# Intel®
# Signal Processing Library

*Reference Manual*


Issued in U.S.A.
Document Number 630508-012

# How to Use This Online Manual

Click to hide or show subtopics when the bookmarks are shown.

Double-click to jump to a topic when the bookmarks are shown.

Click to display bookmarks.

Click to display thumbnails.

Click to close bookmark or thumbnail view.

Click and use on the page to drag the page in vertical direction.

Click and drag to the page to magnify the view.

Click and drag to the page to reduce the view.

Click and drag the selection cursor to the page.

Click to go to the first page of the manual.

Click to go to the previous page.

Click to go to the next page.

Click to go to the last page.

Click to return back to the previous view. Use this button when you need to go back after using the jump button (see below).

Click to go forward from the previous view.

Click to set 100% of the page view.

Click to display the entire page within the window.

Click to fill the width of the window.

Click to open a dialog to search for a word or multiple words.

Click jump button on manual pages to jump to the related subjects. Use the return back icon above to go back.

**Printing an Online File.** Select **Print** from the **File** menu to print an online file. The dialog that opens allows you to print full text, range of pages, or selection.
**Viewing Multiple Online Manuals.** Select **Open** from the **File** menu, and open a .PDF file you need. Select **Cascade** from the **Window** menu to view multiple files.
**Resizing the Bookmark Area.** Drag the double-headed arrow that appears on the area's border as you pass over it.
**Jumping to Topics**. Throughout the text of this manual, you can jump to different topics by clicking on keywords printed in green color, underlined style or on page numbers in a box.

To return to the page from which you jumped, use the icon in the tool bar. Try this example:

This software is briefly described in the Overview; see page 1-1.

If you click on the phrase printed in green color, underlined style, or on the page number, the Overview opens.

# *Intel® Signal Processing Library*

Order Number: 630508-012

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

| Revision | Revision History | Date |
|---|---|---|
| -001 | Original issue. | 01/95 |
| -002 | Added descriptions of new functions for LMS and IIR filtering. Revised Chapter 2, “Error Handling.” | 05/95 |
| -003 | Added descriptions of 70 new functions. Revised Chapter 1. Reorganized material. | 08/95 |
| -004 | Added short integer real and short integer complex functions. Revised pagination. The title of the manual has changed: the former title was “Intel Native Signal Processing Library.” | 01/96 |
| -005 | Added complex integer flavor (“v”) to Goertzel functions. Chapter 5, “Supporting Functions,” included in -004 revision has been deleted. Remaining chapters have been renumbered accordingly. | 03/96 |
| -006 | Added: fast mode FFT and memory reclaim functions, FreeBitRevTbls and FreeTwdTbls in Chapter 7 and Chapter 10, Library Information. | 07/96 |
| -007 | This revision documents release 4.0 Beta of the library. DCT, Norm, bPowerSpectr, brPowerSpectr, bNormalize, and wavelet functions have been added as well as short integer flavors of low-level filter functions. | 05/97 |
| -008 | Documents release 4.0. Miscellaneous edits have been made. | 10/97 |
| -009 | Documents release 4.1. Added FIR filter design functions. | 05/98 |
| -010 | Added the functions DotProdExt, MaxExt, MinExt, NormExt, WtInitUserFilter, WtInitLen, logical and shift functions. | 01/99 |
| -011 | Documents release 4.2. Resampling, memory allocation, median filter, and arctangent functions have been added. | 10/99 |
| -012 | Documents release 4.5. Added bLg1, bLg2, Sum, and not-in-place windowing functions. | 08/00 |

The specification for the Library was developed by Intel Corporation in cooperation with Berkeley Design Technology, Inc. (Fremont, California), a firm specializing in digital signal processing technology.

