index.

Description

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.

Code example shows how to use the function ippsFindCAny_8u.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* or *lenAnyOf* is negative.

Insert

Inserts a string into another string.

Syntax

```
IppStatus ippsInsert_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pInsert, int
insertLen, Ipp8u* pDst, int startIndex);
```

```
IppStatus ippsInsert_8u_I(const Ipp8u* pInsert, int insertLen, Ipp8u* pSrcDst, int*
pSrcDstLen, int startIndex);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pInsert	Pointer to the string to be inserted.
insertLen	Number of elements in the string to be inserted.
pDst	Pointer to the destination string.
pSrcDst	Pointer to the source and destination string for in-place operation.
pSrcDstLen	Pointer to the number of elements in the source and destination string for in-place operation.
startIndex	Index of the insertion point.

Description

This function inserts the string *pInsert* containing *insertLen* elements into a source string *pSrc* of length *srcLen*. Insertion position is specified by *startIndex*. The result is stored in the *pDst*.

The in-place flavors of `ippsInsert` insert the string *pInsert* in the source string *pSrcDst* and store the result in the destination string *pSrcDst*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if one of the *srcLen*, *insertLen*, *pSrcSdtLen*, *startIndex* is negative, or *startIndex* is greater than *srcLen* or *pSrcDstLen*.

Example

The code example below shows how to use the function `ippsInsert_8u`.

```
Ipp8u string[] = " 1st string part 2nd string part ";
Ipp8u substring[] = " substring ";
Ipp8u dst_string [ sizeof (string) + sizeof (substring) - 1];
int dst_string_len ;
ippsInsert_8u( string, sizeof (string) - 1, substring, sizeof (substring) - 1, dst_string , 16 );
dst_string [ sizeof ( dst_string ) - 1] = 0;
printf ( "ippsInsert_8u returned: %s.\n", (char*) dst_string );
dst_string_len = sizeof ( dst_string ) - 1;
ippsRemove_8u_I( dst_string , & dst_string_len , 16, sizeof (substring) - 1 );
dst_string [ dst_string_len ] = 0;
printf ( "ippsRemove_8u_I returned: %s.\n", (char*) dst_string );
```

Result:

```
ippsInsert_8u returned: 1st string part substring 2nd string part .
ippsRemove_8u_I returned: 1st string part 2nd string part .
```

Remove

Removes a specified number of elements from the string.

Syntax

`IppStatus ippsRemove_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int startIndex, int len);`

`IppStatus ippsRemove_8u_I(Ipp8u* pSrcDst, int* pSrcDstLen, int startIndex, int len);`

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.

pDst	Pointer to the destination string.
pSrcDst	Pointer to the source and destination string for in-place operation.
pSrcDstLen	Pointer to the number of elements in the source and destination string for in-place operation.
startIndex	Index of the starting point.
len	Number of elements to be removed.

Description

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*.

Code example shows how to use the function ippsRemove_8u_I.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if one of the *srcLen*, *len*, *pSrcSdtLen*, *startIndex* is negative, or (*startIndex*+*len*) is greater than *srcLen* or *pSrcDstLen*.

Compare

Compares two strings of the fixed length.

Syntax

```
IppStatus ippsCompare_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len, int*
pResult);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Description

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 Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Example

The code example below shows how to use the function ippsCompare_8u.

```
Ipp8u string0[] = "These functions compare two strings";
Ipp8u string1[] = "These FUNCTIONS compare two strings";
int result;
ippsCompare_8u( string0, string1, sizeof (string0) - 1, &result );
printf ( "ippsCompare_8u said: " );
printf ( "string0 is %s string1.\n", (result < 0) ? "less than" :
((result > 0) ? "greater than" : "equal to") );
ippsCompareIgnoreCaseLatin_8u( string0, string1, sizeof (string0) - 1,
&result );
printf ( "ippsCompareIgnoreCaseLatin_8u said: " );
printf ( "string0 is %s string1.\n", (result < 0) ? "less than" :
((result > 0) ? "greater than" : "equal to") );

Output:
ippsCompare_8u said: string0 is greater than string1.
ippsCompareIgnoreCaseLatin_8u said: string0 is equal to string1.
```

CompareIgnoreCase, CompareIgnoreCaseLatin

Compares two strings of the fixed length ignoring case.

Syntax

IppStatus ippsCompareIgnoreCaseLatin_8u(const Ipp8u* *pSrc1*, const Ipp8u* *pSrc2*, int *len*, int* *pResult*);

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Description

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.

Code example shows how to use the function ippsCompareIgnoreCaseLatin_8u.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Equal

Compares two string of the fixed length for equality.

Syntax

```
IppStatus ippsEqual_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len, int* pResult);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc1	Pointer to the first source string.
pSrc2	Pointer to the second source string.
len	Maximum number of elements to be compared.
pResult	Pointer to the result.

Description

This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. Each element of the first string is compared with the corresponding element of the second string. When the first pair of non-matching elements is found, the function stops operation and stores 0 in *pResult*. If the strings are equal, the function stores 1 in *pResult*. The comparison is case-sensitive.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

TrimC

Deletes all occurrences of a specified symbol in the beginning and in the end of the string.

Syntax

```
IppStatus ippsTrimC_8u(const Ipp8u* pSrc, int srcLen, Ipp8u odd, Ipp8u* pDst, int*
pDstLen);
```

```
IppStatus ippsTrimC_8u_I(Ipp8u* pSrcDst, int* pLen, Ipp8u odd);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pSrcDst	Pointer to the source and destination string for the in-place operation.
pDst	Pointer to the destination string.
pDstLen	Pointer to the computed number of elements in the destination string.
pLen	Pointer to the number of elements in the source and destination string for the in-place operation.
odd	Symbol to be deleted.

Description

This function deletes all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrc* containing *srcLen* elements. The function stores the result string containing *pDstLen* elements in *pDst*.

The in-place flavors of `ippsTrimC` delete all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrcDst* containing *pLen* elements. These functions store the result string containing *pLen* elements in *pSrcDst*.

The operation is case-sensitive.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *srcLen* or *pLen* is negative.

Example

The code example below shows how to use the function `ippsTrimC_8u_I`.

```
 Ipp8u string[] = " ### abracadabra $$$ ";
 Ipp8u trim[] = " $#* ";
 Ipp8u dst_string [ sizeof (string)];
 int string_len , dst_string_len ;
 ippsTrimCAny_8u( string, sizeof (string) - 1, trim, sizeof (string) - 1, dst_string,&
dst_string_len );
 dst_string [ dst_string_len ] = 0;
 printf ( "ippsTrimCAny_8u returned: %s.\n", (char*) dst_string );
 string_len = sizeof (string) - 1;
 ippsTrimC_8u_I( string, & string_len , ' # ' );
 string[ string_len ] = 0;
 printf ( "ippsTrimC_8u_I returned: %s.\n", (char*)string );
```

Result:

```
ippsTrimCAny_8u returned: abracadabra .
 ippsTrimC_8u_I returned: abracadabra$$$ .
```

TrimCAny, TrimStartCAny, TrimEndCAny

Deletes all occurrences of any of the specified symbols in the beginning and in the end of the source string.

Syntax

`IppStatus ippsTrimCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);`

`IppStatus ippsTrimStartCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);`

`IppStatus ippsTrimEndCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);`

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source string.
srcLen	Number of elements in the source string.
pTrim	Pointer to the array containing the specified elements.
trimLen	Number of elements in the array.
pDst	Pointer to the destination string.
pDstLen	Pointer to the computed number of elements in the destination string.

Description

The 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.

Code example shows how to use the function `ippsTrimCAny_8u`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *srcLen* or *trimLen* is negative.

ReplaceC

Replaces all occurrences of a specified element in the source string with another element.

Syntax

```
IppStatus ippsReplaceC_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, Ipp8u oldVal, Ipp8u newVal);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source string.
len	Number of elements in the source string.
pDst	Pointer to the destination string.
oldVal	Element to be replaced.

newVal	Element that replaces *oldVal*.

Description

This function replaces all occurrences of a specified element *oldVal* in the source string *pSrc* with another specified element *newVal*, and stores the new string in the *pDst*. The operation is case-sensitive.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Example

The code example below shows how to use the function ippsReplaceC_8u.

```
Ipp8u string[] = "abracadabra";
Ipp8u dst_string [ sizeof (string)];
ippsReplaceC_8u( string, dst_string , sizeof (string), 'a', 'o' );
printf ( "ippsReplaceC_8u returned: %s.\n", (char*) dst_string );

Output:
ippsReplaceC_8u returned: obrocodobro.
```

Uppercase, UppercaseLatin

Converts alphabetic characters of a string to all uppercase symbols.

Syntax

```
IppStatus ippsUppercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsUppercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

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

Description

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.

Code example shows how to use the function ippsUppercaseLatin_8u.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Lowercase, LowercaseLatin

Converts alphabetic characters of a string to all lowercase symbols.

Syntax

```
IppStatus ippsLowercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLowercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

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

Description

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 Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error condition if at least one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Example

The code example below shows how to use the function ippsLowercaseLatin_8u_I.

```
Ipp8u string[] = "These Functions Vary the Case!";
ippsLowercaseLatin_8u_I( string, sizeof (string) - 1 );
printf ( "Lower: %s\n", (char*)string );
ippsUppercaseLatin_8u_I( string, sizeof (string) - 1 );
printf ( "Upper: %s\n", (char*)string );
```

Result:

```
Lower: these functions vary the case!
Upper: THESE FUNCTIONS VARY THE CASE!
```

Hash

Calculates the hash value for the string.

Syntax

```
IppStatus ippsHash_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHash_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHashSJ2_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHashSJ2_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHashMSCS_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHashMSCS_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source string.
len	Number of elements in the string.
pHashVal	Pointer to the result value.

Description

This function produces the hash value *pHasVal* for the specified string *pSrc*. The hash value is fairly unique to the given string. If hash values of two strings are different, the strings are different as well. If the hush values are the same, the strings are probably identical. The hash value is calculated in the following manner: initial value is set to 0, then the hash value is calculated successively for each i-th string element as *pHashVal*[i] = 2* *pHashVal*[i-1]^ *pSrc*[i], where ^ denotes a bitwise exclusive OR (XOR) operator. The hash value for the last element is the hash value for the whole string.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *len* is negative.

Example

The code example below shows how to use the functions ippsHash_8u32u, ippsHashSJ2_8u32u, and ippsHashMSCS_8u32u.

```
Ipp8u string[] = "monkey";
Ipp32u hash;

hash = 0;
ippsHash_8u32u( string, sizeof (string) - 1, &hash );
printf ( "ippsHash_8u32u hash value: %u\n", hash );
hash = 0;
ippsHashSJ2_8u32u( string, sizeof (string) - 1, &hash );
printf ( "ippsHashSJ2_8u32u hash value: %u\n", hash );
hash = 0;
ippsHashMSCS_8u32u( string, sizeof (string) - 1, &hash );
printf ( "ippsHashMSCS_8u32u hash value: %u\n", hash );

Output:
ippsHash_8u32u hash value: 2367
ippsHashSJ2_8u32u hash value: 3226471379
ippsHashMSCS_8u32u hash value: 1466279646
```

Concat

Concatenates several strings together.

Syntax

IppStatus ippsConcat_8u_D2L(const Ipp8u* const *pSrc*[], const int *srcLen*[], int *numSrc*, Ipp8u* *pDst*);

IppStatus ippsConcat_8u(const Ipp8u* *pSrc1*, int *len1*, const Ipp8u* *pSrc2*, int *len2*, Ipp8u* *pDst*);

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc1	Pointer to the first source string.
len1	Number of elements in the first string.
pSrc2	Pointer to the second source string.
len2	Number of elements in the second string.
pSrc	Pointer to the array of source strings.
srcLen	Pointer to the array of lengths of the source strings.
numSrc	Number of source strings.
pDst	Pointer to the destination string.

Description

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 must 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 Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *len1* or *len2* is negative, or `srcLen[i]` is negative for `i <` *numSrc*.
`ippStsSizeErr`	Indicates an error condition if *numSrc* is equal to or less than 0.

Example

The code example below shows how to use the functions `ippsConcat_8u` and `ippsConcat_8u_D2L`.

```
 Ipp8u string0[] = "This is the initial string.";
Ipp8u string1[] = "Extra text added to the string...";
Ipp8u* string_ptr [2] = { string0, string1 };
int string_len_ptr [2] = { sizeof (string0) - 1, sizeof (string1)};
Ipp8u dst_string [ sizeof (string0) + sizeof (string1)];

ippsConcat_8u( string0, sizeof (string0) - 1, string1, sizeof (string1), dst_string );
printf ("ippsConcat_8u said: %s\n", (char*) dst_string );
ippsConcat_8u_D2L( string_ptr , string_len_ptr , 2, dst_string );
printf ("ippsConcat_8u_D2L said: %s\n", (char*) dst_string );
ippsConcatC_8u_D2L( string_ptr , string_len_ptr, 2, '#', dst_string );
printf ("ippsConcatC_8u_D2L said: %s\n", (char*) dst_string );

Output:
ippsConcat_8u said: This is the initial string. Extra text added to the string...
ippsConcat_8u_D2L said: This is the initial string. Extra text added to the string...
ippsConcatC_8u_D2L said: This is the initial string. # Extra text added to the string...
```

ConcatC

Concatenates several strings together and inserts symbol delimiters between them.

Syntax

```
IppStatus ippsConcatC_8u_D2L(const Ipp8u* const pSrc[], const int srcLen[], int numSrc,
Ipp8u delim, Ipp8u* pDst);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the array of source strings.
srcLen	Pointer to the array of lengths of the source strings.
numSrc	Number of source strings.
delim	Symbol delimiter.
pDst	Pointer to the destination string.

Description

This function concatenates *numSrc* strings *pSrc* together and inserts a specified delimiter symbol *delim* between them in the resulting string *pDst*.

Code example shows how to use the function `ippsConcatC_8u_D2L`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if at least one of the specified pointers is `NULL`.
`ippStsLengthErr`	Indicates an error condition if `srcLen[i]` is negative for `i <` *numSrc*.
`ippStsSizeErr`	Indicates an error condition if *numSrc* is equal to or less than 0.

SplitC

Splits source string into separate parts.

Syntax

```
IppStatus ippsSplitC_8u_D2L(const Ipp8u* pSrc, int srcLen, Ipp8u delim, Ipp8u* pDst[],
int dstLen[], int* pNumDst);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source strings.
srcLen	Number of elements in the source string.
delim	Symbol delimiter.
pDst	Pointer to the array of the destination strings.
dstLen	Pointer to array of the destination string lengths.
pNumDst	Number of destination strings.

Description

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 are inserted into the array of destination strings, where n is the number of delimiter occurrences in this position.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if one of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *srcLen* is negative, or *dstLen* is negative for i < *pNumDst*.
ippStsSizeErr	Indicates an error condition if *pNumDst* is equal to or less than 0.
ippStsOvermatchStrings	Indicates a warning if number of output strings exceeds the initially specified number *pNumDst*; in this case odd strings are discarded.
ippStsOverlongString	Indicates a warning if in some output strings the number of elements exceeds the initially specified value *dstLen*; in this case corresponding strings are truncated to initial lengths.

Example

The code example below shows how to use the function ippsSplitC_8u_2DL.

```
 Ipp8u string[] = "1st string # 2nd string";
 Ipp8u dst_string0[ sizeof (string)];
 Ipp8u dst_string1[ sizeof (string)];
 Ipp8u* dst_string_ptr [] = { dst_string0, dst_string1 };
 int dst_string_len_ptr [] = { sizeof (dst_string0), sizeof (dst_string1) };
 int dst_string_num = 2;
 int i ;
 ippsSplitC_8u_D2L( string, sizeof (string) - 1, '#', dst_string_ptr, dst_string_len_ptr, &
dst_string_num );
 printf ( "Destination strings number: %d\n", dst_string_num );
 for( i = 0; i < dst_string_num ; i ++ ) {
     dst_string_ptr [ i ][ dst_string_len_ptr [ i ]] = 0;
     printf ( "%d: %s.\n", i, (char*) dst_string_ptr [ i ] );
 }

 Output:
```

```
Destination strings number: 2
0: 1st string.
1: 2nd string.
```

Regular Expressions

This section describes the Intel IPP functions that perform matching operations with the Perl-compatible regular expression patterns. See http://search.cpan.org/dist/perl/pod/perlre.pod for more details about Perl-compatible regular expressions.

The current version of the Intel IPP functions for regular expressions have some limitations, specifically they do not support literal (metacharacters `\l`, `\L`, `\u`, `\U`, `\N{name}`), embedded Perl code (`?{code}`), extended regular expression (`??{code}`).

RegExpInit

Initializes the structure for processing matching operation with regular expressions.

Syntax

`IppStatus ippsRegExpInit(const char* pPattern, const char* pOptions, IppRegExpState* pRegExpState, int* pErrOffset);`

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pPattern	Pointer to the pattern of regular expression.
pOptions	Pointer to options for compiling and executing regular expressions (possible values `i`, `s`, `m`, `x`, `g`). It must be `NULL` if no options are required.
pRegExpState	Pointer to the structure containing internal form of a regular expression.
pErrOffset	Pointer to the offset in the pattern if compiling is break.

Description

This function initializes a regular expression state structure *pRegExpState* in the external buffer. The size of this buffer must be computed previously by calling the function `ippsRegExpGetSize`. The function compiles the initial pattern of regular expressions *pPattern* in accordance with the compiling options specified by *pOptions*, converts it to the specific internal form, and stores it in the initialized structure. This structure is used by the function `ippsRegExpFind` to perform matching operation.

If the compiling is not completed, the function returns the pointer *pErrOffset* pointed to the position in the pattern where the compiling is interrupted.

NOTE
The parameter *pPattern* must be the same for both functions ippsRegExpInit and ippsRegExpGetSize.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if one of the specified pointers is NULL.
ippStsRegExpOptionsErr	Indicates an error if specified options are incorrect
ippStsRegExpQuantifierErr	Indicates an error if the quantifier is incorrect
ippStsRegExpGroupingErr	Indicates an error if the grouping is incorrect
ippStsRegExpBackRefErr	Indicates an error if the back reference is incorrect
ippStsRegExpChClassErr	Indicates an error if the character class is incorrect
ippStsRegExpMetaChErr	Indicates an error if the metacharacter is incorrect

RegExpGetSize

Computes the size of the regular expression state structure.

Syntax

```
IppStatus ippsRegExpGetSize(const char* pPattern, int* pRegExpStateSize);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pPattern	Pointer to the pattern of regular expression.
pRegExpStateSize	Pointer to the computed size of the regular expression structure.

Description

This function computes the size of the memory that is necessary to initialize by the function ippsRegExpInit the regular expression state structure containing the pattern *pPattern* in the internal form. The value of the computed size is stored in the *pRegExpStateSize*.

NOTE
The parameter *pPattern* must be the same for both functions ippsRegExpInit and ippsRegExpGetSize.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if one of the specified pointers is NULL.

RegExpSetMatchLimit

Sets the value of the matchLimit parameter.

Syntax

```
IppStatus ippsRegExpSetMatchLimit(int matchLimit, IppRegExpState* pRegExpState);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

matchLimit	Value of the of the matches kept in stack.
pRegExpState	Pointer to the structure containing internal form of a regular expression.

Description

This function sets the value of the parameter *matchLimit* that specifies how many times the function ippsRegExpFind can be called through the single execution avoiding the possible stack overflow. The default value is set very large, so you should set this parameter to the reasonable value in accordance with your needs.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if the *pRegExpState* pointer is NULL.

RegExpFind

Looks for the occurrences of the substrings matching the specified regular expression.

Syntax

```
IppStatus ippsRegExpFind_8u(const Ipp8u* pSrc, int srcLen, IppRegExpState*
pRegExpState, IppRegExpFind* pFind, int* pNumFind);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source strings.
srcLen	Number of elements in the source string.
pRegExpState	Pointer to the structure containing internal form of regular expression.
pFind	Array of pointers to the matching substrings.
pNumFind	Size of the array *pFind* on input, number of matching substrings on output.

Description

This function search through the *srcLen* elements of the source string *pSrc* for substrings that match the specified regular expression in accordance with the regular expression pattern that is stored in the structure *pRegExpState*. This structure must be initialized by the ippsRegExpInit function beforehand. Initially the parameter *pNumFind* specifies the size of array *pFind*, the output parameter *pNumFind* returns the number of the matching substrings.

pFind->pFind specifies the offset of the pointer to the matching substring, and *pFind->lenFind* - number of elements in the matching substring. *pFind* [0] points to the substring that matches the whole regular expression, *pFind* [1] points to the substring that matches the first grouping, *pFind*[2] points to the substring that matches the second grouping, and so on.

If number of matches exceeds the size of the *pFind* array, the function returns **ippStsOverflow** status. In this case you should increase *pNumFind* value and repeat the search.

> **NOTE**
> It is recommended to set the default value of the parameter *matchLimit* in accordance with real necessity by calling the function ippsRegExpSetMatchLimit beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error condition if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error condition if *srcLen* is negative, or *pNumFind* is less than or equal to 0.
ippStsRegExpErr	The state structure *pRegExpState* contains wrong data.
ippStsRegExpMatchLimitErr	The match limit has been exhausted.
ippStsOverflow	The size of *pFind* array is less than the number of matching substrings.

Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

RegExpFind.c

RegExpSetFormat

Sets source encoding format for given compiled pattern.

Syntax

```
IppStatus ippsRegExpSetFormat(IppRegExpFormat fmt, IppRegExpState* pRegExpState);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

fmt	New source encoding mode.
pRegExpState	Pointer to the structure containing internal form of a regular expression.

Description

The function sets the new source encoding format for given compiled pattern. Default source encoding format after `ippsRegExpInit` is UTF-8 with ASCII auto detection.

The enumeration `IppRegExpFormat` for representing a source encoding mode is defined as

```
typedef enum {
ippFmtASCII=0,
ippFmtUTF8,
}IppRegExpFormat;
```

CAUTION
The function `ippsRegExpFind` returns `ippStsRegExpErr` when pattern and source string are coded with different encoding, or pattern contains unsupported features by chosen encoding format.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when the *pRegExpState* pointer is `NULL`.
`ippStsRangeErr`	Indicates an error when mode is not a valid element of the enumerated type `IppRegExpFormat`.