# Contents

## Chapter 1 Overview

## Chapter 2 Error Handling



## Chapter 3 Arithmetic and Vector Manipulation Functions

## Chapter 4 Vector Data Conversion Functions

## Chapter 7 Fourier and Discrete Cosine Transform Functions

## Chapter 8 Filtering Functions

## Chapter 9 Convolution Functions

## Examples

## Figures



## Tables

*This page is left blank for double-sided printing*

*This page is left blank for double-sided printing*

# 1 Overview

Library function lists

The Intel® Signal Processing Library provides a set of signal processing routines optimized for general-purpose Intel® processors rather than specialized DSP processors. It is targeted to non-real-time applications.

## About This Software

The computing power of the latest generations of processors enables the use of many signal processing functions which previously were done by add-in DSPs. The library includes functions for finite impulse response (FIR) and infinite impulse response (IIR) filters, fast Fourier transforms (FFTs), wavelet transforms, tone generation, and many vector operations.

The library allows the CPU to process audio, video, and communications data using software only, rather than off-loading the data to fixed-function, dedicated digital signal processing hardware.

### Hardware/Software Requirements

The Signal Processing Library is designed for use on 32-bit processors (Intel386™ processors and higher). The library optimizes performance on processors of the latest generations, such as Pentium® III processors. The library includes a DLL which detects the processor on which it is running and loads an appropriate processor-specific DLL. The processor-specific DLLs are also provided. The user's system must be running the Windows* 3.1 (with the Win32s* extension), Windows 95, 98, or Windows NT* operating systems. The software requires an ANSI C compiler. See *Release Notes* for a complete list of compilers supported.

## Platforms Supported

The Signal Processing Library runs on Windows* platforms. The code and syntax used for function and variable declarations in this manual are written in the ANSI C style. However, versions of this library for different processors or operating systems may, of necessity, vary slightly. As a result, the declarations found in the include files distributed with the library may be slightly different than those shown here. Consult the Release Notes for more information.

# About This Manual

This manual describes the functions in the Intel® Signal Processing Library. The functions are organized around the types of computations performed in signal processing. Each function is introduced by its name and a one-line description of its purpose. This is followed by the function prototype and definitions of its arguments. The following sections are also included in each function description:

| | |
|---|---|
| *Discussion* | This section defines the function and describes the operation which the function performs. Often, code examples, as well as the equations which the function implements are included. |
| *Previous Tasks* | If present, this section describes any tasks you need to perform before calling the function. |
| *Application Notes* | If present, this section describes any special information which applications programmers or other users of the function need to know. |
| *Related Topics* | If present, this section lists the names of functions which perform related tasks. It also lists other sources of information on the operation which the function performs. |

All function names begin with the `nsp?` prefix. However, to help you quickly find the information you are looking for, the `nsp?` prefix is omitted from function names when they appear in the table of contents and in section titles. For example, the title above the section describing the `nsp?RealFft()` function is simply titled "RealFft."

## Meaning of “Implementation Dependent”

In this manual, certain behaviors and results of functions are identified as being “implementation dependent.” There are three reasons why implementation-dependent differences in the behavior and results of functions can occur:

- The library is implemented for use on different processors.
- The library is implemented for use on different operating systems.
- Different versions of the library are implemented.

The manual identifies implementation-dependent items to let you know where the behaviors and results might *potentially* be different. Implementation-dependent differences should be slight.

## Audience for This Manual

This manual assumes that you are a programmer with experience in signal processing and that you possess a working knowledge of signal processing vocabulary and principles. For sources of information on signal processing principles, refer to the Bibliography of this manual.

## Manual Organization

*Chapter 1*, “*Overview*”: Provides information about this manual as well as about the Signal Processing Library software. This chapter also provides general information which applies to the entire manual. For example, this chapter describes the notational conventions used in this manual, the data types for which the functions are implemented, and the contents of the header file `nsp.h`.

*Chapter 2*, “*Error Handling*”: Provides information on the error handling functions included with the library.

*Chapter 3*, “*Arithmetic and Vector Manipulation Functions*”: Provides information on the memory allocation functions, and functions available for initializing and combining scalars and vectors. The following vector manipulation functions are also provided: companding, measure, conjugation, sample manipulation, and correlation.

*Chapter 4*, "*Vector Data Conversion Functions*": Provides information on functions which perform the following conversion operations: components extraction and complex vector construction, floating point to integer and fixed point (and reverse) conversion of vector data, and cartesian to polar (and reverse) coordinate conversion.

*Chapter 5*, "*Sample-Generating Functions*": Provides information about functions which perform tone-generating, triangle-generating, and pseudo-random sample-generating with uniform and Gaussian distribution.

*Chapter 6*, "*Windowing Functions*": Provides information about the windowing functions included in the Signal Processing Library.

*Chapter 7*, "*Fourier and Discrete Cosine Transform Functions*": Provides information on functions which calculate the discrete Fourier transform (DFT) and the fast Fourier transform (FFT). Several variations of the basic DFT and FFT are included to support different application requirements, including normal order versus bit-reversed order, real versus complex signals, and complex arrays versus paired real arrays.

*Chapter 8*, "*Filtering Functions*": Provides information on how to create, use and implement the finite impulse response (FIR) filter, the least mean squares (LMS) adaptive filter, infinite impulse response (IIR) filter, and median filter.

*Chapter 9*, "*Convolution Functions*": Provides information on the functions that perform convolution operations.

*Chapter 10*, "*Wavelet Functions*": Provides information on the functions that perform wavelet-based decomposition and reconstruction of signals.

*Chapter 11*, "*Library Information*": Provides a function to query the version number and the name of the current Signal Processing Library.

*Appendix A*, "*Fast Fourier Transforms*": Provides notes and hints for using the fast Fourier transform algorithms.

*Appendix B*, "*Digital Filtering*": Provides a general background of digital filtering and introduces the concepts of the filters used by the Signal Processing Library.

*Appendix C*, "*Multi-Rate Filtering*": Provides a brief introduction to multi-rate filters, which may be unfamiliar to application programmers.

*Glossary*: Provides definitions of some of the terms used in this manual.
*Bibliography*: Provides references to the books cited in this manual.

## Related Publications

This manual is designed as a reference for the Intel® Signal Processing Library. The manual contains numerous references to additional textbooks on filters and signal processing. A bibliography is provided at the back of the manual.

If you need functions implementing signal processing and recognition algorithm primitives for speech and optical character recognition (OCR), refer to the *Intel® Recognition Primitives Library Reference Manual*, order number 637785.

# Notational Conventions

This section describes the notational conventions used by the Signal Processing Library and the notational conventions for mathematical symbols, data types, function names, and signal names used in this manual.

## Data Type Conventions

Many of the functions in the Signal Processing Library are available for both single-precision real (`float`) and double-precision real (`double`) floating point data types. Additionally, many of the functions are also available for complex numbers and vectors.

The Signal Processing Library provides structures which define a single-precision complex data type, `SCplx`, a double-precision complex data type, `DCplx`, and a short integer complex data type, `WCplx`. The definitions for these structures are listed in Figure 1-1.

**Figure 1-1 Structure Definitions for Complex Data Types**

**Single-Precision Complex**

```
typedef struct _SCplx {
       float re;
       float im;
} SCplx;
typedef struct _WCplx {
      short int re;
      short int im;
} WCplx;
```

**Double-Precision Complex**

```
typedef struct _DCplx {
       double re;
       double im;
} DCplx;
```

In most cases, scalar complex numbers are passed by value and returned by value, not reference.

Thus, a function can be available for the following data types:

- single-precision real (`float`)
- single-precision complex (`SCplx`)
- double-precision real (`double`)
- double-precision complex (`DCplx`)
- short integer real (`short`)
- short integer complex (`WCplx`)

Some functions can have more than these data types because they are able to accept and process input of differing data types. For example, a function can accept as input a complex signal and real filter coefficients.