ConvertUTF

Converts the UTF16BE or UTF16LE format to UTF8 and vice versa.

Syntax

```
IppStatus ippsConvertUTF_8u16u(const Ipp8u* pSrc, Ipp32u* pSrcLen, Ipp16u* pDst,
Ipp32u* pDstLen, int BEFlag);
```

```
IppStatus ippsConvertUTF_16u8u(const Ipp16u* pSrc, Ipp32u* pSrcLen, Ipp8u* pDst,
Ipp32u* pDstLen, int BEFlag);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source vector.
pSrcLen	Length of the *pSrc* vector on input; its used length on output.
pDst	Pointer to the destination vector.
pDstLen	Length of the *pDst* vector on input; its used length on output.
BEFlag	Flag to indicate the UTF16BE format. 0 means the UTF16LE format.

Description

The function flavor `ippsConvertUTF_8u16u` converts the UTF8 format to the UTF16LE or UTF16BE format. The function flavor `ippsConvertUTF_16u8u` converts UTF16LE or UTF16BE format to the UTF8 format.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition if one of the specified pointers is `NULL`.

RegExpReplaceGetSize

Calculates the size of the state structure for the find-replace operation.

Syntax

```
IppStatus ippsRegExpReplaceGetSize(const Ipp8u* pSrcReplacement, Ipp32u* pSize);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrcReplacement	Pointer to the source null-terminated replace pattern.

pSize	Pointer to the size of the state structure for the find and replace operation.

Description

This function calculates the size of the memory that is necessary to initialize the state structure required by the function `ippsRegExpReplaceInit`. The value of the calculated size is stored in the *pSize*.

> **NOTE**
> Value of the parameter *pSrcReplacement* must be the same as used for the `ippsRegExpReplaceInit` function call.

Return Values

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

RegExpReplaceInit

Initialize the state structure for the find-replace operation.

Syntax

```
IppStatus ippsRegExpReplaceInit(const Ipp8u* pSrcReplacement, IppRegExpReplaceState*
pReplaceState);
```

Include Files

`ippch.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrcReplacement	Pointer to the source null-terminated replace pattern.
pReplaceState	Pointer to the state structure for the find and replace operation.

Description

This function initializes a state structure *pState* for the find and replace operation in the external buffer. The size of this buffer must be computed previously by calling the function `ippsRegExpReplaceGetSize`.

> **NOTE**
> Value of the parameter *pSrcReplacement* must be the same as used for the `ippsRegExpReplaceGetSize` function call.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when the *pReplaceState* or *pSrcReplacement* pointer is NULL.

RegExpReplace

Performs find and replace operation.

Syntax

```
IppStatus ippsRegExpReplace_8u(const Ipp8u* pSrc, int* pSrcLenOffset, Ipp8u* pDst, int*
pDstLen, IppRegExpFind* pFind, int* pNumFind, IppRegExpState* pRegExpState,
IppRegExpReplaceState* pReplaceState);
```

Include Files

ippch.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source string.
pSrcLenOffset	Pointer to length of the *pSrc* vector on input; its used length on output.
pDst	Pointer to the destination string.
pDstLen	Pointer to length of the *pDst* vector on input; its used length on output.
pFind	Array of pointers to the matching substrings.
pNumFind	Pointer to size of the array *pFind* on input, to number of matching substrings on output.
pRegExpState	Pointer to the compiled pattern structure.
pReplaceState	Pointer to the state structure for the find and replace operation.

Description

This function search through the *pSrcLen* elements of the source string *pSrc* for substrings that match the specified regular expression in accordance with the regular expression pattern that is stored in the structure *pRegExpState*. This state structure must be initialized by the ippsRegExpInit function beforehand. All found matches are replaced according to the replacement pattern that is stored in the structure *pReplaceState*. This structure must be initialized beforehand by the function ippsRegExpReplaceInit.

Initially the parameter *pNumFind* specifies the size of array *pFind*, the output parameter *pNumFind* returns the number of the matching substrings. *pFind->pFind* specifies the offset of the pointer to the matching substring, and *pFind->lenFind* - number of elements in the matching substring. *pFind* [0] points to the substring that matches the whole regular expression, *pFind*[1] points to the substring that matches the first grouping, *pFind* [2] points to the substring that matches the second grouping, and so on.

NOTE
The compiled regular expression pattern and/or replacement pattern can be used for different input strings in different combinations.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *`pSrcLen`* or *`pDstLen`* is less than or equal to zero.

Fixed-Accuracy Arithmetic Functions

9

This chapter describes Intel® IPP fixed-accuracy transcendental mathematical real and complex functions of vector arguments. These functions take an input vector as argument, compute values of the respective elementary function element-wise, and return the results in an output vector.

Function specifications comply with the common API agreement of Intel IPP, but include some new features essential to scientific arithmetic functions. The main feature is a more elaborate specification of accuracy that differs from the common definition in adding several new levels of accuracy, besides original levels introduced by single precision and double precision data formats.

Fixed-accuracy vector functions implementation supports the 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 the 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 `A26`, `A50`, or `A53` for the double precision data format. Flavors `A11`, `A21`, `A26`, and `A50` provide approximately 3, 6, 8, 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 the 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.

NOTE
Do not confuse fixed-accuracy arithmetic functions described here with common arithmetic functions that have similar functionality but follow different accuracy specifications.

Intel IPP fixed-accuracy arithmetic functions may return status codes of the specific warnings listed in the table below. In this case, the value returned is positive and the computation is continued.

Warning Status Codes for Fixed-Accuracy Arithmetic Functions

	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.

NOTE
All functions described in this chapter support in-place operation.

Arithmetic Functions

Add

Performs element by element addition of two vectors.

Syntax

```
IppStatus ippsAdd_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsAdd_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsAdd_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsAdd_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function performs element by element addition of the vectors *pSrc1* and *pSrc2*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavors `ippsAdd_32f_A24` and `ippsAdd_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAdd_64f_A53` and `ippsAdd_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

pDst[n] = (*pSrc1*[n]) + (*pSrc2*[n]), $0 \le n < len$.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc1*, *pSrc2* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function ippsAdd.

```
IppStatus ippsAdd_32f_A24_sample(void)
{
const Ipp32f x1[4] = {+4.885, -0.543, -3.809, -4.953};
const Ipp32f x2[4] = {-0.543, -3.809, -4.953, +4.885};
Ipp32f  y[4];

    IppStatus st = ippsAdd_32f_A24( x1, x2, y, 4 );

    printf(" ippsAdd_32f_A24:\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:

 ippsAdd_32f_A24:
 x1 = +4.885 -0.543 -3.809 -4.953
 x2 = -0.543 -3.809 -4.953 +4.885
  y = +4.342 -4.352 -8.762 -0.068
```

Sub

Performs element by element subtraction of one vector from another.

Syntax

IppStatus ippsSub_32f_A24 (const Ipp32f* *pSrc1*, const Ipp32f* *pSrc2*, Ipp32f* *pDst*, Ipp32s *len*);

IppStatus ippsSub_64f_A53 (const Ipp64f* *pSrc1*, const Ipp64f* *pSrc2*, Ipp64f* *pDst*, Ipp32s *len*);

IppStatus ippsSub_32fc_A24 (const Ipp32fc* *pSrc1*, const Ipp32fc* *pSrc2*, Ipp32fc* *pDst*, Ipp32s *len*);

IppStatus ippsSub_64fc_A53 (const Ipp64fc* *pSrc1*, const Ipp64fc* *pSrc2*, Ipp64fc* *pDst*, Ipp32s *len*);

Include Files

ippvm.h

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function performs element by element subtraction of the vector *pSrc2* from the vector *pSrc1*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavors `ippsSub_32f_A24` and `ippsSub_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSub_64f_A53` and `ippsSub_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

pDst[n] = (*pSrc1*[n]) - (*pSrc2*[n]), $0 \le n < len$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function `ippsSub`.

```
IppStatus ippsSub_32f_A24_sample(void)
{
const Ipp32f x1[4] = {+4.885, -0.543, -3.809, -4.953};
const Ipp32f x2[4] = {-0.543, -3.809, -4.953, +4.885};
Ipp32f  y[4];

    IppStatus st = ippsSub_32f_A24( x1, x2, y, 4 );

    printf(" ippsSub_32f_A24:\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:

 ippsSub_32f_A24:
 x1 = +4.885 -0.543 -3.809 -4.953
 x2 = -0.543 -3.809 -4.953 +4.885
  y = +5.428 +3.266 +1.144 -9.838
```

Sqr

Performs element by element squaring of the vector.

Syntax

```
IppStatus ippsSqr_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSqr_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

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

len	Number of elements in the vectors.

Description

This function performs element by element squaring of the vector *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsSqr_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 `ippsSqr_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:

$pDst[n] = (pSrc[n])^2, 0 \le n < len.$

Return Values

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

Example

The example below shows how to use the function `ippsSqr`.

```
IppStatus ippsSqr_32f_A24_sample(void)
{
const Ipp32f x[4] = {+4.885, -0.543, -3.809, -4.953};
Ipp32f  y[4];

    IppStatus st = ippsSqr_32f_A24( x, y, 4 );

    printf(" ippsSqr_32f_A24:\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:

 ippsSqr_32f_A24:
 x = +4.885 -0.543 -3.809 -4.953
 y = +23.863 +0.295 +14.508 +24.532
```

Mul

Performs element by element multiplication of two vectors.

Syntax

```
IppStatus ippsMul_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsMul_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsMul_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsMul_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsMul_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsMul_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
IppStatus ippsMul_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
IppStatus ippsMul_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function performs element by element multiplication of the vectors *pSrc1* and *pSrc2*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsMul_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsMul_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsMul_32f_A24` and `ippsMul_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsMul_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsMul_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsMul_64f_A53` and `ippsMul_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

pDst[n] = (*pSrc1*[n]) x (*pSrc2*[n]), 0 ≤ n < *len*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function `ippsMul`.

```
IppStatus ippsMul_32f_A24_sample(void)
{
const Ipp32f x1[4] = {+4.885, -0.543, -3.809, -4.953};
const Ipp32f x2[4] = {-0.543, -3.809, -4.953, +4.885};
Ipp32f  y[4];

    IppStatus st = ippsMul_32f_A24( x1, x2, y, 4 );

    printf(" ippsMul_32f_A24:\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:

 ippsMul_32f_A24:
 x1 = +4.885 -0.543 -3.809 -4.953
 x2 = -0.543 -3.809 -4.953 +4.885
  y = -2.653 +2.068 +18.866 -24.195
```

MulByConj

Performs element by element multiplication of a vector `a` *element and a conjugated vector* `b` *element.*

Syntax

```
IppStatus ippsMulByConj_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
pDst, Ipp32s len);
IppStatus ippsMulByConj_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
pDst, Ipp32s len);
IppStatus ippsMulByConj_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
pDst, Ipp32s len);
IppStatus ippsMulByConj_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
pDst, Ipp32s len);
IppStatus ippsMulByConj_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
pDst, Ipp32s len);
IppStatus ippsMulByConj_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function performs element by element multiplication of the vector *pSrc1* and the conjugated vector *pSrc2*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsMulByConj_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsMulByConj_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsMulByConj_32fc_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 `ippsMulByConj_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsMulByConj_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsMulByConj_64fc_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] x CONJ(*pSrc2*[n]), 0 ≤ n < *len*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc1*, *pSrc2* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function `ippsMulByConj`.

```
IppStatus ippsMulByConj_32fc_A24_sample(void)
{
const Ipp32fc x1[4] = {{+2.885,-1.809}, {-0.543,-2.809}};
const Ipp32fc x2[4] = {{-0.543,-2.809}, {-1.809,-2.809}};
Ipp32fc  y[2];

    IppStatus st = ippsMulByConj_32fc_A24( x1, x2, y, 2 );

    printf(" ippsMulByConj_32fc_A24:\n");
    printf(" x1 = %+.3f%+.3f*i   %+.3f%+.3f*i \n", x1[0].re, x1[0].im, x1[1].re, x1[1].im);
    printf(" x2 = %+.3f%+.3f*i   %+.3f%+.3f*i \n", x2[0].re, x2[0].im, x2[1].re, x2[1].im);
    printf(" y  = %+.3f%+.3f*i   %+.3f%+.3f*i \n",  y[0].re,  y[0].im,  y[1].re,  y[1].im);
    return st;
}

Output results:

 ippsMulByConj_32fc_A24:
 x1 = +2.885-1.809*i   -0.543-2.809*i
 x2 = -0.543-2.809*i   -1.809-2.809*i
  y = +3.515+9.086*i   +8.873+3.556*i
```

Conj

Performs element by element conjugation of the vector.

Syntax

```
IppStatus ippsConj_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
```

```
IppStatus ippsConj_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function performs element by element conjugation of the vector *pSrc* and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsConj_32fc_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 `ippsConj_64fc_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] = CONJ(*pSrc*[n]), $0 \le n < len$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function `ippsConj`.

```
IppStatus ippsConj_32fc_A24_sample(void)
{
const Ipp32fc x[2] = {{+2.885,-1.809}, {-0.543,-2.809}};
Ipp32fc  y[2];

    IppStatus st = ippsConj_32fc_A24( x, y, 2 );

    printf(" ippsConj_32fc_A24:\n");
    printf(" x = %+.3f%+.3f*i   %+.3f%+.3f*i \n", x[0].re, x[0].im, x[1].re, x[1].im);
    printf(" y = %+.3f%+.3f*i   %+.3f%+.3f*i \n", y[0].re, y[0].im, y[1].re, y[1].im);
    return st;
}

Output results:

 ippsConj_32fc_A24:
  x = +2.885-1.809*i   -0.543-2.809*i
  y = +2.885+1.809*i   -0.543+2.809*i
```

Abs

Computes the absolute value of vector elements.

Syntax

```
IppStatus ippsAbs_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAbs_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAbs_32fc_A11 (const Ipp32fc* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAbs_32fc_A21 (const Ipp32fc* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAbs_32fc_A24 (const Ipp32fc* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAbs_64fc_A26 (const Ipp64fc* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAbs_64fc_A50 (const Ipp64fc* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAbs_64fc_A53 (const Ipp64fc* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the absolute value of the vector *pSrc* elements and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsAbs_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAbs_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAbs_32f_A24` and `ippsAbs_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAbs_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsAbs_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAbs_64f_A53` and `ippsAbs_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = |pSrc1[n]|, 0 \le n < len$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function `ippsAbs`.

```
IppStatus ippsAbs_32fc_A24_sample(void)
{
const Ipp32fc x[2] = {{+2.885,-1.809}, {-0.543,-2.809}};
Ipp32fc  y[2];

    IppStatus st = ippsAbs_32fc_A24( x, y, 2 );

    printf(" ippsAbs_32fc_A24:\n");
    printf(" x = %+.3f%+.3f*i   %+.3f%+.3f*i \n", x[0].re, x[0].im, x[1].re, x[1].im);
    printf(" y = %+.3f          %+.3f%       \n", y[0],             y[1]);
    return st;
}

Output results:

 ippsAbs_32fc_A24:
  x = +2.885-1.809*i   -0.543-2.809*i
  y = +3.405           +2.861
```

Arg

Computes the argument of vector elements.

Syntax

```
IppStatus ippsArg_32fc_A11(const Ipp32fc* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsArg_32fc_A21(const Ipp32fc* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsArg_32fc_A24(const Ipp32fc* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsArg_64fc_A26(const Ipp64fc* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsArg_64fc_A50(const Ipp64fc* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsArg_64fc_A53(const Ipp64fc* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

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

len	Number of elements in the vectors.

Description

This function computes the argument of the vector *pSrc* elements and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsArg_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsArg_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsArg_32fc_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 `ippsArg_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsArg_64fc_A50` guarantees 50 correctly rounded bits of significand,or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsArg_64fc_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:

$pDst[n] = \varphi(pSrc1[n]), 0 \le n < len.$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function `ippsArg`.

```
IppStatus ippsArg_32fc_A24_sample(void)
{
const Ipp32fc x[2] = {{+2.885,-1.809}, {-0.543,-2.809}};
Ipp32fc  y[2];

    IppStatus st = ippsArg_32fc_A24( x, y, 2 );

    printf(" ippsArg_32fc_A24:\n");
    printf(" x = %+.3f%+.3f*i   %+.3f%+.3f*i \n", x[0].re, x[0].im, x[1].re, x[1].im);
    printf(" y = %+.3f          %+.3f%       \n", y[0],             y[1]);
    return st;
}

Output results:

 ippsArg_32fc_A24:
  x = +2.885-1.809*i   -0.543-2.809*i
  y = -0.560           -1.762
```

Power and Root Functions

Inv

Computes inverse value of each vector element.

Syntax

```
IppStatus ippsInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInv_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

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_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

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:

$pDst[n] = 1/(pSrc[n]), 0 \le n < len$.

Return Values

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

Example

The example below shows how to use the function `ippsInv`.

```
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
```

Div

Divides each element of the first vector by corresponding element of the second vector.

Syntax

```
IppStatus ippsDiv_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsDiv_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsDiv_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsDiv_64f_A26 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsDiv_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsDiv_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsDiv_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsDiv_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
```

```
IppStatus ippsDiv_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsDiv_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
IppStatus ippsDiv_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
IppStatus ippsDiv_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

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 flavors `ippsDiv_32f_A11`and `ippsDiv_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsDiv_32f_A21` and `ippsDiv_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsDiv_32f_A24` and `ippsDiv_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsDiv_64f_A26` and `ippsDiv_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsDiv_64f_A50` and `ippsDiv_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsDiv_64f_A53` and `ippsDiv_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

pDst[n] = (*pSrc1*[n]) /(*pSrc2*[n]), 0 ≤ n < *len*.

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsSingularity	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc2* is equal to 0.

Example

The example below shows how to use the function ippsDiv.

```
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
```

Sqrt

Computes square root of each vector element.

Syntax

```
IppStatus ippsSqrt_32f_A11 ( const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSqrt_32f_A21 ( const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSqrt_32f_A24 ( const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSqrt_64f_A26 ( const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSqrt_64f_A50 ( const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSqrt_64f_A53 ( const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsSqrt_32fc_A11 ( const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
```

```
IppStatus ippsSqrt_32fc_A21 ( const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSqrt_32fc_A24 ( const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSqrt_64fc_A26 ( const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsSqrt_64fc_A50 ( const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsSqrt_64fc_A53 ( const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsSqrt_32f_A11`and `ippsSqrt_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsSqrt_32f_A21` and `ippsSqrt_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsSqrt_32f_A24` and `ippsSqrt_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSqrt_64f_A26` and `ippsSqrt_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsSqrt_64f_A50` and `ippsSqrt_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsSqrt_64f_A53` and `ippsSqrt_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = (pSrc[n])^{1/2}, 0 \le n < len.$

Return Values

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

IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.

Example

The example below shows how to use the function ippsSqrt.

```
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
```

InvSqrt

Computes inverse square root of each vector element.

Syntax

```
IppStatus ippsInvSqrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

ippvm.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

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_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

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:

$pDst[n] = (pSrc[n])^{-1/2}, 0 \le n < len$.

Return Values

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

Example

The example below shows how to use the function `ippsInvSqrt`.

```
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
```

Cbrt

Computes cube root of each vector element.

Syntax

```
IppStatus ippsCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCbrt_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

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_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

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:

$pDst[n] = (pSrc[n])^{1/3}, 0 \le n < len$.

Return Values

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

Example

The example below shows how to use the function `ippsCbrt`.

```
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;
}

Output results:

 ippsCbrt_32f_A21:
 x = 6456.801 4932.096 -6517.838 7178.869
 y = 18.621 17.022 -18.680 19.291
```

InvCbrt

Computes inverse cube root of each vector element.

Syntax

```
IppStatus ippsInvCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

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

len	Number of elements in the vectors.

Description

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_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

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:

$pDst[n] = (pSrc[n])^{-1/3}, 0 \le n < len.$

Return Values

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

Example

The example below shows how to use the function `ippsInvCbrt`.

```
IppStatus ippsInvCbrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {914.120, 3644.584, 1473.214, 1659.070};
Ipp32f                  y[4];

    IppStatus st = ippsInvCbrt_32f_A21( x, y, 4 );

    printf(" ippsInvCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInvCbrt_32f_A21:
 x = 914.120 3644.584 1473.214 1659.070
 y = 0.103 0.065 0.088 0.084
```

Pow2o3

Computes the value of each vector element raised to the power of 2/3.

Syntax

```
IppStatus ippsPow2o3_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow2o3_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow2o3_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow2o3_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsPow2o3_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsPow2o3_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the value of each vector element of the vector *pSrc* raised to 2/3 power and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsPow2o3_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsPow2o3_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsPow2o3_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 `ippsPow2o3_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsPow2o3_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsPow2o3_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Return Values

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

Pow3o2

Computes the value of each vector element raised to the power of 3/2.

Syntax

```
IppStatus ippsPow3o2_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow3o2_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow3o2_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow3o2_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsPow3o2_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsPow3o2_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the value of each vector element of the vector *pSrc* raised to 3/2 power and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsPow3o2_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsPow3o2_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsPow3o2_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 `ippsPow3o2_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsPow3o2_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsPow3o2_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.

Pow

Raises each element of the first vector to the power of corresponding element of the second vector.

Syntax

```
IppStatus ippsPow_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
```

```
IppStatus ippsPow_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
```

```
IppStatus ippsPow_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
```

```
IppStatus ippsPow_64f_A26 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
```

```
IppStatus ippsPow_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsPow_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsPow_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsPow_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsPow_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst,
Ipp32s len);
IppStatus ippsPow_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
IppStatus ippsPow_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
IppStatus ippsPow_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst,
Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function raises each element of vector *pSrc1* to the power of the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsPow_32f_A11`and `ippsPow_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsPow_32f_A21` and `ippsPow_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsPow_32f_A24` and `ippsPow_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsPow_64f_A26` and `ippsPow_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsPow_64f_A50` and `ippsPow_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsPow_64f_A53` and `ippsPow_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Note that for `ippsPow` complex functions there may be argument ranges where the accuracy specification does not hold.