A character code embedded within the function name indicates which data type can be used with a particular function. Table 1-1 lists the names of the data types and their corresponding character codes.

**Table 1-1 Data Types and Corresponding Character Codes**

| Data Type | C type/structure | Character Code |
|---|---|---|
| Single-Precision Real | float | s |
| Single-Precision Complex | SCplx | c |
| Double-Precision Real | double | d |
| Double-Precision Complex | DCplx | z |
| Short Integer Real | short | w |
| Short Integer Complex | WCplx | v |

In addition, a character code for a complex type (that is, `c`, `z`, or `v`) may be followed by the letter `r`. This indicates a complex vector stored as a pair of real vectors (that is, one vector stores the real part and another vector stores the imaginary part).

## Function Name Conventions

The names of Signal Processing Library functions always begin with the `nsp` prefix and have the following general format:

`nsp <` *`character code`* `> <` *`flags`* `> <` *`name`* `> <` *`mods`* `> ()`

where:

*`character code`* — One of the character codes described in Table 1-1 above (`s`, `c`, `d`, or `z`). The character code indicates which data type to use with the function. Some functions have multiple character codes or non-standard character codes. When this occurs, the function definition describes the exact meaning of the code.

*`flags`* — The *`flags`* field is optional and can be defined as `b` or `r`. The `b` flag indicates a block (or vector) variety of the function. A block variety of a function is generally equivalent to multiple

| | |
|---|---|
| | invocations of the non-block (scalar) function. The `r` flag indicates that the function only uses real-valued arrays. |
| *`name`* | Indicates the core functionality, such as `Tone`, `Fft`, or `Fir`. |
| *`mods`* | The *`mods`* field is optional and indicates a modification to the core functionality of the function group. Examples of *`mods`* are `Nip` (not-in-place) and `Na` (non-adaptive). |

## Examples

| | |
|---|---|
| `nspcFft()` | Computes the FFT of single-precision complex data. |
| `nspzFft()` | Computes the FFT of double-precision complex data. |
| `nspzFftNip()` | Has the modifier `Nip` (not-in-place). Computes the FFT of double-precision complex data using separate input and output arrays. |

## Function Name Shorthand

By convention, a question mark “`?`” is used to indicate any or all possible varieties of the function described in the manual. For example,

| | |
|---|---|
| `nsp?UpSample()` | Refers to all varieties of the `UpSample` function: `nspsUpSample()` `nspcUpSample()` `nspdUpSample()` and `nspzUpSample()`. |
| `nsp?Fft()` | Refers to all varieties of the `Fft` function: `nspcFft()` and `nspzFft()`. |

## Signal Name Conventions

In this manual, the notation:

`x(n)`

refers to a conceptual signal, while the notation:

`x[n]`

refers to an actual array. Typically, both of these are annotated to indicate a specific finite range of values:

`x(n)`, $0 \le n < N$

`x[n]`, $0 \le n < N$

Occasionally, the shorthand below is used to indicate a finite range of values:

`x(0)...x(N-1)`

`x[0]...x[N-1]`

## Mathematical Symbol Conventions

### Floor and Ceiling of Values

The notation:

$\lceil value \rceil$

indicates the ceiling of `value` (that is, the least integer greater than or equal to `value`), while the notation:

$\lfloor value \rfloor$

indicates the floor of `value` (that is, the greatest integer less than or equal to `value`).

### Complex Conjugates of Values

Given a complex value $a$, the complex conjugate is denoted as $a^*$:

$\mathrm{Re}(a^*) = \mathrm{Re}(a)$. $\mathrm{Im}(a^*) = -\mathrm{Im}(a)$

# Macros and Data Structure

The header file `nsp.h`, included with the Signal Processing Library, contains prototypes for all library functions, definitions for data types, and structures, and the most frequently used macros and constants.

## Constant Macros

The `nsp.h` header file contains the following definitions for epsilon (EPS), pi ($\pi$), degree-to-radian conversion, maximum and minimum value comparisons, and the values for `TRUE` and `FALSE`.

```
#define NSP_EPS  (1.0E-12)             /* a very small value */
#define NSP_PI   (3.14159265358979324) /* define value of PI */
#define NSP_2PI  (6.28318530717958648) /* PI*2 */
#define NSP_PI_2 (1.57079632679489662) /* PI/2 */
#define NSP_PI_4 (0.785398163397448310) /* PI/4 */
#define NSP_MAX_SHORT_INT (32767)      /* short integer max value*/
#define NSP_MIN_SHORT_INT (-32768)     /* short integer min value*/

#ifndef FALSE /* define the values of TRUE and FALSE */
# define FALSE 0
# define TRUE  1
#endif
```

## Function Macros

The only macro in this category is defined by the following statement:

```
#define NSP_DegToRad(deg) ((deg)/180.0 * NSP_PI)
      /* degree to radian conversion */
```

## Control Macros

The Signal Processing Library lets you choose the library functions that will be available to your program.  A number of macros have been created which access an include file and the function prototypes it defines.

To include the functions, define the names of the appropriate macros with the `#define` directive. The `#define` directive must always precede the `#include “nsp.h”` statement. For example, the statements

```
#define nsp_UsesVector
#include “nsp.h”
```

access the corresponding include (header) file and the prototypes for the scalar arithmetic and vector initialization functions defined there.

In this example, the include files for the FFT, the finite impulse response filter, and the error handler functions are made available to the program.

```
#define nsp_UsesFft
#define nsp_UsesFir
#include “nsp.h”
```

If you want to make all of the signal processing functions available to your program, define the `nsp_UsesAll` macro.

```
#define nsp_UsesAll
#include “nsp.h”
```

You do not need to include error handling and memory allocation header files or any macros because `nsp.h` includes them by default.

Table 1-2 lists the names of all of the control macros defined in `nsp.h`. The functions are listed in the “Intel® Signal Processing Library Functions” section later in this chapter.

**Table 1-2 Control Macros**

| Macro Name | Description |
|---|---|
| `nsp_UsesVector` | Declares the arithmetic functions. |
| `nsp_UsesConvolution` | Declares the convolution functions. |
| `nsp_UsesTransform` | Declares the discrete and fast Fourier transform functions. |
| `nsp_UsesFir` | Declares the finite impulse response filter functions. |
| `nsp_UsesIir` | Declares the infinite impulse response filter functions. |
| `nsp_UsesMedian` | Declares the median filter functions. |

continued ☛

**Table 1-2 Control Macros (continued)**

| Macro Name | Description |
|---|---|
| nsp_UsesLms | Declares the least mean squares adaptive filter functions. |
| nsp_UsesMisc | Declares the bit-reversal functions and twiddle factor functions. |
| nsp_UsesConversion | Declares vector data type and coordinate conversion functions. |
| nsp_UsesSampleGen | Declares the tone- and triangle-generating functions. |
| nsp_UsesWavelet | Declares the wavelet functions. |
| nsp_UsesWin | Declares the windowing functions. |
| nsp_UsesAll | All functions. |

## Compiler Macros

Table 1-3 lists the macros which your compiler should define in accordance with the ANSI C standard. These macros are used for the error handling functions (see Chapter 2, "Error Handling" for details).

**Table 1-3 Compiler Macros**

| Macro | Description |
|---|---|
| __DATE__ | The date of compilation as a string literal in the form "mm dd yy". |
| __FILE__ | A string literal representing the name of the file being compiled. |
| __LINE__ | The current line number as a decimal constant. |
| __STDC__ | The constant 1 under ANSI C conformance dialect (-Xc); for other dialects, 0 except for -Xk where this macro can be undefined. |
| __TIME__ | The time of compilation as a string literal in the form "hh:mm:ss". |

## Data Type Definitions

The `nsp.h` header file contains the following definitions for complex data: single-precision real (`SCplx`), double-precision real (`DCplx`), short integer (`WCplx`) and integer (`ICplx`) data types.