The computation is performed as follows:

$pDst[n] = (pSrc1[n])^{pSrc2[n]}, 0 \le n < len.$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one pair of the source elements meets the following condition: element of *pSrc1* is finite, less than 0, and element of *pSrc2* is finite, non-integer.
`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of *pSrc1* is equal to 0, and element of *pSrc2* is integer and less than 0.

Example

The example below shows how to use the function `ippsPow`.

```
IppStatus ippsPow_32f_A21_sample(void)
{
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
Ipp32f                  y[4];

    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
```

Powx

Raises each element of a vector to a constant power.

Syntax

```
IppStatus ippsPowx_32f_A11 (const Ipp32f* pSrc1, const Ipp32f ConstValue, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsPowx_32f_A21 (const Ipp32f* pSrc1, const Ipp32f ConstValue, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsPowx_32f_A24 (const Ipp32f* pSrc1, const Ipp32f ConstValue, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsPowx_64f_A26 (const Ipp64f* pSrc1, const Ipp64f ConstValue, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsPowx_64f_A50 (const Ipp64f* pSrc1, const Ipp64f ConstValue, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsPowx_64f_A53 (const Ipp64f* pSrc1, const Ipp64f ConstValue, Ipp64f* pDst,
Ipp32s len);
```

```
IppStatus ippsPowx_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue, Ipp32fc*
pDst, Ipp32s len);
IppStatus ippsPowx_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue, Ipp32fc*
pDst, Ipp32s len);
IppStatus ippsPowx_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue, Ipp32fc*
pDst, Ipp32s len);
IppStatus ippsPowx_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc ConstValue, Ipp64fc*
pDst, Ipp32s len);
IppStatus ippsPowx_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc ConstValue, Ipp64fc*
pDst, Ipp32s len);
IppStatus ippsPowx_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc ConstValue, Ipp64fc*
pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the source vector.
ConstValue	Constant value.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function raises each element of the vector *pSrc1* to the constant power *ConstValue* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsPowx_32f_A11`and `ippsPowx_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsPowx_32f_A21` and `ippsPowx_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsPowx_32f_A24` and `ippsPowx_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsPowx_64f_A26` and `ippsPowx_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsPowx_64f_A50` and `ippsPowx_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsPowx_64f_A53` and `ippsPowx_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Note that for `ippsPowx` complex functions there may be argument ranges where the accuracy specification does not hold.

The computation is performed as follows:

$pDst[n] = (pSrc[n])^{ConstValue}, 0 \le n < len.$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one pair of the elements meets the following condition: element of *pSrc1* is finite, less than 0, and *ConstValue* is finite, non-integer.
`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of *pSrc1* is equal to 0, and *ConstValue* is integer and less than 0.

Example

The example below shows how to use the function `ippsPowx`.

```
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
```

Hypot

Computes a square root of sum of two squared elements.

Syntax

```
IppStatus ippsHypot_32f_A11 (const Ipp32f* pSrc1, const Ipp32f pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsHypot_32f_A21 (const Ipp32f* pSrc1, const Ipp32f pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsHypot_32f_A24 (const Ipp32f* pSrc1, const Ipp32f pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsHypot_64f_A26 (const Ipp64f* pSrc1, const Ipp64f pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsHypot_64f_A50 (const Ipp64f* pSrc1, const Ipp64f pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsHypot_64f_A53 (const Ipp64f* pSrc1, const Ipp64f pSrc2, Ipp64f* pDst,
Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes square of each element of the *pSrc1* and *pSrc2* vectors, sums corresponding elements, computes square roots of each sum and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsHypot_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsHypot_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsHypot_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsHypot_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsHypot_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsHypot_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = ((pSrc1[n])^2 + (pSrc2[n])^2)^{1/2}, 0 \le n < len.$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1* or *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function `ippsHypot`.

```
IppStatus ippsHypot_32f_A21_sample(void)
{
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252}
const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
Ipp32f y[4];

    IppStatus st = ippsHypot_32f_A21( x1, x2, y, 4 );
    printf(" ippsHypot_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}
```

```
Output results:

 ippsHypot_32f_A21:
 x1 = 0.483 0.565 0.776 0.252
 x2 = 0.823 0.991 0.411 0.692
 y  = 0.954 1.141 0.878 0.736
```

Exponential and Logarithmic Functions

Exp

Raises e to the power of each vector element.

Syntax

```
IppStatus ippsExp_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExp_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExp_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExp_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExp_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExp_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

```
IppStatus ippsExp_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsExp_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsExp_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsExp_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsExp_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsExp_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function raises *e* to the power of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsExp_32f_A11`and `ippsExp_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsExp_32f_A21` and `ippsExp_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsExp_32f_A24` and `ippsExp_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsExp_64f_A26` and `ippsExp_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsExp_64f_A50` and `ippsExp_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsExp_64f_A53` and `ippsExp_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsOverflow`	In real functions, indicates a warning that the function overflows, that is, at least one of elements of *pSrc* is greater than `Ln(FPMAX)`, where `FPMAX` is the maximum representable floating-point number.
`IppStsUnderflow`	In real functions, indicates a warning that the function underflows, that is, at least one of elements of *pSrc* is less than `Ln(FPMIN)`, where `FPMIN` is the minimum positive floating-point value.

Example

The example below shows how to use the function `ippsExp`.

```
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
```

Expm1

Computes e raised to the power of each vector element and decreased by 1.

Syntax

```
IppStatus ippsExpm1_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExpm1_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExpm1_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExpm1_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExpm1_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExpm1_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes e raised to the power of each vector element of *pSrc* and decreased by 1, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsExpm1_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsExpm1_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsExpm1_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 `ippsExpm1_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsExpm1_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsExpm1_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Return Values

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

Ln

Computes natural logarithm of each vector element.

Syntax

```
IppStatus ippsLn_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLn_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLn_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLn_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLn_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLn_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsLn_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLn_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLn_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLn_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsLn_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsLn_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

ippvm.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsLn_32f_A11`and `ippsLn_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsLn_32f_A21` and `ippsLn_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsLn_32f_A24` and `ippsLn_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsLn_64f_A26` and `ippsLn_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsLn_64f_A50` and `ippsLn_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsLn_64f_A53` and `ippsLn_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.
`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Example

The example below shows how to use the function `ippsLn`.

```
IppStatus ippsLn_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.188, 3.841, 5.363, 5.755};
Ipp32f                  y[4];

    IppStatus st = ippsLn_32f_A21( x, y, 4 );

    printf(" ippsLn_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:

 ippsLn_32f_A21:
 x = 0.188 3.841 5.363 5.755
 y = -1.670 1.346 1.680 1.750
```

Log10

Computes common logarithm of each vector element.

Syntax

```
IppStatus ippsLog10_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog10_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog10_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog10_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog10_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog10_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog10_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLog10_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLog10_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLog10_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsLog10_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsLog10_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsLog10_32f_A11`and `ippsLog10_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsLog10_32f_A21` and `ippsLog10_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsLog10_32f_A24` and `ippsLog10_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsLog10_64f_A26` and `ippsLog10_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsLog10_64f_A50` and `ippsLog10_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsLog10_64f_A53` and `ippsLog10_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \log_{10}(pSrc[n]), 0 \le n < len.$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 0.
`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

Example

The example below shows how to use the function `ippsLog10`.

```
IppStatus ippsLog10_32f_A21_sample(void)
{
const Ipp32f x[4] = {6.057, 6.111, 1.746, 6.664};
Ipp32f          y[4];

    IppStatus st = ippsLog10_32f_A21( x, y, 4 );

    printf(" ippsLog10_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsLog10_32f_A21:
 x = 6.057 6.111 1.746 6.664
 y = 0.782 0.786 0.242 0.824
```

Log1p

Computes natural logarithm of each vector element decreased by 1.

Syntax

```
IppStatus ippsLog1p_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog1p_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog1p_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog1p_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog1p_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog1p_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a natural logarithm of each vector element of *pSrc* decreased by 1, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor ippsLog1p_32f_A11 guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor ippsLog1p_32f_A21 guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor ippsLog1p_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 ippsLog1p_64f_A26 guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor ippsLog1p_64f_A50 guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor ippsLog1p_64f_A53 guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than -1.
IppStsSingularity	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to -1.

Trigonometric Functions

Cos

Computes cosine of each vector element.

Syntax

```
IppStatus ippsCos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCos_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

```
IppStatus ippsCos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

```
IppStatus ippsCos_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCos_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCos_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCos_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCos_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCos_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsCos_32f_A11`and `ippsCos_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsCos_32f_A21` and `ippsCos_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsCos_32f_A24` and `ippsCos_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsCos_64f_A26` and `ippsCos_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsCos_64f_A50` and `ippsCos_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsCos_64f_A53` and `ippsCos_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

`ippStsNoErr`	Indicates no error.

ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

Example

The example below shows how to use the function ippsCos.

```
IppStatus ippsCos_32f_A21_sample(void)
{
const Ipp32f x[4] = {-984.222, -2957.549, -8859.218, 2153.691};
Ipp32f                  y[4];

    IppStatus st = ippsCos_32f_A21( x, y, 4 );

    printf(" ippsCos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:

 ippsCos_32f_A21:
 x = -984.222 -2957.549 -8859.218 2153.691
 y = -0.619 -0.258 0.997 0.129
```

Sin

Computes sine of each vector element.

Syntax

```
IppStatus ippsSin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSin_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsSin_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSin_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSin_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSin_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

```
IppStatus ippsSin_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsSin_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsSin_32f_A11`and `ippsSin_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsSin_32f_A21` and `ippsSin_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsSin_32f_A24` and `ippsSin_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSin_64f_A26` and `ippsSin_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsSin_64f_A50` and `ippsSin_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsSin_64f_A53` and `ippsSin_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to `± INF`.

Example

The example below shows how to use the function `ippsSin`.

```
IppStatus ippsSin_32f_A21_sample(void)
{
const Ipp32f x[4] = {5666.372, 6052.125, 397.656, -3960.997};
Ipp32f                  y[4];

    IppStatus st = ippsSin_32f_A21( x, y, 4 );

    printf(" ippsSin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSin_32f_A21:
 x = 5666.372 6052.125 397.656 -3960.997
 y = -0.873 0.988 0.970 -0.524
```

SinCos

Computes sine and cosine of each vector element.

Syntax

```
IppStatus ippsSinCos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst1, Ipp32f* pDst2, Ipp32s len);
IppStatus ippsSinCos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst1, Ipp32f* pDst2, Ipp32s len);
IppStatus ippsSinCos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst1, Ipp32f* pDst2, Ipp32s len);
IppStatus ippsSinCos_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst1, Ipp64f* pDst2, Ipp32s len);
IppStatus ippsSinCos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst1, Ipp64f* pDst2, Ipp32s len);
IppStatus ippsSinCos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst1, Ipp64f* pDst2, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the first source vector.
pDst1	Pointer to the destination vector for sine values.
pDst2	Pointer to the destination vector for cosine values.
len	Number of elements in the vectors.

Description

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_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

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]), $0 \le n < len$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pDst1* or *pDst2* or *pSrc* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the *pSrc* elements is equal to ± `INF`.

Example

The example below shows how to use the function `ippsSinCos`.

```
IppStatus ippsSinCos_32f_A21_sample(void)
{
const Ipp32f x[4] = {3857.845, -3939.024, -1468.856, -8592.486};
Ipp32f          y1[4];
Ipp32f          y2[4];

    IppStatus st = ippsSinCos_32f_A21( x, y1, y2, 4 );

    printf(" ippsSinCos_32f_A21:\n");
    printf(" x  = %.3f %.3f %.3f %.3f \n", x[0],  x[1],  x[2],  x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
    return st;
}
```

```
Output results:

 ippsSinCos_32f_A21:
 x  = 3857.845 -3939.024 -1468.856 -8592.486
 y1 = -0.031 0.508 0.987 0.228
 y2 = 1.000 0.861 0.161 -0.974
```

CIS

Computes complex exponent of each vector element.

Syntax

```
IppStatus ippsCIS_32fc_A11 (const Ipp32f* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCIS_32fc_A21 (const Ipp32f* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCIS_32fc_A24 (const Ipp32f* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCIS_64fc_A26 (const Ipp64f* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCIS_64fc_A50 (const Ipp64f* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCIS_64fc_A53 (const Ipp64f* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a complex exponent of each vector element of *pSrc* and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsCIS_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCIS_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCIS_32fc_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 `ippsCIS_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsCIS_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCIS_64fc_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of the *pSrc* elements is equal to ±`INF`.

Tan

Computes tangent of each vector element.

Syntax

```
IppStatus ippsTan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTan_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsTan_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
```

```
IppStatus ippsTan_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTan_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTan_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsTan_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsTan_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsTan_32f_A11`and `ippsTan_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsTan_32f_A21` and `ippsTan_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsTan_32f_A24` and `ippsTan_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsTan_64f_A26` and `ippsTan_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsTan_64f_A50` and `ippsTan_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsTan_64f_A53` and `ippsTan_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

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

IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is equal to ± INF.

Example

The example below shows how to use the function ippsTan.

```
IppStatus ippsTan_32f_A21_sample(void)
{
const Ipp32f x[4] = {7519.456, 4533.524, 9118.015, 8514.359};
Ipp32f                  y[4];

    IppStatus st = ippsTan_32f_A21( x, y, 4 );

    printf(" ippsTan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsTan_32f_A21:
 x = 7519.456 4533.524 9118.015 8514.359
 y = -18.656 0.209 2.028 0.750
```

Acos

Computes inverse cosine of each vector element.

Syntax

```
IppStatus ippsAcos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcos_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsAcos_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAcos_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAcos_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAcos_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAcos_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAcos_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

ippvm.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the inverse cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors ippsAcos_32f_A11and ippsAcos_32cf_A11 guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors ippsAcos_32f_A21 and ippsAcos_32fc_A21 guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors ippsAcos_32f_A24 and ippsAcos_32fc_A24 guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors ippsAcos_64f_A26 and ippsAcos_64fc_A26 guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors ippsAcos_64f_A50 and ippsAcos_64fc_A50 guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors ippsAcos_64f_A53 and ippsAcos_64fc_A53 guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has an absolute value greater than 1.

Example

The example below shows how to use the function `ippsAcos`.

```
IppStatus ippsAcos_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.079, -0.715, -0.076, -0.529};
Ipp32f          y[4];

    IppStatus st = ippsAcos_32f_A21( x, y, 4 );

    printf(" ippsAcos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAcos_32f_A21:
 x = 0.079 -0.715 -0.076 -0.529
 y = 1.492  2.368  1.647  2.129
```

Asin

Computes inverse sine of each vector element.

Syntax

```
IppStatus ippsAsin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsin_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsAsin_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsin_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsin_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsin_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAsin_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAsin_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the inverse sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAsin_32f_A11`and `ippsAsin_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAsin_32f_A21` and `ippsAsin_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAsin_32f_A24` and `ippsAsin_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAsin_64f_A26` and `ippsAsin_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAsin_64f_A50` and `ippsAsin_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAsin_64f_A53` and `ippsAsin_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{asin}(pSrc[n]), 0 \le n < len$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has an absolute value greater than 1.

Example

The example below shows how to use the function `ippsAsin`.

```
IppStatus ippsAsin_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.724, -0.581, 0.559, 0.687};
Ipp32f                  y[4];

    IppStatus st = ippsAsin_32f_A21( x, y, 4 );

    printf(" ippsAsin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;


Output results:

 ippsAsin_32f_A21:
 x = 0.724 -0.581 0.559 0.687
 y = 0.810 -0.620 0.594 0.758
```

Atan

Computes inverse tangent of each vector element.

Syntax

```
IppStatus ippsAtan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtan_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsAtan_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtan_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtan_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtan_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAtan_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAtan_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the inverse tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAtan_32f_A11`and `ippsAtan_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAtan_32f_A21` and `ippsAtan_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAtan_32f_A24` and `ippsAtan_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAtan_64f_A26` and `ippsAtan_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAtan_64f_A50` and `ippsAtan_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAtan_64f_A53` and `ippsAtan_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{atan}(pSrc[n]), 0 \le n < len$.

Return Values

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

Example

The example below shows how to use the function `ippsAtan`.

```
IppStatus ippsAtan_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.994, 0.999, 0.223, -0.215};
Ipp32f                  y[4];

    IppStatus st = ippsAtan_32f_A21( x, y, 4 );

    printf(" ippsAtan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:

 ippsAtan_32f_A21:
 x = 0.994 0.999 0.223 -0.215
 y = 0.782 0.785 0.219 -0.212
```

Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

```
IppStatus ippsAtan2_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsAtan2_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsAtan2_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
Ipp32s len);
IppStatus ippsAtan2_64f_A26 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsAtan2_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
IppStatus ippsAtan2_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc1	Pointer to the first source vector.
pSrc2	Pointer to the second source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

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_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

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] = atan2(*pSrc1*[n], *pSrc2*[n]), $0 \le n < len$.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc1*, *pSrc2* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function `ippsAtan2`.

```
IppStatus ippsAtan2_32f_A21_sample(void)
{
const Ipp32f x1[4] = {1.492, 1.700, 1.147, 1.142};
const Ipp32f x2[4] = {1.064, 1.505, 1.950, 1.905};
Ipp32f                  y[4];

    IppStatus st = ippsAtan2_32f_A21( x1, x2, y, 4 );

    printf(" ippsAtan2_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}
```

```
Output results:

 ippsAtan2_32f_A21:
 x1 = 1.492 1.700 1.147 1.142
 x2 = 1.064 1.505 1.950 1.905
 y  = 0.951 0.846 0.532 0.540
```

Hyperbolic Functions

Cosh

Computes hyperbolic cosine of each vector element.

Syntax

```
IppStatus ippsCosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCosh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsCosh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCosh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
```

```
IppStatus ippsCosh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCosh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCosh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCosh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsCosh_32f_A11`and `ippsCosh_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsCosh_32f_A21` and `ippsCosh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsCosh_32f_A24` and `ippsCosh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsCosh_64f_A26` and `ippsCosh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsCosh_64f_A50` and `ippsCosh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsCosh_64f_A53` and `ippsCosh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

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

`IppStsOverflow`	In real functions, indicates a warning that the function overflows, that is, at least one of elements of *pSrc* has the absolute value greater than `Ln(FPMAX) + Ln(2)`, where `FPMAX` is the maximum representable floating-point number.

Example

The example below shows how to use the function `ippsCosh`.

```
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
```

Sinh

Computes hyperbolic sine of each vector element.

Syntax

```
IppStatus ippsSinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSinh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsSinh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSinh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSinh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSinh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsSinh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

```
IppStatus ippsSinh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

ippvm.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors ippsSinh_32f_A11and ippsSinh_32cf_A11 guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors ippsSinh_32f_A21 and ippsSinh_32fc_A21 guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors ippsSinh_32f_A24 and ippsSinh_32fc_A24 guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors ippsSinh_64f_A26 and ippsSinh_64fc_A26 guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors ippsSinh_64f_A50 and ippsSinh_64fc_A50 guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors ippsSinh_64f_A53 and ippsSinh_64fc_A53 guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

pDst[n] = sinh(*pSrc*[n]), 0 ≤ n < *len*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsOverflow	In real functions, indicates a warning that the function overflows, that is, at least one of elements of *pSrc* has the absolute value greater than Ln(FPMAX) + Ln(2), where FPMAX is the maximum representable floating-point number.

Example

The example below shows how to use the function `ippsSinh`.

```
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
```

Tanh

Computes hyperbolic tangent of each vector element.

Syntax

```
IppStatus ippsTanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTanh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsTanh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTanh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTanh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTanh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsTanh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsTanh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the hyperbolic tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsTanh_32f_A11`and `ippsTanh_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsTanh_32f_A21` and `ippsTanh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsTanh_32f_A24` and `ippsTanh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsTanh_64f_A26` and `ippsTanh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsTanh_64f_A50` and `ippsTanh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsTanh_64f_A53` and `ippsTanh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

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

Example

The example below shows how to use the function `ippsTanh`.

```
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
```

Acosh

Computes inverse hyperbolic cosine of each vector element.

Syntax

```
IppStatus ippsAcosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcosh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsAcosh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAcosh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAcosh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAcosh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAcosh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAcosh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

ippvm.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the inverse (nonnegative) hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors ippsAcosh_32f_A11and ippsAcosh_32cf_A11 guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors ippsAcosh_32f_A21 and ippsAcosh_32fc_A21 guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors ippsAcosh_32f_A24 and ippsAcosh_32fc_A24 guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors ippsAcosh_64f_A26 and ippsAcosh_64fc_A26 guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors ippsAcosh_64f_A50 and ippsAcosh_64fc_A50 guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors ippsAcosh_64f_A53 and ippsAcosh_64fc_A53 guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \mathrm{acosh}(pSrc[n]), 0 \le n < len.$

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.
IppStsDomain	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* is less than 1.

Example

The example below shows how to use the function `ippsAcosh`.

```
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
```

Asinh

Computes inverse hyperbolic sine of each vector element.

Syntax

```
IppStatus ippsAsinh_32f_A11(const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsinh_32f_A21(const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsinh_32f_A24(const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsinh_64f_A26(const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsinh_64f_A50(const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsinh_64f_A53(const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsAsinh_32fc_A11(const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsinh_32fc_A21(const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsinh_32fc_A24(const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsinh_64fc_A26(const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAsinh_64fc_A50(const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAsinh_64fc_A53(const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the inverse hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAsinh_32f_A11`and `ippsAsinh_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAsinh_32f_A21` and `ippsAsinh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAsinh_32f_A24` and `ippsAsinh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAsinh_64f_A26` and `ippsAsinh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAsinh_64f_A50` and `ippsAsinh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAsinh_64f_A53` and `ippsAsinh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{asinh}(pSrc[n]), 0 \le n < len$.

Return Values

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

Example

The example below shows how to use the function `ippsAsinh`.

```
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
```

Atanh

Computes inverse hyperbolic tangent of each vector element.