**NOTE.** *The Signal Processing Library supports signed short integers only. The unsigned integer data type is not supported.*
*The `ICplx` data type is for returned single values only.*

```
typedef struct _SCplx {
      float re;
      float im;
} SCplx;
typedef struct _DCplx {
      double re;
      double im;
} DCplx;
typedef struct _WCplx {
      short int re;
      short int im;
} WCplx;
typedef struct _ICplx {
      int re;
      int im;
} ICplx;
```

For all functions that return an `SCplx`, `WCplx`, or `ICplx` structure the Signal Processing Library contains its counterpart function which returns this structure through an additional pointer in the arguments list. This is done for compatibility with compilers which implement the returning of structures in different ways.

The name of the counterpart function is formed by appending the `Out` suffix to the name of the original function. For example, the function

```
SCplx nspcDotProd(const SCplx *srcA, const SCplx *srcB, int n);
```

has its counterpart as

```
void nspcDotProdOut(const SCplx *srcA, const SCplx *srcB, int n,
                    SCplx *val);
```

where the function value is returned in *`val`* pointer to the `SCplx` structure.

## Integer Scaling

Most integer functions in the Signal Processing Library perform their internal computations using a higher precision than the 16-bit integer data types used for input and output. This higher precision can be `long int` or `float`, depending on the implementation.

In addition to the regular set of arguments, most of the library integer functions use two variables, *`ScaleMode`* and *`ScaleFactor`*, which determine how the output vector is converted before function return.

A typical integer function for which the scaling of output is performed has the following format:

```
nspwdummy(..., int ScaleMode, int *ScaleFactor);
```

### Scaling Arguments

*`ScaleMode`* Indicates the scaling control options to be used in returning the output. There are two strategies to control scaling: output vector scaling control and integer overflow scaling control. These strategies are described below.

**Output Vector Scaling Control**

The output vector scaling control includes the following three modes:

`NSP_NO_SCALE`
No scaling is performed for the output vector. The output results can be erroneous if overflow or underflow occurs. With this mode, the overflow is handled by the overflow control option (see page 1-15 for integer overflow control). The *`ScaleFactor`* is ignored and can be `NULL`. This mode provides the fastest performance.

`NSP_FIXED_SCALE`
Scaling is performed in accordance with the *`ScaleFactor`* values (see page 1-15). The output is always multiplied by $2^{-\mathtt{ScaleFactor}}$ before function return. *`ScaleFactor`* is returned unchanged. The function will be implemented using a higher precision data type internally. The output then will be scaled according to the scale factor. If an overflow occurs during the translation, it is handled by the overflow control option (see page 1-15

for integer overflow control).

`NSP_AUTO_SCALE`
The output vector is automatically scaled up or down to prevent from the overflow or underflow and to provide the best precision. The scaling is accomplished by multiplying the output vector by $2^{-\mathtt{ScaleFactor}}$, and the argument *`ScaleFactor`* is returned. This is the most memory and time consuming, yet the safest mode.

**Integer Overflow Control**

The integer overflow control includes the following two modes:

`NSP_OVERFLOW`
When overflow or undeflow occurs, the most significant bits of the output vectors are truncated. This is the default overflow mode.

**NOTE.** *For some functions and some argument ranges, truncating the most significant bits means a* ***complete loss of precision****. Therefore, you should not usually rely on the values remaining after the truncation of the most significant bits.*

`NSP_SATURATE`
When overflow or underflow occurs, the output for `short int` data is clipped to `NSP_MAX_SHORT_INT` (=32767) or `NSP_MIN_SHORT_INT` (= -32768), respectively.

*`ScaleFactor`* The scale factor is defined as a pointer to an integer value. The scale mode dictates which scale factor must be used. With the `NSP_NO_SCALE` mode, the *`ScaleFactor`* argument has no meaning and will be ignored. With the `NSP_FIXED_SCALE` mode, *`ScaleFactor`* is an input argument. It points to the value to which the output vector should be scaled. With the `NSP_AUTO_SCALE` mode, *`ScaleFactor`* is an output argument. It points to a variable in which the *`ScaleFactor`* is returned.

Upon function return, the actual output vector is defined as `actual_output = output` * $2^{\mathtt{ScaleFactor}}$.

### Compatibility with the Recognition Primitives Library

If you are using Intel® Recognition Primitives Library (RPL), you can continue to use the RPL's scale mode literals. The RPL scale modes are mapped to the signal processing library scale modes as follows:

```
#define RPL_NO_SCALE     NSP_NO_SCALE | NSP_OVERFLOW
#define RPL_SATURATE     NSP_NO_SCALE | NSP_SATURATE
#define RPL_FIXED_SCALE NSP_FIXED_SCALE | NSP_OVERFLOW
#define RPL_AUTO_SCALE  NSP_AUTO_SCALE
```

### Application Notes

Unless otherwise specified, the input data of the integer functions is treated as having no scaling. The value of the data is in the range of `NSP_MAX_SHORT_INT` to `NSP_MIN_SHORT_INT`. The application should track the input data scaling by maintaining the scale factors separately and doing additional scaling to adjust these data. In this release of the Signal Processing Library, the scaling is performed for the output data only.

With the `NSP_FIXED_SCALE` mode, if *`ScaleMode`* is `NULL`, it is treated as a pointer to a value of zero. The main purpose of this condition is to simplify the coding.

---

**NOTE.** *To obtain the best performance results with the `NSP_NO_SCALE` mode, the code might not include any higher precision representation for the internal data. In this case, the overflow condition might occur during the intermediate calculations. Inaccuracy might then propagate in the consecutive calculations thus generating erroneous results. You should examine your application to see if this scaling mode is appropriate.*

---

# Intel® Signal Processing Library Functions

Tables 1-4 through 1-22 describe the functions available in the Signal Processing Library. The table titles include the names of the macros (in parentheses) that define the functions listed in the table.

### Table 1-4 Error Handler Functions (macro included by default)

Error Handler Functions

| Function Name | Description |
|---|---|
| `Error` | Performs basic error handling. |
| `ErrStr` | Translates an error/status code into a textual description. |
| `GetErrMode` | Gets the error mode which describes how the error is processed. |
| `GetErrStatus` | Gets the error code which describes the type of error being reported. |
| `GuiBoxReport` | Reports errors to Windows* message box. |
| `NulDevReport` | Reports absence of error messages. |
| `RedirectError` | Assigns a new error handler to call when an error occurs. |
| `SetErrMode` | Sets the error mode which describes how the error is processed. |
| `SetErrStatus` | Sets the error code which describes the error that is being reported. |
| `StdErrReport` | Returns error messages to `stderr`. |

### Table 1-5 Memory Allocation Functions (macro included by default)

Memory Allocation Functions

| Function Name | Description |
|---|---|
| `Malloc` | Allocates 32-byte aligned memory block for a given number of data items. |
| `Free` | Frees memory block, previously allocated by `nsp?Malloc` function. |