Syntax

```
IppStatus ippsAtanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtanh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

IppStatus ippsAtanh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtanh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtanh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtanh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAtanh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAtanh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the inverse hyperbolic tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAtanh_32f_A11` and `ippsAtanh_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAtanh_32f_A21` and `ippsAtanh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAtanh_32f_A24` and `ippsAtanh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAtanh_64f_A26` and `ippsAtanh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAtanh_64f_A50` and `ippsAtanh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAtanh_64f_A53` and `ippsAtanh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

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

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when len is less than or equal to 0.
`IppStsDomain`	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of *pSrc* has absolute value greater than 1.
`IppStsSingularity`	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* has absolute value equal to 1.

Example

The example below shows how to use the function `ippsAtanh`.

```
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
```

Special Functions

Erf

Computes the error function value.

Syntax

```
IppStatus ippsErf_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErf_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErf_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErf_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErf_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErf_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

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_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

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:

$pDst[n] = erf(pSrc[n]), 0 \le n < len$, where

Return Values

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

Example

The example below shows how to use the function `ippsErf`.

```
IppStatus ippsErf_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f          y[4];

    IppStatus st = ippsErf_32f_A21( x, y, 4 );

    printf(" ippsErf_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:

 ippsErf_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.835 0.764 -0.474 -0.479
```

Erfc

Computes the complementary error function value.

Syntax

```
IppStatus ippsErfc_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfc_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfc_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfc_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfc_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfc_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.

pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

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_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

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] = erfc(*pSrc*[n]), 0 ≤ n < *len*, where

$$\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt \ .$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsUnderflow`	Indicates a warning that the function underflows, that is, at least one element of *pSrc* is less than some threshold value, where the function result is less than the minimum positive floating-point value in target precision.

Example

The example below shows how to use the function `ippsErfc`.

```
IppStatus ippsErfc_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f          y[4];

    IppStatus st = ippsErfc_32f_A21( x, y, 4 );

    printf(" ippsErfc_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:

 ippsErfc_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

CdfNorm

Computes the cumulative normal distribution function values of vector element.

Syntax

```
IppStatus ippsCdfNorm_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the cumulative normal distribution function values of *pSrc* element and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsCdfNorm_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCdfNorm_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCdfNorm_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 `ippsCdfNorm_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsCdfNorm_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCdfNorm_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] = CdfNorm(*pSrc*[n]), 0 ≤ n < *len*, where

$$\mathrm{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt \ .$$

Example 12-37 below shows how to use the function `ippsCdfNorm`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`IppStsUnderflow`	Indicates a warning that the function underflows, that is, at least one element of *pSrc* is less than some threshold value, where the function result is less than the minimum positive floating-point value in the target precision.

Example 12-37. Using ippsCdfNorm Function

```
IppStatus ippsCdfNorm_32f_A24_sample(void)
{
const Ipp32f x[4] = {+4.885, -0.543, -3.809, -4.953};
Ipp32f          y[4];

    IppStatus st = ippsCdfNorm_32f_A24( x, y, 4 );

    printf(" ippsCdfNorm_32f_A24:\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:

 ippsCdfNorm_32f_A24:
 x = +4.885 -0.543 -3.809 -4.953
 y = +1.000 +0.294 +0.000 +0.000
```

ErfInv

Computes the inverse error function value.

Syntax

```
IppStatus ippsErfInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfInv_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

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

len	Number of elements in the vectors.

Description

This function computes the inverse error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErfInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErfInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErfInv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErfInv_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsErfInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErfInv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

pDst[n] = erfinv(*pSrc*[n]), $0 \le n < len$, where erfinv(x) = $\mathrm{erf}^{-1}(x)$, and erf(x) denotes the error function defined as given by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of *pSrc* elements has the absolute value greater than 1.
`ippStsSingularity`	Indicates a warning that the argument is a singularity point, that is, at least one of the elements of *pSrc* has the absolute value equal to 1.

Example

The example below shows how to use the function `ippsErfcInv`.

```
IppStatus ippsErfInv_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.842, 0.638, -0.345, -0.774};
Ipp32f          y[4];

    IppStatus st = ippsErfInv_32f_A21( x, y, 4 );

    printf(" ippsErfInv_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsErfInv_32f_A21:
 x = -0.842 0.638 -0.345 -0.774
 y = -0.998 0.645 -0.316 -0.856
```

ErfcInv

Computes the inverse complementary error function value of vector element.

Syntax

```
IppStatus ippsErfcInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfcInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfcInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfcInv_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfcInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfcInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the inverse complementary error function value of each *pSrc* vector element and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErfcInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErfcInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErfcInv_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 `ippsErfcInv_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsErfcInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErfcInv_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] = erfcinv(*pSrc*[n]), 0 ≤ n < *len*, where erfcinv(*x*) = erfinv(1 – *x*, and erfinv(*x*) denotes the error function defined as given by:

$$\mathrm{erfinv}(x) = \mathrm{erf}^{-1}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt ,$$

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of *pSrc* elements is outside the function domain [0; 2].
`ippStsSingularity`	Indicates a warning that the argument is a singularity point, that is, at least one of the elements of *pSrc* is equal to 0 or 2.

Example

The example below shows how to use the function `ippsErfcInv`.

```
IppStatus ippsErfcInv_32f_A24_sample(void)
{
const Ipp32f x[4] = {+0.885, +0.543, +1.809, +0.953};
Ipp32f          y[4];

    IppStatus st = ippsErfcInv_32f_A24( x, y, 4 );

    printf(" ippsErfcInv_32f_A24:\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:

 ippsErfcInv_32f_A24:
 x = +0.885 +0.543 +1.809 +0.953
 y = +0.102 +0.430 -0.925 +0.042
```

CdfNormInv

Computes the inverse cumulative normal distribution function values of vector elements.

Syntax

```
IppStatus ippsCdfNormInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNormInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNormInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNormInv_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCdfNormInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCdfNormInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes the inverse cumulative normal distribution function values of *pSrc* vector elements and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsCdfNormInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCdfNormInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCdfNormInv_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 `ippsCdfNormInv_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsCdfNormInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCdfNormInv_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] = CdfNormInv(*pSrc*[n]), $0 \le n < len$, where CdfNormInv(*x*) = $\mathrm{CdfNorm}^{-1}(x)$, and CdfNorm(*x*) denotes the cumulative normal distribution function:

$$\mathrm{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt \ .$$

Example 12-40 below shows how to use the function `ippsCdfNormInv`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when *pSrc* or *pDst* pointer is `NULL`.
`ippStsSizeErr`	Indicates an error when *len* is less than or equal to 0.
`ippStsDomain`	Indicates a warning that the argument is out of the function domain, that is, at least one of *pSrc* elements is outside the function domain [0; 1].
`ippStsSingularity`	Indicates a warning that the argument is a singularity point, that is, at least one of the elements of *pSrc* is equal to 0 or 1.

Example 12-40. Using ippsCdfNormInv Function

```
IppStatus ippsCdfNormInv_32f_A24_sample(void)
{
const Ipp32f x[4] = {+0.085, +0.543, +1.809, +0.953};
Ipp32f          y[4];

    IppStatus st = ippsCdfNormInv_32f_A24( x, y, 4 );

    printf(" ippsCdfNormInv_32f_A24:\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:

 ippsCdfNormInv_32f_A24:
 x = +0.085 +0.543 +1.809 +0.953
 y = -1.372 +0.108 +0.874 +1.675
```

Rounding Functions

Floor

Computes integer value rounded toward minus infinity for each vector element.

Syntax

```
IppStatus ippsFloor_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsFloor_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

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

len	Number of elements in the vectors.

Description

This function computes an integer value rounded towards minus infinity for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Return Values

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

Example

The example below shows how to use the function ippsFloor.

```
IppStatus ippsFloor_32f_ sample(void)
{
    const Ipp32f x[4] = {-0.883, -0.265, 0.176, 0.752};
    Ipp32f y[4];
    IppStatus st = ippsFloor_32f ( x, y, 4 );
    printf(" ippsFloor_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:
ippsFloor_32f:
x = -0.883 -0.265 0.176 0.752
y = -1.000 -1.000 0.000 0.000
```

Frac

Computes a signed fractional part for each element of a vector.

Syntax

```
IppStatus ippsFrac_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsFrac_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

ippvm.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a fractional part of each element of the *pSrc* vector. The result is stored in the corresponding element of the *pDst* vector.

Return Values

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

Example

The example below shows how to use the ippsFrac function.

```
IppStatus ippsFrac_32f_sample(void)
{
const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
Ipp32f y[4];
IppStatus st = ippsFrac_32f ( x, y, 4 );
printf(" ippsFrac_32f:\n");
printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
return st;
}
```

Result:

```
ippsFrac_32f:
x = -1.883 -0.265 0.176 1.752
y = -0.883 -0.265 0.176 0.752
```

Ceil

Computes integer value rounded toward plus infinity for each vector element.

Syntax

```
IppStatus ippsCeil_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCeil_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

ippvm.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes an integer value rounded towards plus infinity for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Return Values

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

Example

The example below shows how to use the function ippsCeil.

```
IppStatus ippsCeil_32f_sample(void)
{
    const Ipp32f x[4] = {-0.883, -0.265, 0.176, 0.752};
    Ipp32f y[4];
    IppStatus st = ippsCeil_32f ( x, y, 4 );
    printf(" ippsCeil_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:
ippsCeil_32f:
x = -0.883 -0.265 0.176 0.752
y = 0.000 0.000 1.000 1.000
```

Trunc

Computes integer value rounded toward zero for each vector element.

Syntax

```
IppStatus ippsTrunc_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTrunc_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

ippvm.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes an integer value rounded towards zero for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Return Values

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

Example

The example below shows how to use the function ippsTrunc.

```
IppStatus ippsTrunc_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y[4];
    IppStatus st = ippsTrunc_32f ( x, y, 4 );
    printf(" ippsTrunc_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:
ippsTrunc_32f:
x = -1.883 -0.265 0.176 1.752
y = -1.000 0.000 0.000 1.000
```

Round

Computes integer value rounded to nearest for each vector element.

Syntax

```
IppStatus ippsRound_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsRound_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a rounded to the nearest integer value for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*. Halfway values, that is, 0.5, -1.5, and the like, are rounded off away from zero, that is, 0.5 -> 1, -1.5 -> -2, and so on.

Return Values

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

Example

The example below shows how to use the function `ippsRound`.

```
IppStatus ippsRound_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y[4];
    IppStatus st = ippsRound_32f ( x, y, 4 );
    printf(" ippsRound_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

```
Output results:
ippsRound_32f:
x = -1.883 -0.265 0.176 1.752
y = -2.000  0.000 0.000 2.000
```

NearbyInt

Computes rounded integer value in current rounding mode for each vector element.

Syntax

```
IppStatus ippsNearbyInt_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsNearbyInt_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a rounded integer value in a current rounding mode for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Return Values

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

Example

The example below shows how to use the ippsNearbyInt function.

```
#include <fenv.h>

void ippsNearbyInt_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4], y2[4];

    fesetround(FE_TONEAREST);
    ippsNearbyInt_32f ( x, y1, 4 );

    fesetround(FE_TOWARDZERO);
    ippsNearbyInt_32f ( x, y2, 4 );

    printf(" ippsNearbyInt_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
}

Output results:
ippsNearInt_32f:
x = -1.883 -0.265 0.176 1.752
y1 = -2.000  0.000 0.000 2.000
y2 = -1.000  0.000 0.000 1.000
```

Rint

Computes rounded integer value in current rounding mode for each vector element with inexact result exception raised for each changed value.

Syntax

```
IppStatus ippsRint_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsRint_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the vectors.

Description

This function computes a rounded integer value in a current rounding mode for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst* raising inexact result exception if the value has changed.

Return Values

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

Example

The example below shows how to use the `ippsRint` function.

```
#include <fenv.h>

void ippsRint_32f_ sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4], y2[4];

    fesetround(FE_TONEAREST);
    ippsRint_32f ( x, y1, 4 );

    fesetround(FE_TOWARDZERO);
    ippsRint_32f ( x, y2, 4 );

    printf(" ippsRint_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
}
```

```
Output results:
ippsRint_32f:
x = -1.883 -0.265 0.176 1.752
y1 = -2.000  0.000 0.000 2.000
y2 = -1.000  0.000 0.000 1.000
```

Modf

Computes truncated integer value and remaining fraction part for each vector element.

Syntax

```
IppStatus ippsModf_32f (const Ipp32f* pSrc, Ipp32f* pDst1, Ipp32f* pDst2, Ipp32s len);
IppStatus ippsModf_64f (const Ipp64f* pSrc, Ipp64f* pDst1, Ipp64f* pDst2, Ipp32s len);
```

Include Files

`ippvm.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

pSrc	Pointer to the source vector.
pDst1	Pointer to the first destination vector.
pDst2	Pointer to the second destination vector.
len	Number of elements in the vectors.

Description

This function computes a truncated value and a remainder of each element of the vector *pSrc*. The truncated integer value is stored in the corresponding element of the *pDst1* vector and the remainder is stored in the corresponding element of the *pDst2* vector.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when *pSrc* or *pDst1* or *pDst2* pointer is NULL.
ippStsSizeErr	Indicates an error when *len* is less than or equal to 0.

Example

The example below shows how to use the function ippsModf.

```
IppStatus ippsModf_32f_sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4], y2[4];
    IppStatus st = ippsModf_32f ( x, y1, y2, 4 );
    printf(" ippsModf_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
    return st;
}
```

```
Output results:
ippsModf_32f:
x = -1.883 -0.265 0.176 1.752
y1 = -1.000 0.000 0.000 1.000
y2 = -0.883 -0.265 0.176 0.752
```

Data Compression Functions

10

This chapter describes the Intel® IPP functions for data compression that support a number of different compression methods: Huffman and variable-length coding, dictionary-based coding methods (including support of ZLIB compression), and methods based on Burrows-Wheeler Transform.

Application Notes

- The functions in this domain can be divided into two types: the functions that actually compress data, and transformation functions. The latters do not compress data but only modify them and prepare for further compression. The examples of such transformation are the *Burrows-Weller Transform*, or *MoveToFront* algorithm. To do data compression efficient, you should develop the proper consequence of functions of different type that will transform data and then compress them.
- Compression ratio depends on the statistics of input data. For some types of input data no compression could be achieved at all.
- The size of memory required for the output of data compression functions typically is not obvious. As a rule encoding functions uses less memory for output than the size of the input buffer, on the contrary decoding functions use more memory for output than the size of the input buffer. You should account these issues and allocate the proper quantity of output memory using the techniques provided by functions in this domain. For example, you can use a double-pointer technique for automatic shifting the user submitted pointer. For some other functions it is possible to compute the upper limit of the size of the required output buffer.

Dictionary-Based Compression Functions

This section describes the Intel IPP functions that use different dictionary-based compression methods.

LZSS Compression Functions

These functions implement the LZSS (Lempel-Ziv-Storer-Szymanski) compression algorithm [Storer82]. The functions perform LZSS coding with a vocabulary size of 32KB and 256-byte maximum match string length.

EncodeLZSSInit

Initializes the LZSS encoder state structure.

Syntax

```
IppStatus ippsEncodeLZSSInit_8u (IppLZSSState_8u* pLZSSState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pLZSSState	Pointer to the LZSS encoder state structure.

Description

This function initializes the LZSS state structure *pLZSSState* in the external buffer. Its size must be computed previously by calling the function ippsLZSSGetSize.

The LZSS encoder state structure is required for the encoder functions ippsEncodeLZSS and ippsEncodeLZSSFlush.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pLZSSState* is NULL.

LZSSGetSize

Computes the size of the LZSS state structure.

Syntax

```
IppStatus ippsLZSSGetSize_8u (int* pLZSSStateSize);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pLZSSStateSize	Pointer to the size of the LZSS state structure.

Description

This function computes the size in bytes of the LZSS state structure for encoding and decoding and stores it to an integer pointed to by *pLZSSStateSize*. The function must be called prior to the function ippsEncodeLZSSInit or ippsDecodeLZSSInit.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pLZSSStateSize* is NULL.

EncodeLZSS

Performs LZSS encoding.

Syntax

```
IppStatus ippsEncodeLZSS_8u (Ipp8u** ppSrc, int* pSrcLen, Ipp8u** ppDst, int* pDstLen,
IppLZSSState_8u* pLZSSState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the number of elements in the source buffer; it is updated after encoding.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer; it is updated and returns the length of the destination buffer after encoding.
pLZSSState	Pointer to the LZSS encoding state structure.

Description

This function performs LZSS encoding of data in the source buffer *ppSrc* of length *pSrcLen* and stores the result in the destination buffer *pDst* of length *pDstLen*. The LZSS encoder state structure *pLZSSState* must be initialized by ippsEncodeLZSSInit beforehand.

After encoding the function returns the pointers to source and destination buffers shifted by the number of successfully read and encoded bytes, respectively. The function updates *pSrcLen* and *pDstLen* so they return the actual number of elements in the source and destination buffers respectively.

Code example shows how to use the function ippsEncodeLZSS_8u and supporting functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *srcLen* is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

EncodeLZSSFlush

Encodes the last few bits in the bitstream and aligns the output data on the byte boundary.

Syntax

```
IppStatus ippsEncodeLZSSFlush_8u (Ipp8u** ppDst, int* pDstLen, IppLZSSState_8u*
pLZSSState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of destination buffer.
pLZSSState	Pointer to the LZSS encoder state structure.

Description

This function encodes the last few bits (remainder) in the bitstream, writes them to *ppDst*, and aligns the output data on a byte boundary.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if *pDstLen* is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

Example

To better understand usage of this function, refer to the following example in the examples archive available for download from https://software.intel.com/en-us/ipp-manual-examples :

EncodeLZSSFlush.c

DecodeLZSSInit

Initializes the LZSS decoder state structure.

Syntax

```
IppStatus ippsDecodeLZSSInit_8u (IppLZSSState_8u* pLZSSState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pLZSSState	Pointer to the LZSS decoder state structure.

Description

This function initializes the LZSS decoder state structure in the external buffer, the size of which must be computed previously by calling the function ippsLZSSGetSize.

The LZSS decoder state structure is required for the function ippsDecodeLZSS.

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error if the pointer *pLZSSState* is NULL.

DecodeLZSS

Performs LZSS decoding.

Syntax

```
IppStatus ippsDecodeLZSS_8u (Ipp8u** ppSrc, int* pSrcLen, Ipp8u** ppDst, int* pDstLen,
IppLZSSState_8u* pLZSSState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pLZSSState	Pointer to the LZSS decoding state structure.

Description

This function performs LZSS decoding of the *pSrcLen* elements of the *ppSrc* source buffer and stores the result in the *pDst* destination vector. The length of the destination vector is stored in *pDstLen*. The LZSS decoder state structure *pLZSSState* must be initialized by ippsDecodeLZSSInit beforehand.

After decoding the function returns the pointers to source and destination buffers shifted by the number of successfully read and decoded bytes respectively. The function updates *pSrcLen* so it is equal to the actual number of elements in the source buffer.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *pSrcLen* or *pDstLen* is negative.
ippStsDstSizeLessExpected	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

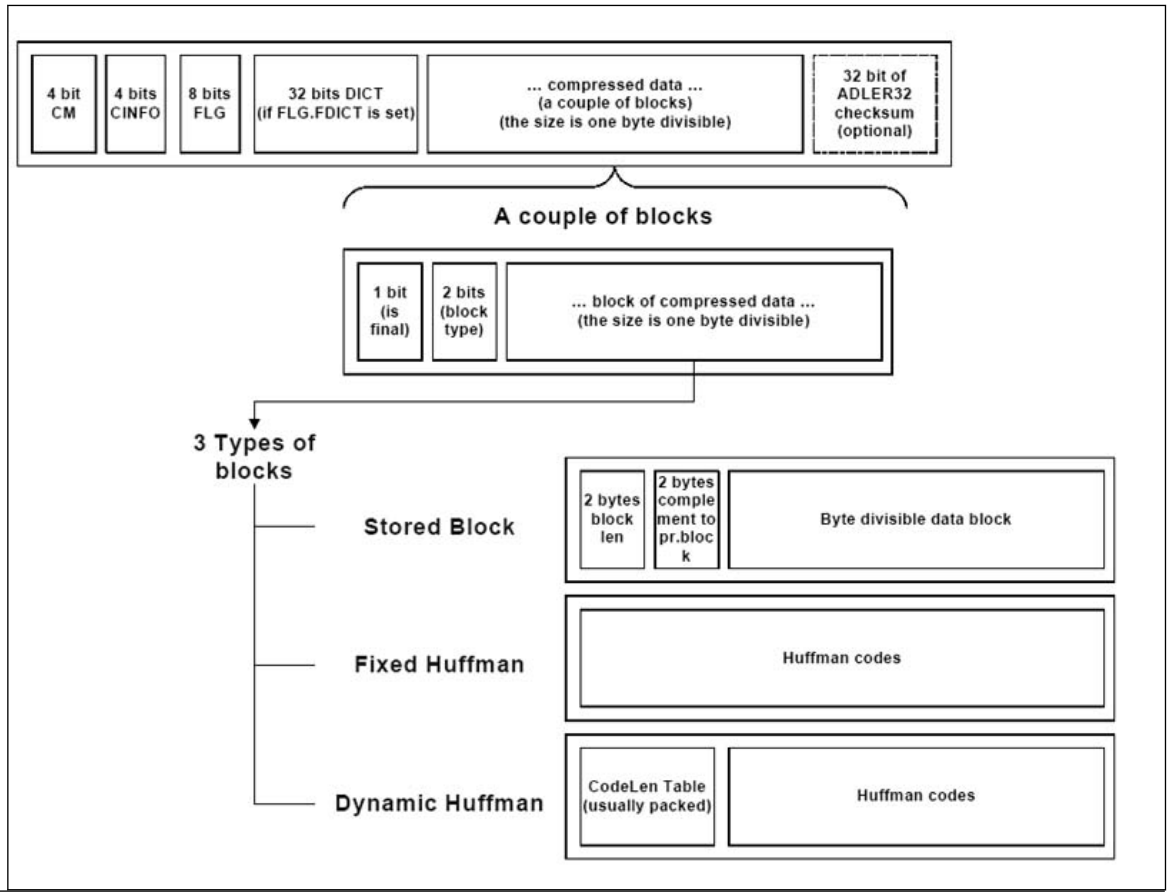
ZLIB Coding Functions

This section describes Intel IPP data compression functions that implement compression methods and data formats defined by the following specifications: [RFC1950], [RFC1951], and [RFC1952]. These formats are also known as ZLIB, DEFLATE, and GZIP, respectively.

A basic algorithm for these data compression methods is based on the Lempel-Ziv (LZ77) [Ziv77] dictionary-based compression.

The structure of ZLIB data is schematically shown in Figure .

ZLIB Data Structure



The full version of the `zlib` library is included with the product at `<ipp directory>/interfaces/data-compression/ipp_zlib`.

Special Parameters

The ZLIB coding functions have several special parameters.

The *`comprLevel`* parameter specifies the level of compression rate and compression ratio. The table below lists the possible values of the *`comprLevel`* parameter and their meanings.

Parameter *comprLevel* for ZLIB Functions

Value	Descriptions
`IppLZ77FastCompr`	Fast compression, maximum compression rate, and below average compression ratio
`IppLZ77AverageCompr`	Average compression rate, average compression ratio
`IppLZ77BestCompr`	Slow compression, maximum compression ratio

The *`checksum`* parameter specifies what algorithm is used to compute checksum for input data. The table below lists the possible values of the *`checksum`* parameter and their meanings.

Parameter *checksum* for ZLIB Functions

Value	Description
`IppLZ77NoChcksm`	Checksum is not calculated.
`IppLZ77Adler32`	Checksum is calculated using Adler32 algorithm.
`IppLZ77CRC32`	Checksum is calculated using the CRC32 algorithm.

The *flush* parameter specifies the encoding mode for data block encoding. The table below lists the possible values of the *flush* parameter and their meanings.

Parameter *flush* for ZLIB Functions

Value	Descriptions
IppLZ77NoFlush	The end of the block is aligned to a byte boundary.
IppLZ77SyncFlush	The end of the block is aligned to a byte boundary, and 4-byte marker is written to pDst.
IppLZ77FullFlush	The end of the block is aligned to a byte boundary, 4-byte marker is written to pDst, sliding dictionary is zeroed.
IppLZ77FinishFlush	The end of the block is aligned to a byte boundary and the function returns the ippStsStreamEnd status.

The *deflateStatus* parameter specifies the encoding status to ensure the compatibility with the RFC1951 specification. This parameter is used by Intel IPP ZLIB encoding functions. The table below lists the possible values of the *deflateStatus* parameter and their meanings.

Parameter *deflateStatus* for ZLIB Encoding Functions

Value	Descriptions
IppLZ77StatusInit	Specified the deflate implementation of encoding functions, must be used before stream encoding.
IppLZ77StatusLZ77Process	Call the deflate implementation of encoding function.
IppLZ77StatusHuffProcess	Call the deflate implementation of the encoding function with the fixed Huffman codes.
IppLZ77StatusFinal	Specified the last block in the stream.

The *inflateStatus* parameter specifies the decoding status to ensure the compatibility with the RFC1951 specification. This parameter is used by Intel IPP ZLIB decoding functions. The table below lists the possible values of the *inflateStatus* parameter and their meanings.

Parameter *inflateStatus* for ZLIB Decoding Functions

Value	Descriptions
IppLZ77inflateStatusInit	Specified the deflate implementation of encoding functions, must be used before stream encoding.
IppLZ77InflateStatusHuffProcess	Call the deflate implementation of the encoding function with the fixed Huffman codes.
IppLZ77InflateStatusLZ77Process	Call the deflate implementation of encoding function.
IppLZ77InflateStatusFinal	Specified the last block in the stream.

Adler32

Computes the Adler32 checksum for the source data buffer.

Syntax

```
IppStatus ippsAdler32_8u (const Ipp8u* pSrc, int srcLen, Ipp32u* pAdler32);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source data buffer.
srcLen	Number of elements in the source data buffer.

pAdler32	Pointer to the checksum value.

Description

This function computes the checksum for *srcLen* elements of the source data buffer *pSrc* and stores it in the *pAdler32*. The checksum is computed using the Adler32 algorithm that is a modified version of the Fletcher algorithm [Flet82], [ITU224], [RFC1950].

You can use this function to compute the accumulated value of the checksum for multiple buffers in the data stream by specifying as an input parameter the checksum value obtained in the preceding function call.

Return Values

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

CRC32, CRC32C

Computes the CRC32 checksum for the source data buffer.

Syntax

```
IppStatus ippsCRC32_8u (const Ipp8u* pSrc, int srcLen, Ipp32u* pCRC32);
IppStatus ippsCRC32C_8u(const Ipp8u* pSrc, Ipp32u srcLen, Ipp32u* pCRC32C);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source data buffer.
srcLen	Number of elements in the source data buffer.
pCRC32, *pCRC32C*	Pointer to the checksum value.

Description

These functions compute the checksum for *srcLen* elements of the source data buffer *pSrc* using different algorithms and stores it in the *pCRC32* or *pCRC32C* respectively.

The function ippsCRC32 uses algorithm described in [Griff87], [Nel92], the function ippsCRC32C uses algorithm described in [Cast93].

These functions can be used to compute the accumulated value of the checksum for multiple buffers in the data stream by specifying as an input parameter the checksum value obtained in the preceding function call.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pSrc* pointer is NULL.

`ippStsSizeErr`	Indicates an error if the length of the source vector is less than or equal to 0.

Example

The example below shows how to use the function `ippsCRC32C_8u`.

```
#include <iostream>
#include <iomanip>
#include "ipp.h"

using namespace std;
void crc32c_core( Ipp8u* src, Ipp32u src_len )
{
    Ipp32u crc32c = ~(Ipp32u)0;
    ippsCRC32C_8u( src, src_len, &crc32c );
    ippsSwapBytes_32u_I( &crc32c, 1 );
    cout << "0x" << setbase(16) << ~crc32c << endl;
}
int main()
{
    Ipp8u buff[48] = { 0x01, 0xc0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x14, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00,
                       0x00, 0x00, 0x00, 0x14, 0x00, 0x00, 0x00, 0x18,
                       0x28, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
    cout << "An iSCSI - SCSI Read (10) Command PDU: ";
    crc32c_core( buff, 48 );

    cout << "32 bytes of zeroes: ";
    for( int i = 0; i < 32; i++ ) buff[i] = 0;
    crc32c_core( buff, 32 );

    cout << "32 bytes of ones: ";
    for( int i = 0; i < 32; i++ ) buff[i] = 0xff;
    crc32c_core( buff, 32 );

    cout << "32 bytes of incrementing 00..1f: ";
    for( int i = 0; i < 32; i++ ) buff[i] = i;
    crc32c_core( buff, 32 );

    cout << "32 bytes of decrementing 1f..00: ";
    for( int i = 0; i < 32; i++ ) buff[i] = 31 - i;
    crc32c_core( buff, 32 );

    return 0;
}
```

```
of ones: 0x43aba862 32 bytes of incrementing 00..1f: 0x4e79dd46 32 bytes of decrementing 1f..00:
0x5cdb3f11
```

DeflateLZ77

Performs LZ77 encoding according to the specified compression level.

Syntax

```
IppStatus ippsDeflateLZ77_8u(const Ipp8u** ppSrc, Ipp32u* pSrcLen, Ipp32u* pSrcIdx,
const Ipp8u* pWindow, Ipp32u winSize, Ipp32s* pHashHead, Ipp32s* pHashPrev, Ipp32u
hashSize, IppDeflateFreqTable pLitFreqTable[286], IppDeflateFreqTable
pDistFreqTable[30], Ipp8u* pLitDst, Ipp16u* pDistDst, Ipp32u* pDstLen, int comprLevel,
IppLZ77Flush flush);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the length of the source vector.
pSrcIdx	Pointer to the index of the current position in the source vector.
pWindow	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
winSize	Size of the sliding window and the *pHashPrev* table.
pHashHead	Pointer to the table containing heads of the hash chains.
pHashPrev	Pointer to the table contating indexes to the previous strings with the same hash key.
hashSize	Size of the *pHashHead* table.
pLitFreqTable	Pointer to the literals/lengths frequency table.
pDistFreqTable	Pointer to the distances frequency table.
pLitDst	Pointer to the destination vector containing literals/lengths.
pDistDst	Pointer to the destination vector containing distances.
pDstLen	Pointer to the length of the destination vectors.
comprLevel	Compression level in range [0..9] in accordance with ZLIB.
flush	Specifies the encoding mode for data blocks (see flush parameter).

Description

This function performs LZ77 encoding of source data *ppSrc* according to the compression level *comprLevel*, which is similar to the ZLIB compression level.

To correctly process the first bytes of the source vector, initialize the *pHashHead* table with the *winSize* value.

The *pSrcIdx* parameter returns the index of the current position in the source vector, and is used to establish a correlation between the current position in the source vector and indexes in hash tables. After processing each 2GB of source data, the index and hash tables must be normalized (instead of 64K of source data in ZLIB).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.

See Also

Special Parameters

DeflateLZ77Fast

Performs LZ77 encoding according to the fast algorithm and parameters of a match.

Syntax

```
IppStatus ippsDeflateLZ77Fast_8u(const Ipp8u** ppSrc, Ipp32u* pSrcLen, Ipp32u* pSrcIdx,
const Ipp8u* pWindow, Ipp32u winSize, Ipp32s* pHashHead, Ipp32s* pHashPrev, Ipp32u
hashSize, IppDeflateFreqTable pLitFreqTable[286], IppDeflateFreqTable
pDistFreqTable[30], Ipp8u* pLitDst, Ipp16u* pDistDst, Ipp32u* pDstLen, int pVecMatch,
IppLZ77Flush flush);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the length of the source vector.
pSrcIdx	Pointer to the index of the current position in the source vector.
pWindow	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
winSize	Size of the sliding window and the *pHashPrev* table.
pHashHead	Pointer to the table containing heads of the hash chains.
pHashPrev	Pointer to the table containing indexes to the previous strings with the same hash key.
hashSize	Size of the *pHashHead* table.
pLitFreqTable	Pointer to the literals/lengths frequency table.
pDistFreqTable	Pointer to the distances frequency table.
pLitDst	Pointer to the destination vector containing literals/lengths.
pDistDst	Pointer to the destination vector containing distances.

pDstLen	Pointer to the length of the destination vectors.
pVecMatch	Pointer to the vector containing the following parameters of a match: `max_chain_length`, `good_match`, `nice_match`, `max_lazy_match` (for more information, see [ZLIB]).
flush	Specifies the encoding mode for data blocks (see flush parameter).

Description

This function performs LZ77 encoding of the *ppSrc* data according to the fast algorithm and parameters of a match.

To correctly process the first bytes of the source vector, initialize the *pHashHead* table with the *winSize* value.

The *pSrcIdx* parameter returns the index of the current position in the source vector, and is used to establish a correlation between the current position in the source vector and indexes in hash tables. After processing each 2GB of source data, the index and hash tables must be normalized (instead of 64K of source data in ZLIB).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error when *winSize* is less than 256, or more than 32768; or *hashSize* is less than 256, or more than 65536.
`ippStsBadArgErr`	Indicates an error when `good_match`, `nice_match`, or `max_lazy_match` is less than 4.

See Also

Special Parameters

DeflateLZ77Fastest

Performs LZ77 encoding according to the fastest algorithm.

Syntax

```
IppStatus ippsDeflateLZ77Fastest_8u(const Ipp8u** ppSrc, Ipp32u* pSrcLen, Ipp32u*
pSrcIdx, const Ipp8u* pWindow, Ipp32u winSize, Ipp32s* pHashHead, Ipp32u hashSize,
Ipp16u* pCode, Ipp32u* pCodeLenBits, Ipp8u* pDst, Ipp32u pDstLen, Ipp32u* pDstIdx,
IppDeflateHuffCode pLitHuffCodes[286], IppDeflateHuffCode pDistHuffCode[30],
IppLZ77Flush flush);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

ppSrc	Double pointer to the source vector.

pSrcLen	Pointer to the length of the source vector.
pSrcIdx	Pointer to the index of the current position in the source vector.
pWindow	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
winSize	Size of the sliding window and the *pHashPrev* table.
pHashHead	Pointer to the table containing heads of the hash chains.
hashSize	Size of the *pHashHead* table.
pCode	Pointer to the bit buffer.
pCodeLenBits	Pointer to the number of valid bits in the bit buffer.
pDst	Pointer to the destination vector.
pDstIdx	Pointer to the index in the destination vector.
pLitHuffCodes	Pointer to the literals/lengths Huffman codes.
flush	Specifies the encoding mode for data blocks (see flush parameter).

Description

This function performs LZ77 encoding of the *ppSrc* data according to the fastest algorithm.

To correctly process the first bytes of the source vector, initialize the *pHashHead* table with the *winSize* value.

The *pSrcIdx* parameter returns the index of the current position in the source vector, and is used to establish a correlation between the current position in the source vector and indexes in hash tables. After processing each 2GB of source data, the index and hash tables must be normalized (instead of 64K of source data in ZLIB).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when at least one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *winSize* is less than 256, or more than 32768; or *hashSize* is less than 256, or more than 65536.

See Also

Special Parameters

DeflateLZ77FastestGenHeader

Computes a header of the provided Huffman tables description.

Syntax

```
IppStatus ippsDeflateLZ77FastestGenHeader_8u(const IppDeflateHuffCode
pLitCodeTable[286], const IppDeflateHuffCode pDistCodeTable[30], Ipp8u* pDstHeader,
int* pDstLen, int* pDstBits);
```

Include Files

ippdc.h

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pLitCodeTable	Pointer to literals/lengths in a Huffman code.
pDistCodeTable	Pointer to distances in a Huffman code.
pDstHeader	Pointer to the header.
pDstLen	Pointer to the header length.
pDstBits	Pointer to the header length, in bits.

Description

This function gets a header of the provided Huffman tables description.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when one of the specified pointers is `NULL`.

DeflateLZ77FastestGenHuffTable

Builds Huffman tables according to statistical data collections.

Syntax

```
IppStatus ippsDeflateLZ77FastestGenHuffTable_8u(const int pLitStat[286], const int
pDistStat[30], IppDeflateHuffCode pLitCodeTable[286], IppDeflateHuffCode
pDistCodeTable[30]);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pLitStat	Pointer to data collection for literals and match lengths.
pDistStat	Pointer to data collection for distances.
pLitCodeTable	Pointer to the literals/lengths Huffman codes.
pDistCodeTable	Pointer to the distances Huffman codes.

Description

This function builds Huffman tables for literals/lengths according to the provided statistical data collection.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.

DeflateLZ77FastestGetStat

Performs statistical data collection required to build user's Huffman table.

Syntax

```
IppStatus ippsDeflateLZ77FastestGetStat_8u(const Ipp8u** ppSrc, Ipp32u* pSrcLen,
Ipp32u* pSrcIdx, const Ipp8u* pWindow, Ipp32u winSize, Ipp32s* pHashHead, Ipp32u
hashSize, int pLitStat[286], int pDistStat[30], IppLZ77Flush flush);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the length of the source vector.
pSrcIdx	Pointer to the index of the current position in the source vector.
pWindow	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
winSize	Size of the sliding window and the pHashPrev table.
pHashHead	Pointer to the table containing heads of the hash chains.
hashSize	Size of the *pHashHead* table.
pLitStat	Pointer to data collection for literals and match lengths.
pDistStat	Pointer to data collection for distances.
flush	Specifies the encoding mode for data blocks.

Description

This function performs collection of statistical data. This data is needed for functions building user's Huffman table ippsDeflateLZ77FastestGenHuffTable and functions computing a header of Huffman tables description ippsDeflateLZ77FastestGenHeader.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *winSize* is less than 256 or more than 32768, or if *hashSize* is less than 256 or more than 65536.

DeflateLZ77FastestPrecompHeader

Performs LZ77 encoding using the fastest algorithm with prebuilding of customer Huffman tables and prediction header.

Syntax

```
IppStatus ippsDeflateLZ77FastestPrecompHeader_8u(const Ipp8u** ppSrc, Ipp32u* pSrcLen,
Ipp32u* pSrcIdx, const Ipp8u* pWindow, Ipp32u winSize, Ipp32s* pHashHead, Ipp32u
hashSize, Ipp16u* pCode, Ipp32u* pCodeLenBits, Ipp8u* pDst, Ipp32u pDstLen, Ipp32u*
pDstIdx, IppDeflateHuffCode pLitHuffCodes[288], IppDeflateHuffCode pDistHuffCode[30],
const Ipp8u* pHeaderCodeLens, int numBitsHeader , IppLZ77Flush flush);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the length of the source vector.
pSrcIdx	Pointer to the index of the current position in the source vector.
pWindow	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
winSize	Size of the sliding window and the `pHashPrev` table.
pHashHead	Pointer to the table containing heads of the hash chains.
hashSize	Size of the *pHashHead* table.
pCode	Pointer to the bit buffer.
pCodeLenBits	Pointer to the number of valid bits in the bit buffer.
pDst	Pointer to the destination vector.
pDstIdx	Pointer to the index in the destination vector.
pLitHuffCodes	Pointer to the literals/lengths Huffman codes.
pDistHuffCodes	Pointer to the distances Huffman codes.
pHeaderCodeLens	Pointer to the prediction header with description of Huffman tables.
numBitsHeader	Length of the prediction header, in bits.
flush	Specifies the encoding mode for data blocks.

Description

This function performs LZ77 encoding of source data *ppSrc* using the fastest algorithm.

To correctly process the first bytes of the source vector, initialize the *pHashHead* table with -*winSize* value.

The *pSrcIdx* parameter returns the index of the current position in the source vector and is used to correlate the current position in the source vector and indexes in the hash tables. After processing each 2GB of source data, this index and hash tables must be normalized (instead of 64K of source data in ZLIB).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error when one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error when *winSize* is less than 256 or more than 32768, or if *hashSize* is less than 256 or more than 65536, or if **pDstIdx* is more than or equal to *dstLen*.

DeflateLZ77Slow

Performs LZ77 encoding according to the slow algorithm and parameters of a match.

Syntax

```
IppStatus ippsDeflateLZ77Slow_8u(const Ipp8u** ppSrc, Ipp32u* pSrcLen, Ipp32u* pSrcIdx,
const Ipp8u* pWindow, Ipp32u winSize, Ipp32s* pHashHead, Ipp32s* pHashPrev, Ipp32u
hashSize, IppDeflateFreqTable pLitFreqTable[286], IppDeflateFreqTable
pDistFreqTable[30], Ipp8u* pLitDst, Ipp16u* pDistDst, Ipp32u* pDstLen, int pVecMatch,
IppLZ77Flush flush);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the length of the source vector.
pSrcIdx	Pointer to the index of the current position in the source vector.
pWindow	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
winSize	Size of the sliding window and the *pHashPrev* table.
pHashHead	Pointer to the table containing heads of the hash chains.
pHashPrev	Pointer to the table contating indexes to the previous strings with the same hash key.
hashSize	Size of the *pHashHead* table.
pLitFreqTable	Pointer to the literals/lengths frequency table.
pDistFreqTable	Pointer to the distances frequency table.
pLitDst	Pointer to the destination vector containing literals/lengths.
pDistDst	Pointer to the destination vector containing distances.

pDstLen	Pointer to the length of the destination vectors.
pVecMatch	Pointer to the vector containing the following parameters of a match: `max_chain_length`, `good_match`, `nice_match`, `max_lazy_match` (for more information, see [ZLIB]).
flush	Specifies the encoding mode for data blocks (see flush parameter).

Description

This function performs LZ77 encoding of source data *ppSrc* according to the slow algorithm and match parameters.

To correctly process the first bytes of the source vector, initialize the *pHashHead* table with the *winSize* value.

The *pSrcIdx* parameter returns the index of the current position in the source vector, and is used to establish a correlation between the current position in the source vector and indexes in hash tables. After processing each 2GB of source data, the index and hash tables must be normalized (instead of 64K of source data in ZLIB).

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error when at least one of the specified pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if winSize is less than 256 or more than 32768 or hashSize is less than 256 or more than 65536.
`ippStsBadArgErr`	Indicates an error when `good_match`, `nice_match`, or `max_lazy_match` is less than 4, or `max_chain_length` is less than 1.

See Also

Special Parameters

DeflateDictionarySet

Presets the user's dictionary for LZ77 encoding.

Syntax

```
IppStatus ippsDeflateDictionarySet_8u(const Ipp8u* pDictSrc, Ipp32u dictLen, Ipp32s*
pHashHeadDst, Ipp32u hashSize, Ipp32s* pHashPrevDst, Ipp8u* pWindowDst, Ipp32u winSize,
int comprLevel);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pDictSrc	Pointer to the user's dictionary.
dictLen	Length of the user's dictionary.

pHashHeadDst	Pointer to the table containing heads of the hash chains.
pHashPrevDst	Pointer to the table containing indexes to the previous strings with the same hash key.
hashSize	Size of the *pHashHeadDst* table.
pWindowDst	Pointer to the sliding window that is used as the dictionary for LZ77 encoding.
winSize	Size of the sliding window and the elements of the *pHashPrevDst* table.
comprLevel	Compression level in range [0..9] in accordance with ZLIB.

Description

This function presets the user's dictionary for LZ77 encoding.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

DeflateUpdate Hash

Performs LZ77 encoding according to the specified compression level.

Syntax

```
IppStatus ippsDeflateUpdateHash_8u(const Ipp8u* pSrc, Ipp32u srcIdx, Ipp32u srcLen,
Ipp32s* pHashHeadDst, Ipp32u hashSize, Ipp32s* pHashPrevDst, Ipp32u winSize, int
comprLevel);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source vector.
srcIdx	Index of the current position in the source vector.
srcLen	Length of the source vector.
pHashHeadDst	Pointer to the table containing heads of the hash chains.
hashSize	Size of the *pHashHeadDst* table.
pHashPrevDst	Pointer to the table contating indexes to the previous strings with the same hash key.
winSize	Size of the sliding window and the *pHashPrevDst* table.

comprLevel	Compression level in range [0..9] in accordance with ZLIB.

Description

This function updates hash tables according to the source context.

The function parameter *srcIdx* - index of the current position in the source vector - is used to correlate the current position in the source vector and indexes in the hash tables. After processing each 2GB of source data, this index and hash tables must be normalized (instead of 64K of source data in ZLIB).

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.
ippStsSizeErr	Indicates an error if *winSize* is less than or equal to 256, or greater than 32768; or if *hashSize* is less than or equal to 256, or greater than 65536.

DeflateHuff

Performs Huffman encoding .

Syntax

```
IppStatus ippsDeflateHuff_8u(const Ipp8u* pLitSrc, const Ipp16u* pDistSrc, Ipp32u
srcLen, Ipp16u* pCode, Ipp32u* pCodeLenBits, IppDeflateHuffCode pLitHuffCodes[286],
IppDeflateHuffCode pDistHuffCodes[30], Ipp8u* pDst, Ipp32u* pDstIdx);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pLitSrc	Pointer to the literals/lengths source vector.
pDistSrc	Pointer to the distances source vector.
srcLen	Length of the source vectors.
pCode	Pointer to the bit buffer.
pCodeLenBits	Pointer to the number of valid bits in the bit buffer.
pLitHuffCodes	Pointer to the literals/lengths Huffman codes.
pDistHuffCodes	Pointer to the distances Huffman codes.
pDst	Pointer to the destination vector.
pDstIdx	Pointer to the index in the destination vector.

Description

This function performs Huffman encoding of source data.

The function parameter *pDstIdx* returns the index of the current position in the destination vector: zlib uses the intermediate buffer for the Huffman encoding and we need to know the indexes of the first (input parameter) and the last (output parameter) symbols, which are written by the function.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

InflateBuildHuffTable

Builds the Huffman code table for compressed block in the "deflate" format.

Syntax

```
IppStatus ippsInflateBuildHuffTable(const Ipp16u* pCodeLens, unsigned int nLitCodeLens,
unsigned int nDistCodeLens, IppInflateState* pIppInflateState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pCodeLens	Pointer to the common array with lengths of the Huffman codes for literals/lengths and distances.
nLitCodeLens	Number of lengths of the Huffman codes for literals/lengths.
nDistCodeLens	Number of lengths of the Huffman codes for distances.
pIppInflateState	Pointer to the structure with the parameters of decoding.

Description

This function builds tables of Hufman codes for literals/lengths and distances to decode a block compressed with use of the dynamic Huffman codes in accordance with the "deflate" format [RFC1951].

The structure *IppInflateState* contains the following fields:

pWindow	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
winSize	Size of the sliding window in the range [256, 32768].
tableType	Type of the Huffman code tables. For dynamic Huffman code it is greater than 0, for fixed Huffman codes is equal to 0.
tableBufferSize	Size of the buffer containing the tables. Its value is 8192 – sizeof(IppInflateState). (8192 =ENOUGH*sizeof(code); ENOUGH is defined in ZLIB and is equal to 2048, sizeof(code)=4.)

Return Values

ippStsNoErr	Indicates no error.

ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if *nLitCodeLens* is greater than 286 or *nDistCodeLens* is greater than 30.
ippStsSrcDataErr	Indicates an error if a not valid literal/length and distance set occurs in the common lengths array.

Inflate

Decodes data in the "deflate" format.

Syntax

```
IppStatus ippsInflate_8u(Ipp8u** ppSrc, unsigned int* pSrcLen, Ipp32u* pCode, unsigned
int* pCodeLenBits, unsigned int winIdx, Ipp8u** ppDst, unsigned int* pDstLen, unsigned
int dstIdx, IppInflateMode* pMode, IppInflateState* pIppInflateState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the length of source vector.
pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
winIdx	Index of the start position of the sliding window.
ppDst	Double pointer to the destination vector.
pDstLen	Pointer to the length of destination vector.
dstIdx	Index of the current position in the destination vector.
pMode	Pointer to the current decode mode. Possible values are: ippTYPE - block decoding is completed; ippLEN - decoding from the beginning of the sequence; ippLENEXT - extra bits are required to decode the sequence.
pIppInflateState	Pointer to the structure that contains parameters of decoding.

Description

This function decodes the data encoded in the "deflate" format [RFC1951] in accordance with the parameters set in the structure *pIppInflateState*. If the data is compressed using dynamic Huffman codes, the Huffman code tables must be built by the function ippsInflateBuildHuffTable beforehand. If the data is compressed using the fixed Huffman codes, the fileld *tableType* in the *pIppInflateState* must be set to 0, and code tables are not required to be built at all.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if *pCodeLenBits* is greater than 32, or if *winIdx* is greater than *pIppInflateState->winSize*, or if *dstIdx* is greater than *pDstLen*.
ippStsSrcDataErr	Indicates an error if a not valid literal/length and distance set occurs during decoding.

LZO Compression Functions

This section describes Intel IPP data compression functions, that implement the LZO (Lempel-Ziv-Oberhumer) compressed data format. This format and algorithm use 64KB compression dictionary and do not require additional memory for decompression. (See original code of the LZO library at http://www.oberhumer.com.)

Special Parameters

The LZO coding initialization functions have a special parameter *method*. This parameter specifies level of parallelization and generic LZO compatibility to be used in the LZO encoding. The table below lists possible values of the *method* parameter and their meanings.

Parameter *method* for the LZO Compression Functions

Value	Descriptions
IppLZO1XST	The compression and decompression are performed sequentially in a single-thread mode with full binary compatibility with generic LZO libraries and applications
IppLZO1XMT	The compression and decompression are performed in parallel (multi-threaded mode), it is more fast, but not compatible with the generic LZO.

EncodeLZOGetSize

Calculates the size of LZO encoding structure.

Syntax

```
IppStatus ippsEncodeLZOGetSize(IppLZOMethod method, Ipp32u maxInputLen, Ipp32u* pSize);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

method	Specifies required LZO compression method, possible values are listed in Table "*method* Parameter").

maxInputLen	Specifies maximum length of input data buffer during compression operations. Not required for `IppLZO1XST` compression method.
pSize	Pointer to the variable, receiving the size of LZO encoding structure.

Description

This function calculates the size of the memory buffer that must be allocated for the LZO encoding structure.

For the single-thread compression (*method* = `IppLZO1XST`) the size of the structure is fixed, and the value of the *maxInputLen* parameter is ignored, for example, it can be set to 0.

For the multi-threaded compression (*method* = `IppLZO1XMT`) *maxInputLen* parameter is important and affects the size of the structure. If it is set to 0, then each compression operation starts with memory allocation for internal buffers and ends with memory freeing. This significantly decreases the performance of compression/decompression.

Code example shows how the Intel IPP functions for the LZO compression can be used.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the pointer *pSize* is `NULL`.
`ippStsBadArgErr`	Indicates an error if the parameter *method* has an illegal value.

EncodeLZOInit

Initializes LZO encoding structure.

Syntax

```
IppStatus ippsEncodeLZOInit_8u(IppLZOMethod method, Ipp32u maxInputLen, IppLZOState_8u*
pLZOState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

method	Specifies required LZO compression method, possible values are listed in Table "*method* Parameter").
maxInputLen	Specifies maximum length of input data buffer during compression operations. Not required for `IppLZO1XST` compression method.
pLZOState	Pointer to the LZO encoding structure.

Description

This function initializes the LZO encoding structure in the external buffer. Its size must be calculated by calling the function `ippsEncodeLZOGetSize` beforehand.

The parameter *method* must be the same for both functions.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pLZOState* is NULL.
ippStsBadArgErr I	ndicates an error if the parameter *method* has an illegal value.

EncodeLZO

Compresses input data, returns the length of the compressed data.

Syntax

```
IppStatus ippsEncodeLZO_8u (const Ipp8u* pSrc, Ipp32u srcLen, Ipp8u* pDst, Ipp32u*
pDstLen, IppLZOState_8u* pLZOState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source buffer.
srcLen	Length of the source buffer.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pLZOState	Pointer to the LZO state structure.

Description

This function performs compression of the source data *pSrc* according to the method specified in the LZO state structure *pLZOState*. It must be previously initialized by the function ippsEncodeLZOInit.

Compressed data are stored in the *pDst*, the pointer *pDstLen* points to the number of elements in this buffer.

Code example shows how the Intel IPP functions for the LZO compression can be used.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

DecodeLZO

Decompresses input data, returns the length of the decompressed data.

Syntax

```
IppStatus ippsDecodeLZO_8u (const Ipp8u* pSrc, Ipp32u srcLen, Ipp8u* pDst, Ipp32u*
pDstLen);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source data buffer.
srcLen	Number of elements in the source data buffer.
pDst	Pointer to the destination data buffer.
pDstLen	Pointer to the variable with the number of elements in the destination data buffer.

Description

The function decompresses the source (compressed) data according to the compressed data format. This function can decompress both single-thread and multi-threaded data. Note that the maximum performance can be obtained only in multi-threaded decompression of data compressed in the multi-threaded mode.

> **NOTE**
> Destination data buffer must have enough free space to hold uncompressed data. No output buffer check is performed and no error code is returned. In the case of doubts use safe version of this function DecodeLZOSafe .

Code example shows how the Intel IPP functions for the LZO compression can be used.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.

DecodeLZOSafe

Decompresses input data with constantly checking integrity of output.

Syntax

```
IppStatus ippsDecodeLZOSafe_8u(const Ipp8u* pSrc, Ipp32u srcLen, Ipp8u* pDst, Ipp32u*
pDstLen);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source data buffer.
srcLen	Number of elements in the source data buffer.
pDst	Pointer to the destination data buffer.
pDstLen	Pointer to the variable with the number of elements in the destination data buffer.

Description

This function is a version of the function `ippsDecodeLZO`. It decompresses the source (compressed) data according to the compressed data format. The function can decompress both single-thread and multi-threaded data. The maximum performance can be obtained only in multi-threaded decompression of data compressed in the multi-threaded mode. Additionally this function checks the integrity of the destination data buffer, that is checks the buffer boundary limits. This function works slower, it can be used in doubtful cases when the compressed data integrity is not guaranteed, for example, decoding data received via non-reliable communication lines.

Destination data buffer must have enough free space to hold uncompressed data. Prior to the function call the destination buffer size variable pointed to by *pDstLen* must be initialized with actual number of free bytes in the destination buffer.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is `NULL`.
`ippStsLzoBrokenStreamErr`	Indicates an error if compressed data is not valid - not an LZO compressed data.
`ippStsDstSizeLessExpected`	Destination buffer is too small to store decompressed data.

Example

The code example below shows how to use Intel IPP functions for the LZO compression.

```
/* Simple example of file  compression using IPP LZO functions */
#include <stdio.h>
#include "ippdc.h"
#include "ipps.h"

#define BUFSIZE 1024
void CompressFile(const char* pInFileName, const char* pOutFileName)
{
    FILE *pIn, *pOut;
    IppLZOState_8u *pLZOState;
    Ipp8u src[BUFSIZE];
    /* For uncompressible data the size of output will be bigger */
    Ipp8u dst[BUFSIZE + BUFSIZE/10];
    Ipp32u srcLen, dstLen, lzoSize;

    pIn = fopen(pInFileName, "rb");
    pOut = fopen(pOutFileName, "wb");
    ippsEncodeLZOGetSize(IppLZO1XST, BUFSIZE, &lzoSize);
    pLZOState = (IppLZOState_8u*)ippsMalloc_8u(lzoSize);
    ippsEncodeLZOInit_8u(IppLZO1XST, BUFSIZE, pLZOState);
    while ((srcLen = (Ipp32u)fread(src, 1, BUFSIZE, pIn)) > 0) {
        ippsEncodeLZO_8u(src, srcLen, dst, &dstLen, pLZOState);
        fwrite(&srcLen, 1, sizeof(srcLen), pOut);
        fwrite(&dstLen, 1, sizeof(dstLen), pOut);
        fwrite(dst, 1, dstLen, pOut);
    }
    fclose(pIn);
    fclose(pOut);
}
/* Example of using of DecodeLZO function to decompress the file */
void DecompressFile(const char* pInFileName, const char* pOutFileName)
{
    FILE *pIn, *pOut;
    size_t allocSizeSrc = 0;
    size_t allocSizeDst = 0;
    Ipp32u srcLen, dstLen;
    Ipp8u *pSrc, *pDst;

    pIn = fopen(pInFileName, "rb");
```

BWT-Based Compression Functions

This section describes the Intel IPP functions that support composed algorithms based on the Burrows-Wheeler transform (BWT).

Burrows-Wheeler Transform

Burrows-Wheeler Transform (BWT) does not compress data, but it simplifies the structure of input data and makes more effective further compression. One of the distinctive feature of this method is operation on the block of data (as a rule of size 512kB - 2 mB). The main idea of this method is block sorting which groups symbols with a similar context. Let us consider how BWT works on the input data block `abracadabra'. The first step is to create a matrix containing all its possible cyclic permutations. The first row is input string, the second is created by shifting it to the left by one symbol and so on:

```
abracadabra bracadabraa racadabraab acadabraabr cadabraabra adabraabrac dabraabraca abraabracad
braabracada raabracadab aabracadabr
```

Then all rows are sorted in accordance with the lexicographic order:

```
 0 aabracadabr  1 abraabracad  2 abracadabra  3 acadabraabr  4 adabraabrac  5 braabracada  6
bracadabraa  7 cadabraabra  8 dabraabraca  9 raabracadab 10 racadabraab
```

The last step is to write out the last column and the index of the input string: `rdarcaaaabb`, 2 - this is a result of the forward BWT transform.

Inverse BTW is performed as follows:

elements of the input string are numbered in ascending order

```
 0  r  1  d  2  a  3  r  4  c  5  a  6  a  7  a  8  a  9  b 10  b
```

and sorted in accordance with the lexicographic order:

```
 2  a  5  a  6  a  7  a  8  a  9  b 10  b  4  c  1  d  0  r  3  r
```

This index array is a vector of the inverse transform (`Inv`), the further reconstruction of the string is performed in the following manner:

```
src[] = "rdarcaaaabb";

Inv[] = {2,5,6,7,8,9,10,4,1,0,3};

index = 2; // index of the initial string is known from the forward BWT

for( i = 0; i < len; i++ ) {

  index = Inv[index];

  dst[i] = src[index];

}
```

BWTFwdGetSize

Computes the size of the external buffer for the forward BWT transform.

Syntax

```
IppStatus ippsBWTFwdGetSize_8u(int wndSize, int* pBWTFwdBuffSize);
```

Include Files

ippdc.h

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

wndSize	Window size for BWT transform.
pBWTFwdBuffSize	Pointer to the computed size of the additional buffer.

Description

This function computes the size of memory (in bytes) of the external buffer that is required by the function `ippsBWTFwd` for the forward BWT transform.

Code example shows how to use the function `ippsBWTFwdGetSize_8u`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pBWTFwdBuffSize* pointer is `NULL`.

BWTFwd

Performs the forward BWT transform.

Syntax

```
IppStatus ippsBWTFwd_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int* pIndex, Ipp8u*
pBWTFwdBuff);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the source and destination vectors.
pIndex	Pointer to the index of first position for the forward BWT transform.
pBWTFwdBuff	Pointer to the additional buffer.

Description

This function performs the forward BWT transform of *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTFwdBuff*. The size of this buffer must be computed by calling the function `ippsBWTFwdGetSize` beforehand.

Code example shows how to use the function `ippsBWTFwd_8u`.

Return Values

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

BWTFwdGetBufSize_SelectSort

Computes the size of the external buffer for the forward BWT transform.

Syntax

```
IppStatus ippsBWTFwdGetBufSize_SelectSort_8u(Ipp32u wndSize, Ipp32u* pBWTFwdBufSize,
IppBWTSortAlgorithmHint sortAlgorithmHint);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

wndSize	Window size for BWT transform.
pBWTFwdBufSize	Pointer to the computed size of the additional buffer.
sortAlgorithmHint	Specifies what sort algorithm is used, possible values: ippBWTItohTanakaLimSort; ippBWTItohTanakaUnlimSort; ippBWTSuffixSort; ippBWTAutoSort.

Description

This function computes the size of memory (in bytes) of the external buffer that is required by the function BWTFwd_SelectSort for the forward BWT transform.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pBuffSize* pointer is NULL.

BWTFwd_SelectSort

Performs the forward BWT transform with specified sort algorithm.

Syntax

```
IppStatus ippsBWTFwd_SelectSort_8u(const Ipp8u* pSrc, Ipp8u* pDst, Ipp32u len, Ipp32u*
index, Ipp8u* pBWTFwdBuf, IppBWTSortAlgorithmHint sortAlgorithmHint);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the source and destination vectors.
index	Index of the first position for the forward BWT transform.
pBWTFwdBuf	Pointer to the additional buffer.
sortAlgorithmHint	Specifies what sort algorithm is used, possible values: ippBWTItohTanakaLimSort; ippBWTItohTanakaUnlimSort; ippBWTSuffixSort; ippBWTAutoSort.

Description

This function performs the forward BWT transform of *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The parameter *sortAlgorithmHint* specifies the desired algorithm of sorting. The function uses the external buffer *pBuff*. The size of this buffer must be computed by calling the function BWTFwdGetBufSize_SelectSort beforehand.

Return Values

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

BWTInvGetSize

Computes the size of the external buffer for the inverse BWT transform.

Syntax

```
IppStatus ippsBWTInvGetSize_8u(int wndSize, int* pBWTInvBuffSize);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

wndSize	Window size for BWT transform.
pBWTInvBuffSize	Pointer to the computed size of the additional buffer.

Description

This function computes the size of memory (in bytes) of the external buffer that is required by the function ippsBWTInv for the inverse BWT transform.

Code example shows how to use the function ippsBWTInvGetSize_8u.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pBWTInvBuffSize* pointer is NULL.

BWTInv

Performs the inverse BWT transform.

Syntax

```
IppStatus ippsBWTInv_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int index, Ipp8u*
pBWTInvBuff);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source vector.
pDst	Pointer to the destination vector.
len	Number of elements in the source and destination vectors.
index	Index of first position for the inverse BWT transform.
pBWTInvBuff	Pointer to the additional buffer.

Description

This function performs the inverse BWT transform of *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTInvBuff*. The size of this buffer must be computed by calling the function ippsBWTInvGetSize beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the specified pointers is NULL.

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

Example

The code example below shows how to use the function ippsBWTInv_8u.

```
void func_BWT()
{
    int wndSize = 8;
    int pBWTFwdBuffSize;
    int pBWTInvBuffSize;

    Ipp8u pSrc[] = "baadeffg";
    int len = 8;
    int pIndex;

    Ipp8u* pDst = ippsMalloc_8u(len);
    Ipp8u* pDstInv = ippsMalloc_8u(len);

    ippsBWTFwdGetSize_8u(wndSize, &pBWTFwdBuffSize);
    Ipp8u* pBWTFwdBuff = ippsMalloc_8u(pBWTFwdBuffSize);
    ippsBWTFwd_8u(pSrc, pDst, len, &pIndex, pBWTFwdBuff);

    ippsBWTInvGetSize_8u( wndSize, &pBWTInvBuffSize);
    Ipp8u* pBWTInvBuff = ippsMalloc_8u(pBWTInvBuffSize);
    ippsBWTInv_8u(pDst, pDstInv, len, pIndex, pBWTInvBuff);
}
Result:
    pDst ->     "bagadeff"
    pDstInv ->  "baadeffg"
```

Move To Front Functions

This section describes the functions that performs Move To Front (MTF) data transform method. The basic idea is to represent the symbols of the source sequence as the current indexes of that symbols in the modified alphabet. This alphabet is a list where frequently used symbols are placed in the upper lines. When the given symbols occurs it is replaced by its index in the list, then this symbol is moved in the first position in the list, and all indexes are updated. For example, the sequence "baabbfffacczzdd" contains symbols that form the ordered 'alphabet'{'a', 'b', 'c', 'd', 'f', 'z'}. The function will operate in the following manner:

Move To Front Operation

source	destination	alphabet
		0, 1, 2, 3, 4, 5, 6
		'a', 'b', 'c', 'd', 'f', 'z'
b	1	'b', 'a', 'c', 'd', 'f', 'z'
a	1	'a', 'b', 'c', 'd', 'f', 'z'

source	destination	alphabet
a	0	'a', 'b', 'c', 'd', 'f', 'z'
b	1	'b', 'a', 'c', 'd', 'f', 'z'
f	4	'f', 'b', 'a', 'c', 'd', 'z'
f	0	'f', 'b', 'a', 'c', 'd', 'z'
f	0	'f', 'b', 'a', 'c', 'd', 'z'
a	2	'a', 'f', 'b', 'c', 'd', 'z'
c	3	'c', 'a', 'f', 'b','d', 'z'
c	0	'c', 'a', 'f', 'b', 'd', 'z'
z	5	'z', 'c', 'a', 'f', 'b', 'd'
z	0	'z', 'c', 'a', 'f', 'b', 'd'
d	5	'd', 'z', 'c', 'a', 'f', 'b'
d	0	'd', 'z', 'c', 'a', 'f', 'b'

Finally, the function returns the destination sequence: 11014002305050.

These transformed data can be used for the following effective compression. This method is often used after Burrows-Wheeler transform.

MTFInit

Initializes the MTF structure.

Syntax

```
IppStatus ippsMTFInit_8u(IppMTFState_8u* pMTFState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pMTFState	Pointer to the MTF structure.

Description

This function initializes the MTF structure that contains parameters for the MTF transform in the external buffer. This structure is used by the functions `ippsMTFFwd` and `ippsMTFInv`. The size of this buffer must be computed previously by calling the function `ippsMTFGetSize`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pMTFState* pointer is `NULL`.

MTFGetSize

Computes the size of the MTF structure.

Syntax

```
IppStatus ippsMTFGetSize_8u(int* pMTFStateSize);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pMTFStateSize	Pointer to the computed MTF structure size.

Description

This function computes the size of memory (in bytes) that is required for the MTF structure. This function must be called prior to the function `ippsMTFInit`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pMTFStateSize* pointer is NULL.

MTFFwd

Performs the forward MTF transform.

Syntax

```
IppStatus ippsMTFFwd_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, IppMTFState_8u*
pMTFState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source buffer.
pDst	Pointer to the destination buffer.
len	Number of elements in the source and destination buffers.
pMTFState	Pointer to the MTF structure.

Description

This function performs the forward MTF transform of *len* elements of the data in the source buffer *pSrc* and stores result in the buffer *pDst*. The parameters of the MTF transform are specified in the MTF structure *pMTFState* that must be initialized by `ippsMTFInit` beforehand.

Return Values

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

MTFInv

Performs the inverse MTF transform.

Syntax

```
IppStatus ippsMTFInv_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, IppMTFState_8u*
pMTFState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source buffer.
pDst	Pointer to the destination buffer.
len	Number of elements in the source and destination buffers.
pMTFState	Pointer to the MTF structure.

Description

This function performs the inverse MTF transform of *len* elements of data in the source buffer *pSrc* and stores result in the buffer *pDst*. The parameters of the MTF transform are specified in the MTF structure *pMTFState* that must be initialized by `ippsMTFInit` beforehand.

Return Values

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

bzip2 Coding Functions

This section describes different Intel IPP functions to perform bzip2 encoding and decoding.

EncodeRLEInit_BZ2

Initializes the bzip2-specific RLE structure.

Syntax

```
IppStatus ippsEncodeRLEInit_BZ2_8u(IppRLEState_BZ2* pRLEState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pRLEState	Pointer to the bzip2-specific RLE structure.

Description

This function initializes the bzip2-specific RLE structure that contains parameters for the RLE in the external buffer. This structure is used by the function `ippsEncodeRLE_BZ2`. The size of this buffer must be computed previously by calling the function `ippsRLEGetSize_BZ2`.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pRLEState* pointer is `NULL`.

RLEGetSize_BZ2

Compute the size of the state structure for the bzip2-specific RLE.

Syntax

```
IppStatus ippsRLEGetSize_BZ2_8u(int* pRLEStateSize);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pRLEStateSize	Pointer to the size of the state structure for bzip2-specific RLE.

Description

This function computes the size of memory (in bytes) of the internal state structure for the bzip2-specific RLE.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if *pRLEStateSize* pointer is `NULL`.

EncodeRLE_BZ2

Performs the bzip2-specific RLE.

Syntax

```
IppStatus ippsEncodeRLE_BZ2_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen,
IppRLEState_BZ2* pRLEState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pRLEState	Pointer to the bzip2-specific RLE state structure.

Description

This function performs RLE encoding with thresholding equal to 4. It processes the input data *ppSrc* and writes the results to the *pDst* buffer. The function uses the bzip2-specific RLE state structure *pRLEState*. This structure must be initialized by `ippsEncodeRLEInit_BZ2` beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the source or destination buffer is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

EncodeRLEFlush_BZ2

Flushes the remaining data after RLE.

Syntax

```
IppStatus ippsEncodeRLEFlush_BZ2_8u(Ipp8u* pDst, int* pDstLen, IppRLEState_BZ2*
pRLEState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pRLEState	Pointer to the bzip2-specific RLE state structure.

Description

This function flushes the remaining data after RLE encoding with thresholding equal to 4. The function uses the initialized bzip2-specific RLE state structure *pRLEState*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the destination buffer is less than or equal to 0.

RLEGetInUseTable

Gets the pointer to the inUse vector from the RLE state structure.

Syntax

```
IppStatus ippsRLEGetInUseTable_8u(Ipp8u inUse[256], IppRLEState_BZ2* pRLEState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

inUse	Pointer to the inUse vector.
pRLEState	Pointer to the bzip2-specific RLE state structure.

Description

This function gets the pointer to the *inUse* vector (table) from the initialized bzip2-specific RLE state structure *pRLEState*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.

DecodeRLEStateInit_BZ2

Initializes the bzip2-specific RLE structure.

Syntax

```
IppStatus ippsDecodeRLEStateInit_BZ2_8u(IppRLEState_BZ2* pRLEState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pRLEState	Pointer to the bzip2-specific RLE structure.

Description

This function initializes the bzip2-specific RLE structure that contains parameters for the RLE in the external buffer. This structure is used by the function DecodeRLEState_BZ2.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if *pRLEState* pointer is NULL.

DecodeRLEState_BZ2

Performs the bzip2-specific RLE decoding.

Syntax

```
IppStatus ippsDecodeRLEState_BZ2_8u(Ipp8u** ppSrc, Ipp32u* pSrcLen, Ipp8u** ppDst,
Ipp32u* pDstLen, IppRLEState_BZ2* pRLEState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.
pRLEState	Pointer to the bzip2-specific RLE state structure.

Description

This function performs RLE decoding with thresholding equal to 4. It processes the input data *ppSrc* and writes the results to the *ppDst* buffer. The function uses the bzip2-specific RLE state structure *pRLEState*. This structure must be initialized by the functions DecodeRLEStateInit_BZ2 beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.

ippStsSizeErr	Indicates an error if length of the source or destination buffer is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

DecodeRLEStateFlush_BZ2

Flushes the remaining data after RLE decoding.

Syntax

```
IppStatus ippsDecodeRLEStateFlush_BZ2_8u(IppRLEState_BZ2* pRLEState, Ipp8u** ppDst,
Ipp32u* pDstLen);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pRLEState	Pointer to the bzip2-specific RLE state structure.
ppDst	Double pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer.

Description

This function flushes the remaining data after RLE decoding with thresholding equal to 4. The function uses the initialized bzip2-specific RLE state structure *pRLEState*.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the destination buffer is less than or equal to 0.

EncodeZ1Z2_BZ2

Performs the bzip2-specific Z1Z2 encoding.

Syntax

```
IppStatus ippsEncodeZ1Z2_BZ2_8u16u(Ipp8u** ppSrc, int* pSrcLen, Ipp16u* pDst, int*
pDstLen, int freqTable[258]);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer, after decoding - pointer to the size of the remaining data.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer, after decoding - pointer to the resulting size of the destination buffer.
freqTable	Table of frequencies collected for the alphabet symbols.

Description

This function performs the bzip2-specific Z1Z2 encoding.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the source or destination buffer is less than or equal to 0.
`ippStsDstSizeLessExpected`	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

DecodeZ1Z2_BZ2

Performs the bzip2-specific Z1Z2 decoding.

Syntax

```
IppStatus ippsDecodeZ1Z2_BZ2_16u8u(Ipp16u** ppSrc, int* pSrcLen, Ipp8u* pDst, int*
pDstLen);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

ppSrc	Double pointer to the source buffer.
pSrcLen	Pointer to the length of the source buffer, after decoding - pointer to the size of the remaining data.
pDst	Pointer to the destination buffer.
pDstLen	Pointer to the length of the destination buffer, after decoding - pointer to the resulting size of the destination buffer.

Description

This function performs the bzip2-specific Z1Z2 decoding.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the source or destination buffer is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

ReduceDictionary

Performs the dictionary reducing.

Syntax

```
IppStatus ippsReduceDictionary_8u_I(const Ipp8u inUse[256], Ipp8u* pSrcDst, int
srcDstLen, int* pSizeDictionary);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

inUse	Table of 256 values of the Ipp8u type.
pSrcDst	Pointer to the source and destination buffer.
srcDstLen	Length of the source and destination buffer.
pSizeDictionary	Pointer to the size of the dictionary on entry, and to the size of reduced dictionary after operation.

Description

This function performs the dictionary reducing.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the source and destination buffer is less than or equal to 0.

ExpandDictionary

Performs the dictionary expanding.

Syntax

```
IppStatus ippsExpandDictionary_8u_I(const Ipp8u inUse[256], Ipp8u* pSrcDst, int
srcDstLen, int sizeDictionary);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

inUse	Table of 256 values of the Ipp8u type.
pSrcDst	Pointer to the source and destination buffer.
srcDstLen	Length of the source and destination buffer.
sizeDictionary	Size of the dictionary on entry, and to the size of expanded dictionary after operation.

Description

This function performs the dictionary expanding.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the source and destination buffer is less than or equal to 0.

CRC32_BZ2

Computes the CRC32 checksum for the source data buffer.

Syntax

```
IppStatus ippsCRC32_BZ2_8u(const Ipp8u* pSrc, int srcLen, Ipp32u* pCRC32);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pSrc	Pointer to the source data buffer.
srcLen	Number of elements in the source data buffer.
pCRC32	Pointer to the accumulated checksum value.

Description

This function computes the checksum for *srcLen* elements of the source data buffer *pSrc* and stores it in the *pCRC32*. The checksum is computed using the CRC32 direct algorithm that is specific for the bzip2 coding.

You can use this function to compute the accumulated value of the checksum for multiple buffers by specifying as an input parameter the checksum value obtained in the preceding function call.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pSrc* pointer is NULL.
ippStsSizeErr	Indicates an error if the length of the source vector is less than or equal to 0.

EncodeHuffGetSize_BZ2

Computes the size of the internal state for bzip2-specific Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuffGetSize_BZ2_16u8u(int wndSize, int* pEncodeHuffStateSize);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

wndSize	Size of the block to be processed.
pEncodeHuffStateSize	Pointer to the size of the internal state for bzip2-specific Huffman coding.

Description

This function computes the size of the internal state structure for bzip2-specific Huffman encoding in dependence of the size of the block to be encoded.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pEncodeHuffStateSize* is NULL.
ippStsSizeErr	Indicates an error if *wndSize* is less than or equal to 0.

EncodeHuffInit_BZ2

Initializes the elements of the bzip2-specific internal state for Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuffInit_BZ2_16u8u(int sizeDictionary, const int freqTable[258],
const Ipp16u* pSrc, int srcLen, IppEncodeHuffState_BZ2* pEncodeHuffState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

sizeDictionary	Size of the dictionary.
freqTable	Table of frequencies of symbols.
pSrc	Pointer to the source vector.
srcLen	Length of the source vector.
pEncodeHuffState	Pointer to internal state structure for bzip2 specific Huffman coding.

Description

This function initializes the elements of the bzip2-specific internal state for Huffman encoding. This structure is used by the function `ippsEncodeHuff_BZ2`. The size of this buffer must be computed previously by calling the function `ippsEncodeHuffGetSize_BZ2`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is `NULL`.
ippStsSizeErr	Indicates an error if length of the source buffer is less than or equal to 0.

PackHuffContext_BZ2

Performs the bzip2-specific encoding of Huffman context.

Syntax

```
IppStatus ippsPackHuffContext_BZ2_16u8u(Ipp32u* pCode, int* pCodeLenBits, Ipp8u* pDst,
int* pDstLen, IppEncodeHuffState_BZ2* pEncodeHuffState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
pDst	Pointer to the destination vector.
pDstLen	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.

pEncodeHuffState	Pointer to internal state structure for bzip2 specific Huffman encoding.

Description

This function performs the bzip2-specific encoding of the *Huffman context*. The function uses the bzip2-specific Huffman encoding state structure *pEncodeHuffState*. This structure must be initialized by ippsEncodeHuffInit_BZ2 beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the destination buffer is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

EncodeHuff_BZ2

Performs the bzip2-specific Huffman encoding.

Syntax

```
IppStatus ippsEncodeHuff_BZ2_16u8u(Ipp32u* pCode, int* pCodeLenBits, Ipp16u** ppSrc,
int* pSrcLen, Ipp8u* pDst, int* pDstLen, IppEncodeHuffState_BZ2* pEncodeHuffState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the length of source vector.
pDst	Pointer to the destination vector.
pDstLen	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.
pEncodeHuffState	Pointer to internal state structure for bzip2 specific Huffman encoding.

Description

This function performs the bzip2-specific Huffman encoding. The function uses the bzip2-specific Huffman encoding state structure *pEncodeHuffState*. This structure must be initialized by ippsEncodeHuffInit_BZ2 beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the source or destination buffer is less than or equal to 0.
ippStsDstSizeLessExpected	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

DecodeHuffGetSize_BZ2

Computes the size of the internal state for bzip2-specific Huffman decoding.

Syntax

```
IppStatus ippsDecodeHuffGetSize_BZ2_8u16u(int wndSize, int* pDecodeHuffStateSize);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

wndSize	Size of the block to be processed.
pDecodeHuffStateSize	Pointer to the size of the internal state for bzip2-specific Huffman coding.

Description

This function computes the size of the internal state structure for bzip2-specific Huffman decoding.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the pointer *pDecodeHuffStateSize* is NULL.
ippStsSizeErr	Indicates an error if *wndSize* is less than or equal to 0.

DecodeHuffInit_BZ2

Initializes the elements of the bzip2-specific internal state for Huffman decoding.

Syntax

```
IppStatus ippsDecodeHuffInit_BZ2_8u16u(int sizeDictionary, IppDecodeHuffState_BZ2*
pDecodeHuffState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

sizeDictionary	Size of the dictionary.
pDecodeHuffState	Pointer to internal state structure for bzip2 specific Huffman coding.

Description

This function initializes the elements of the bzip2-specific internal state for Huffman decoding. This structure is used by the function `ippsDecodeHuff_BZ2`. The size of this buffer must be computed previously by calling the function `ippsDecodeHuffGetSize_BZ2`.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if the *pDecodeHuffState* pointer is NULL.
ippStsSizeErr	Indicates an error if *sizeDictionary* is less than or equal to 0.

UnpackHuffContext_BZ2

Performs the bzip2-specific decoding of Huffman context.

Syntax

```
IppStatus ippsUnpackHuffContext_BZ2_8u16u(Ipp32u* pCode, int* pCodeLenBits, Ipp8u**
ppSrc, int* pSrcLen, IppDecodeHuffState_BZ2* pDecodeHuffState);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the size of source buffer on input, pointer to the resulting length of the source vector on output.
pDecodeHuffState	Pointer to internal state structure for bzip2 specific Huffman decoding.

Description

This function performs the bzip2-specific decoding of the *Huffman context*. The function uses the bzip2-specific Huffman decoding state structure *pDecodeHuffState*. This structure must be initialized by `ippsDecodeHuffInit_BZ2` beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the destination buffer is less than or equal to 0.
ippStsSrcSizeLessExpected	Indicates a warning if size of the source buffer is insufficient to store all output elements.

DecodeHuff_BZ2

Performs the bzip2-specific Huffman decoding.

Syntax

```
IppStatus ippsDecodeHuff_BZ2_8u16u(Ipp32u* pCode, int* pCodeLenBits, Ipp8u** ppSrc,
int* pSrcLen, Ipp16u* pDst, int* pDstLen, IppDecodeHuffState_BZ2* pDecodeHuffState);
```

Include Files

ippdc.h

Domain Dependencies

Headers: ippcore.h, ippvm.h, ipps.h

Libraries: ippcore.lib, ippvm.lib, ipps.lib

Parameters

pCode	Pointer to the bit buffer.
pCodeLenBits	Number of valid bits in the bit buffer.
ppSrc	Double pointer to the source vector.
pSrcLen	Pointer to the size of source buffer.
pDst	Pointer to the destination vector.
pDstLen	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.
pDecodeHuffState	Pointer to internal state structure for bzip2 specific Huffman decoding.

Description

This function performs the bzip2-specific Huffman decoding. The function uses the bzip2-specific Huffman decoding state structure *pDecodeHuffState*. This structure must be initialized by ippsDecodeHuffInit_BZ2 beforehand.

Return Values

ippStsNoErr	Indicates no error.
ippStsNullPtrErr	Indicates an error if one of the pointers is NULL.
ippStsSizeErr	Indicates an error if length of the destination buffer is less than or equal to 0.
ippStsSrcSizeLessExpected	Indicates a warning if size of the source buffer is insufficient to store all output elements.

DecodeBlockGetSize_BZ2

Computes the size of the additional buffer for bzip2-specific block decoding.

Syntax

```
IppStatus ippsDecodeBlockGetSize_BZ2_8u(int blockSize, int* pBuffSize);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

blockSize	Size of the block to be processed.
pBuffSize	Pointer to the size of the buffer for bzip2-specific decoding.

Description

This function computes the size of the additional buffer for bzip2-specific decoding.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if the pointer *pBuffSize* is `NULL`.

DecodeBlock_BZ2

Performs the bzip2-specific block decoding.

Syntax

```
IppStatus ippsDecodeBlock_BZ2_16u8u(const Ipp16u* pSrc, int srcLen, Ipp8u* pDst, int*
pDstLen, int index, int dictSize, const Ipp8u inUse[256], Ipp8u* pBuff);
```

Include Files

`ippdc.h`

Domain Dependencies

Headers: `ippcore.h`, `ippvm.h`, `ipps.h`

Libraries: `ippcore.lib`, `ippvm.lib`, `ipps.lib`

Parameters

pSrc	Pointer to the source vector.
srcLen	Pointer to the length of the source vector.
pDst	Pointer to the destination vector.

pDstLen	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.
index	Index of first position for the inverse BWT transform
dictSize	Size of the reduced dictionary.
inUse	Table of 256 values of `Ipp8u` type.
pBuff	Pointer to the additional buffer.

Description

This function performs the bzip2-specific block decoding. The function uses the bzip2-specific additional buffer *pBuff*. The size of this buffer must be computed by the function DecodeBlockGetSize_BZ2 beforehand.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the pointers is `NULL`.
`ippStsSizeErr`	Indicates an error if length of the source or destination buffer is less than or equal to 0; or if *index* is greater than or equal to *srcLen*.
`ippStsSrcSizeLessExpected`	Indicates a warning if size of the source buffer is insufficient to store all output elements.

Long Term Evolution (LTE) Wireless Support Functions

11

NOTE
This functionality is available only within the Intel® System Studio suite.

This section describes functions that implement the Long Term Evolution (LTE) multiple input multiple output (MIMO) algorithm to estimate the minimum mean square error (MMSE).

The LTE MIMO uplink provides the following:

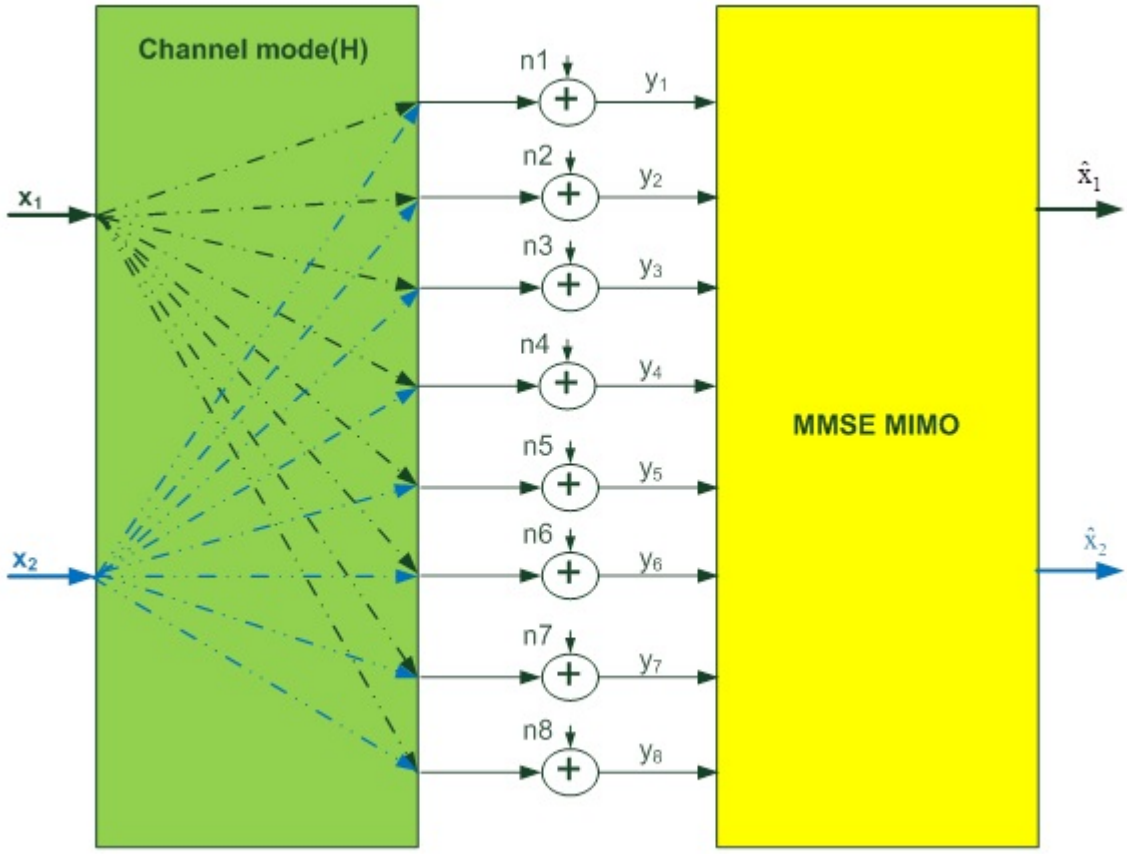
- *Spatial multiplexing* to enable high data rates within a limited bandwidth
- Additional *diversity* against fading on the radio channel
- *Beam-forming* to shape the overall antenna beam in a certain way to maximize the overall antenna gain in the direction of the target receiver

MIMO MMSE Estimator

NOTE
This functionality is available only within the Intel® System Studio suite.

The MIMO MMSE is based on the output of FFT (Y) and channel estimation (H).

The figure below shows the system model used for the MMSE estimation per subcarrier.



According to this figure, the basic equation is

$y=H*x+n$

where

$$\mathbf{y}=\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{Nr} \end{bmatrix},\ \mathbf{H}=\begin{bmatrix} H_{11} & H_{12} & \cdots & H_{1,Nt} \\ H_{21} & H_{22} & \cdots & H_{2,Nt} \\ \vdots & \vdots & \ddots & \vdots \\ H_{Nr,1} & H_{Nr,2} & \cdots & H_{Nr,Nt} \end{bmatrix},\text{and } \mathbf{x}=\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{Nt} \end{bmatrix}$$

- N_r is the number of receive antennas (2, 4, or 8)
- N_t is the number of transmit antennas (1 or 2)
- x is the data frequency domain symbol

MimoMMSE

Implements the MIMO MMSE estimator algorithm.

Syntax

```
IppStatus ippsMimoMMSE_1X2_16sc(Ipp16sc* pSrcH[2], int srcHStride2, int srcHStride1,
int srcHStride0, Ipp16sc* pSrcY[4][12], int Sigma2, IppFourSymb* pDstX, int
dstXStride1, int dstXStride0, int numSymb, int numSC, int SINRIdx, Ipp32f* pDstSINR,
int scaleFactor);
```

```
IppStatus ippsMimoMMSE_2X2_16sc(Ipp16sc* pSrcH[2], int srcHStride2, int srcHStride1,
int srcHStride0, Ipp16sc* pSrcY[4][12], int Sigma2, IppFourSymb* pDstX, int
dstXStride1, int dstXStride0, int numSymb, int numSC, int SINRIdx, Ipp32f* pDstSINR,
int scaleFactor);
```

```
IppStatus ippsMimoMMSE_1X4_16sc(Ipp16sc* pSrcH[2], int srcHStride2, int srcHStride1,
int srcHStride0, Ipp16sc* pSrcY[4][12], int Sigma2, IppFourSymb* pDstX, int
dstXStride1, int dstXStride0, int numSymb, int numSC, int SINRIdx, Ipp32f* pDstSINR,
int scaleFactor);
```

```
IppStatus ippsMimoMMSE_2X4_16sc(Ipp16sc* pSrcH[2], int srcHStride2, int srcHStride1,
int srcHStride0, Ipp16sc* pSrcY[4][12], int Sigma2, IppFourSymb* pDstX, int
dstXStride1, int dstXStride0, int numSymb, int numSC, int SINRIdx, Ipp32f* pDstSINR,
int scaleFactor);
```

Include Files

ippe.h

Parameters

pSrcH	Pointer to line 2 of the H matrix.
srcHStride2	Stride between H matrices (H[symb0] and (H[symb1]).
srcHStride1	Stride between rows of the H matrix (h00 and h10)
srcHStride0	Stride between elements of the row (h00 and h01).
pSrcY	Array of pointers to the RX signal Y. The maximum size is four TX antennas and 12 symbols.
Sigma2	Noise power.
numSymb	Number of symbols.
numSC	Number of subcarriers.
pDstX	Pointer to the estimated TX signal grouped by four symbols (quads).
dstXStride1	Stride between TX signals (X[ant0] and X[ant1]).
dstXStride0	Stride between quads inside one antenna.
SINRIdx	Index of symbol to calculate the SINR.
pDstSINR	Pointer to an array of SINR for layer 1,2.
scaleFactor	Scale factor, refer to Integer Scaling.

Description

NOTE
This functionality is available only within the Intel® System Studio suite.

This function implements the MMSE estimator algorithm. The MMSE estimation process consists of the following steps:

1. Calculate $B=H^H*H$, where H is the channel matrix, and superscript ^{H} denotes Hermitian operator (transpose and conjugate).
2. Calculate $A=H^H*H+\sigma^2_n*I_{Nt}$, where

   N_t is the number of transmit antennas (1 or 2).
3. Calculate $A^{-1}=(H^H*H+\sigma^2_n*I_{Nt})^{-1}$.
4. Calculate $Z=H^H*y$.
5. Calculate $x=A^{-1}*Z=(H^H*H+\sigma^2_n*I_{Nt})^{-1}*H^H*y$.
6. Calculate $D=W*H=A^{-1}*H^H*H$.

Signal to interference plus noise ratio (SINR) is computed as follows:

$$pSINR_1 = \frac{\sum_{k=0}^{M-1}\left(x_1^k\right)^H x_1^k}{\sum_{k=0}^{M-1}\left\{\left(\frac{\mathbf{H}^H\mathbf{H}}{\sigma_n^2}+\mathbf{I}_{N_t}\right)^{-1}\right\}_{jj}}, j=0$$

$$pSINR_2 = \frac{\sum_{k=0}^{M-1}\left(x_2^k\right)^H x_2^k}{\sum_{k=0}^{M-1}\left\{\left(\frac{\mathbf{H}^H\mathbf{H}}{\sigma_n^2}+\mathbf{I}_{N_t}\right)^{-1}\right\}_{jj}}, j=1$$

where

M is the number of subcarriers.

The destination data is grouped by four symbols.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error if one of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error if *numSymb* or *numSC* is less than, or equal to zero.

Handling of Special Cases

A

Some mathematical functions implemented in Intel IPP are not defined for all possible argument values. This appendix describes how the corresponding Intel IPP functions used in signal processing domains handle situations when their input arguments fall outside the range of function definition or may lead to ambiguously determined output results.

The table below summarizes these special cases for general vector functions described in Essential Functions and lists result values together with status codes returned by these functions. The status codes ending with `Err` (except for the `ippStsNoErr` status) indicate an error. When an error occurs, the function execution is interrupted. All other status codes indicate that the input argument is outside the range, but the function execution is continued with the corresponding result value.

Special Cases for Intel IPP Signal Processing Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
Sqrt	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
Div, Div_Round	8u	Div (0/0)	0	ippStsDivByZero
		Div (x/0)	IPP_MAX_8U	ippStsDivByZero
	16s	Div (0/0)	0	ippStsDivByZero
		Div (x/0), x>0	IPP_MAX_16S	ippStsDivByZero
		Div (x/0), x<0	IPP_MIN_16S	ippStsDivByZero
	16sc	Div (0/0)	0	ippStsDivByZero
		Div (x/0)	0	ippStsDivByZero
	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
	64f	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0), x>0	INF_32F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_32F	ippStsDivByZero

Function Base Name	Data Type	Case Description	Result Value	Status Code
	64fc	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0)	NAN_32F	ippStsDivByZero
DivC	all	Div(x/0)	-	ippStsDivByZeroErr
Ln	16s	Ln (0)	IPP_MIN_16S	ippStsLnZeroArg
		Ln (x<0)	IPP_MIN_16S	ippStsLnZeroArg
	32s	Ln (0)	IPP_MIN_32S	ippStsLnZeroArg
		Ln (x<0)	IPP_MIN_32S	ippStsLnNegArg
	32f	Ln (x<0)	NAN_32F	ippStsLnNegArg
		Ln(x<IPP_MINABS_32F)	INF_NEG_32F	ippStsLnZeroArg
	64f	Ln (x<0)	NAN_64F	ippStsLnNegArg
		Ln(x<IPP_MINABS_64F)	INF_NEG_64F	ippStsLnZeroArg
Exp	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

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 different results for 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, the logarithm function ippiLn operating on 16s data treats zero argument values in the same way for all its flavors ippsLn_16s_Sfs and ippiLn_16s_ISfs.

The table below summarizes special cases for fixed-accuracy arithmetic functions.

Special Cases for Intel IPP Fixed-Accuracy Arithmetic Functions

Function Base Name	Data Type	Case Description	Result Value	Status Code
Inv	32f	Inv (x=+0)	INF_32F	ippStsSingularity
		Inv (x=-0)	-INF_32F	
	64f	Inv (x=+0)	INF_64F	ippStsSingularity
		Inv (x=-0)	-INF_64F	

Function Base Name	Data Type	Case Description	Result Value	Status Code
				ippStsSingularity
				ippStsSingularity
Div	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
				ippStsSingularity
				ippStsSingularity
				ippStsSingularity
				ippStsSingularity
Sqrt	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
InvSqrt	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

Function Base Name	Data Type	Case Description	Result Value	Status Code
InvCbrt	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
Pow3o2	32f	Pow3o2 (x<0)	NAN_32F	ippStsDomain
		Pow3o2 (x=-INF)	NAN_32F	ippStsDomain
	64f	Pow3o2 (x<0)	NAN_64F	ippStsDomain
		Pow3o2 (x=-INF)	NAN_64F	ippStsDomain
Pow, Powx	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	ippStsSingularity
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsDomain
		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	ippStsSingularity
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity
Exp	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
Expm1	32f	Expm1(x), x>overflow	INF_32F	ippStsOverflow
	64f	Expm1(x), x>overflow	INF_64F	ippStsOverflow

Function Base Name	Data Type	Case Description	Result Value	Status Code
Ln, Log10	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
Log1p	32f	Ln(x<-1)	NAN_32F	ippStsDomain
		Ln(x=-INF)	NAN_32F	ippStsDomain
		Ln(x=-1)	-INF_32F	ippStsSingularity
	64f	Ln(x<-1)	NAN_64F	ippStsDomain
		Ln(x=-INF)	NAN_64F	ippStsDomain
		Ln(x=-1)	-INF_64F	ippStsSingularity
Cos	32f	Cos(INF)	NAN_32F	ippStsDomain
	64f	Cos(INF)	NAN_64F	ippStsDomain
Sin	32f	Sin(INF)	NAN_32F	ippStsDomain
	64f	Sin(INF)	NAN_64F	ippStsDomain
SinCos	32f	SinCos(INF)	NAN_32F, NAN_32F	ippStsDomain
	64f	SinCos(INF)	NAN_64F, NAN_64F	ippStsDomain
CIS	32fc	CIS(INF)	NAN_32F, NAN_32F	ippStsDomain
	64fc	CIS(INF)	NAN_64F, NAN_64F	ippStsDomain
Tan	32f	Tan(INF)	NAN_32F	ippStsDomain
	64f	Tan(INF)	NAN_64F	ippStsDomain
Acos	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
Asin	32f	Asin(x), |x|>1	NAN_32F	ippStsDomain
		Asin(INF)	NAN_32F	ippStsDomain
	64f	Asin(x), |x|>1	NAN_64F	ippStsDomain
		Asin(INF)	NAN_64F	ippStsDomain
Cosh	32f	Cosh(x), |x|>overflow	INF_32F	ippStsOverflow
	64f	Cosh(x), |x|>overflow	INF_64F	ippStsOverflow

Function Base Name	Data Type	Case Description	Result Value	Status Code
Sinh	32f	Sinh(x), |x|>overflow	INF_32F	ippStsOverflow
	64f	Sinh(x), |x|>overflow	INF_64F	ippStsOverflow
Acosh	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
Atanh	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
Erfc	32f	Erfc(x), |x|>underflow	0	ippStsUnderflow
	64f	Erfc(x), |x|>underflow	0	ippStsUnderflow
ErfInv	32f	ErfInv(x=1)	INF_32F	ippStsSingularity
		ErfInv(x=-1)	-INF_32F	ippStsSingularity
		ErfInv(x), |x|>1	NAN_32F	ippStsDomain
		ErfInv(INF)	NAN_32F	ippStsDomain
	64f	ErfInv(x=1)	INF_64F	ippStsSingularity
		ErfInv(x=-1)	-INF_64F	ippStsSingularity
		ErfInv(x), |x|>1	NAN_64F	ippStsDomain
		ErfInv(INF)	NAN_64F	ippStsDomain
ErfcInv	32f	ErfcInv(x=2)	-INF_32F	ippStsSingularity
		ErfcInv(x=0)	INF_32F	ippStsSingularity
		ErfcInv(x), |x|<0	NAN_32F	ippStsDomain
		ErfcInv(x), |x|>2	NAN_32F	ippStsDomain
		ErfcInv(INF)	NAN_32F	ippStsDomain
	64f	ErfcInv(x=2)	-INF_64F	ippStsDomain

Function Base Name	Data Type	Case Description	Result Value	Status Code
		ErfcInv(x=0)	INF_64F	ippStsSingularity
		ErfcInv(x), |x|<0	NAN_64F	ippStsSingularity
		ErfcInv(x), |x|>2	NAN_64F	ippStsDomain
		ErfcInv(INF)	NAN_64F	ippStsDomain
				ippStsDomain
CdfNorm	32f	CdfNorm(x), |x|<underflow	0	ippStsUnderflow
	64f	CdfNorm(x), |x|<underflow	0	ippStsUnderflow
CdfNormInv	32f	CdfNormInv(x=1)	INF_32F	ippStsSingularity
		CdfNormInv(x=0)	-INF_32F	ippStsSingularity
		CdfNormInv(x), x<0	NAN_32F	ippStsDomain
		CdfNormInv(x), x>1	NAN_32F	ippStsDomain
		CdfNormInv(INF)	NAN_32F	ippStsDomain
	64f	CdfNormInv(x=1)	INF_64F	ippStsSingularity
		CdfNormInv(x=0)	-INF_64F	ippStsSingularity
		CdfNormInv(x), x<0	NAN_64F	ippStsDomain
		CdfNormInv(x), x>1	NAN_64F	ippStsDomain
		CdfNormInv(INF)	NAN_64F	ippStsDomain

Bibliography for Signal Processing

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], [Jack89], and [Zie83] are undergraduate signal processing texts. [Opp89] is a much revised edition of the classic [Opp75]; [Jack89] is more concise than the others; and [Zie83] also covers continuous-time systems.

[AMRWB+] 3GPP Technical Specification 26.290: *Extended Adaptive Multi-Rate - Wideband (AMR-WB+) codec; Transcoding functions*, v.6.3.0, rel.6, June-2005.

[Arn97] Z. Arnavut, S. Magliveras. *Block-Sorting and Compression*. IEEE Data Compression Conference, Snowbird, Utah, pp.181-190, March 1997.

[Ash94] M.R. Asharif and F. Amano. *Acoustic Echo Canceler Using the FBAF Algorithm*. IEEE Trans. Comm., Vol. 42, No. 12, pp. 3090-3094, Dec. 1994.

[Berl68] Elwin R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill Book Company, 1968

[Bla84] Richard E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley Publishing Company, 1984

[Bris94] C. Brislawn. *Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks*. Los Alamos Report LA-UR-94-1747, 1994.

[Casey08] Sh.Casey. *x86 and SSE Floating Point Assists in IA-32: Flush-To-Zero (FTZ) and Denormals-To-Zero (DAZ)*. Intel Software Network, October 2008. http://software.intel.com/en-us/articles/x87-and-sse-floating-point-assists-in-ia-32-flush-to-zero-ftz-and-denormals-are-zero-daz/

[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).

[Cast93] G.Castagnoli, S.Brauer, M.Herrmann. *Optimization of cyclic redundancy-check codes with 24 and 32 parity bits*. IEEE Transactions on Communications, Vol.41, No.6, June, 1993, pp.883-892.

[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 Enhancement*. IEEE Signal Proc. Letters, Vol. 9, No. 1, Jan. 2002, pp. 12-15.

[Cro83] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets*. Springer Verlag, Pennsylvania, 1992.

[Eph84] Y. Ephraim and D. Malah. Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator. IEEE Trans. ASSP, Vol. 32, No. 6, Dec. 1984, pp. 1109-1121.

[ETSI02] ETSI TS 102 114:2002 (DTS Coherent Acoustics - Core and Extensions)(2002). http://www.etsi.org/services_products/freestandard/home.htm

[EC126] ETSI TS 126 404 V6.0.0. UMTS. *General audio codec audio processing functions; Enhanced aacPlus general audio codec; Encoder specification*; SBR PART - 3GPP TS 26.404 version 6.0.0 Release 6 (09/2004).

[ES201] ETSI ES 201 108 V1.1.2. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Front-end feature extraction algorithm; Compression algorithms*.

[ES202] ETSI ES 202 050 V1.1.1. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Advanced front-end feature extraction algorithm; Compression algorithms.*

[Feig92] E. Feig and S. Winograd. *Fast algorithms for DCT. IEEE Transactions on Signal Processing*, vol.40, No.9, 1992.

[Gal75] R. Gallager, D.Van Voorhis. *Optimal source codes for geometrically distributed integer alphabets*. IEEE Transactions on Information Theory, vol. IT-21, No. 3, pp.228-230, 1975.

[Griff87] G.Griffiths, G.C.Stones. *The tea-leaf reader algorithm: an efficient implementation of CRC-16 and CRC-32.* Communications of the ACM , vol.30, No.7, 1987, pp.617-620.

[Ham83] R.W. Hamming. *Digital Filters*, Prentice-Hall, New Jersey 1983.

[Har78] F. Harris. On the Use of Windows. Proceedings of the IEEE, vol. 66, No.1, IEEE, 1978.

[Hay91] S. Haykin. *Adaptive Filter Theory*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[ICCC] *Intel (R) C++ Compiler 11.1. User and Reference Guides.*. Document number 304968-023US.

[ISO11172] ISO/IEC 11172-3 - Information technology. *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s*. Part 3: Audio (1993)

[ISO13818] ISO/IEC 13818-3 - Information technology. *Generic coding of moving pictures and associated audio information*. Part 3: Audio (1998).

[ISO14496] ISO/IEC 14496-3 - Information technology. *Coding of audio-visual objects.* Part 3: Audio (2001).

[ISO14496A] ISO/IEC 14496-3/Amd.1:2003 - *Bandwidth Extension* (2003).

[ISO14496B] ISO/IEC 14496-3/Amd.2:2004 - *Parametric coding for high-quality audio* (2004).

[ITU169] JITU-T Recommendation G.169. *Automatic Level Control Devices*, (06/99).

[ITU224] ITU-T Recommendation X.224 Annex D. *Checksum Algorithms*, (11/93), pp144,145.

[ITU722] ITU-T Recommendation G.722.1 (09/99), *Coding at 24 and 32 8 kbit/s for hand-free operation in systems with low frame loss*.

[ITU723] ITU-T Recommendation G.723.1 . *Dual Rate speech coder for Multimedia Communications transmitting at 5.3 and 6.3 Kbit/s*, (03/96).

[ITU723A] ITU-T Recommendation G.723.1 Annex A. *Silence compression scheme*, (11/96).

[ITU728] ITU-T Recommendation G.728. *Coding of Speech at 16 kbits/s Using Low-Delay Code Excited Linear Prediction*, 1992.

[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).

[ITU7291] ITU-T Recommendation G.729.1. *G729 Based Embeded Variable Bit-Rate Coder: a 8-32 kbits/s Scalable Wideband Coder Bitstream Interoperable with G.729*, (06/06).

[ITUV34] ITU-T Recommendation V.34. *A modem operating at data signalling rates of up to 33 600 bit/s for use on the general switched telephone network and on leased point-to-point 2-wire telephone-type circuit*. (02/98).

[Jack89] Leland B. Jackson. *Digital Filters and Signal Processing*. Kluwer Academic Publishers, second edition, 1989.

[Kab86] P. Kabal and P.Ramachandran. *The Computation of line Spectral Frequencies Using Chebyshev Polynomials. IEEE transaction on acoustic, speech and signal processing*, vol. ASSP-34, No.6, 1986.

[Lyn89] Paul A. Lynn. *Introductory Digital Signal Processing with Computer Applications*. John Wiley&Sons, Inc., New York, 1993.

[Mar01] R. Martin. *Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics*. IEEE Trans. Speech and Audio, Vol. 9, No. 5, July 2001, pp. 504-512.

[Med91] Y. Medan, E. Yair, D. Chazan. *Super Resolution Pitch Determination of Speech Signals*. IEEE Transactions on Signal Processing, vol 39, No.1, 1991.

[Mit93] Sanjit K. Mitra and James F. Kaiser editors. *Handbook for Digital Signal Processing*. John Wiley & Sons, Inc., New York, 1993

[Mit98] Sanjit K. Mitra. *Digital Signal Processing*. McGraw Hill, 1998.

[Mor02] Robert H.Morelos-Zaragoza. *The Art of Error Correcting Coding.* Wiley & Sons, Ltd., 2002.

[Nel92] M.R.Nelson. *File verification using CRC 32-bit cyclical redundancy check*. Dr.Dobb's Journal, vol. 17, Issue 5, 1992, p.64.

[NIC91] Nam Ik Cho and Sang Uk Lee. *Fast algorithm and implementation of 2D DCT*. IEEE Transactions on Circuits and Systems, vol. 31, No.3, 1991.

[Opp75] A.V. Oppenheim and R.W. Schafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Opp89] A.V. Oppenheim and R.W. Schafer. *Discrete-Time Signal Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Rab78] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals.* Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Rao90] K.R. Rao and P. Yip. *Discrete Cosine Transform. Algorithms, Advantages and Applications*. Academic Press, San Diego, 1990.

[RFC1950] P.Deutsch, J-L.Gailly. *ZLIB Compressed Data Format Specification* version 3.3, May, 1996. http://www.ietf.org/rfc/rfc1950.txt

[RFC1951] P.Deutsch. DEFLATE Compressed Data Format Specification version 1.3, May, 1996.

http://www.ietf.org/rfc/rfc1951.txt

[RFC1952] P.Deutsch. GZIP file format specification version 4.3, May, 1996. http://www.ietf.org/rfc/rfc1952.txt

[Seg78] R. Sedgewick. Implementing quicksort programs. Communications of the ACM, Vol. 21, No. 10, pp. 847-857, Oct. 1978.

[Storer82] J. Storer, T.Szymanski. Data Compression via Textual Substitution. Journal of the Association for Computing Machinery (ACM), vol.19, No. 4, pp.928-951, Oct.1982.

[Strang96] G. Strang and T. Nguyen. Wavelet and Filter Banks. Wellesley-Cambridge Press, 1996, pp. 153-157.

[Tuc92] R. Tucker. Voice Activity Detection Using a Periodicity Measure. IEEE Proceedings-I, vol. 139, No. 4, August 1992, pp. 377-380.

[Vai93] P. P. Vaidyanathan. Multirate Systems and Filter Banks. Prentice Hall, Englewood Cliffs, New Jersey.

[Wid85] B. Widrow and S.D. Stearns. Adaptive Signal Processing. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Zie83] Rodger E. Ziemer, William H. Tranter and D. Ronald Fannin. Signals and Systems: Continuous and Discrete. Macmillan Publishing Co., New York, 1983.

[Ziv77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, vol.23, No.3, pp.337-343, May 1977.

[ZLIB] J. Gailly and M. Adler. Zlib 1.2.8 Manual, April 2013. http://www.zlib.net/manual.html

Glossary

adaptive filter	Colors specified by each pixel's coordinates in a color space. Intel Integrated Performance Primitives for image processing use images with absolute colors.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.
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 *{x0, x1, x2}* and *{y0, y1, y2}* is performed as follows: *{x0, x1, x2} + {y0, y1, y2} = {x0 + y0, x1 + y1, x2 + y2}*.
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 operation	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 operation*.
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.
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 operation	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. The `Perm` 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*
Intel® 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

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z