**Table 1-6 Arithmetic, Logical, and Vector Manipulation Functions (nsp_UsesVector)**

Arithmetic and Vector Manipulation Functions

| Function Name | Description |
|---|---|
| `Add` | Adds two complex values. |
| `AutoCorr` | Estimates a normal, biased or unbiased auto-correlation of an input vector and stores the result in a second vector. |
| `bAbs1` | Changes vector elements to their absolute values. |
| `bAbs2` | Computes the absolute values of elements in a vector and stores the result in a second vector. |
| `bAdd1` | Adds a value to each element of a vector. |
| `bAdd2` | Adds the elements of two vectors. |
| `bAdd3` | Adds the elements of two vectors and stores the result in a third vector. |
| `bAnd1` | Computes the bitwise AND of a scalar and each element of a vector. |
| `bAnd2` | Computes the bitwise AND of the corresponding elements of two vectors. |
| `bAnd3` | Computes the bitwise AND of the elements of two vectors and stores the results in a third vector. |
| `bArctan1` | Computes the arctangent of each element of a vector in-place. |
| `bArctan2` | Computes the arctangent of each element of a vector and stores the results in a second vector. |
| `bConj1` | Computes the complex conjugate of a vector. |
| `bConj2` | Computes the complex conjugate of a vector and stores the result in a second vector. |
| `bConjExtend1` | Computes the conjugate-symmetric extension of a vector in-place. |
| `bConjExtend2` | Computes the conjugate-symmetric extension of a vector and stores the result in a second vector. |
| `bConjFlip2` | Computes the conjugate of a vector and stores the result, in reverse order, in a second vector. |
| `bCopy` | Initializes a vector with the contents of a second vector. |

continued ☞

**Table 1-6 Arithmetic, Logical, and Vector Manipulation Functions (nsp_UsesVector)** (continued)

| Function Name | Description |
|---|---|
| `bExp1` | Computes e to the power of each element of a vector in-place. |
| `bExp2` | Computes e to the power of each element of a vector and stores the resuls in a second vector. |
| `bInvThresh1` | Computes the inverse of the elements of a vector in-place. |
| `bInvThresh2` | Computes the inverse of the elements of a vector and stores the result in a second vector. |
| `bLg1` | Computes the decimal logarithm of each element of a vector in-place. |
| `bLg2` | Computes the decimal logarithm of each element of a vector and stores the result in a second vector. |
| `bLn1` | Computes the natural logarithm of each element of a vector in-place. |
| `bLn2` | Computes the natural logarithm of each element of a vector and stores the result in a second vector. |
| `bMpy1` | Multiplies each element of a vector by a value. |
| `bMpy2` | Multiplies the elements of two vectors and stores the result in the multiplicand vector. |
| `bMpy3` | Multiplies the elements of two vectors and stores the result in a third vector. |
| `bNormalize` | Subtracts a constant from vector elements and divides the result by another constant. |
| `bNot` | Computes the bitwise NOT of all vector elements. |
| `bOr1` | Computes the bitwise OR of a scalar and each element of a vector. |
| `bOr2` | Computes the bitwise OR of the corresponding elements of two vectors. |
| `bOr3` | Computes the bitwise OR of the elements of two vectors and stores the results in a third vector. |
| `bSet` | Initializes a vector to a specified value. |
| `bShiftL` | Shifts bits in all vector elements to the left. |
| `bShiftR` | Shifts bits in all vector elements to the right. |
| `bSqr1` | Computes the square of each element of a vector in-place. |

continued ☞

**Table 1-6 Arithmetic, Logical, and Vector Manipulation Functions (nsp_UsesVector)** (continued)

| Function Name | Description |
|---|---|
| `bSqr2` | Computes the square of each element of a vector and stores the result in a second vector. |
| `bSqrt1` | Computes the square root of each element of a vector in-place. |
| `bSqrt2` | Computes the square root of each element of a vector and stores the result in a second vector. |
| `bSub1` | Subtracts a value from each element of a vector. |
| `bSub2` | Subtracts the elements of two vectors. |
| `bSub3` | Subtracts the elements of two vectors and stores the result in a third vector. |
| `bThresh1` | Performs the threshold operation on a vector in-place. |
| `bThresh2` | Performs the threshold operation on a vector and places the result in a second vector. |
| `bXor1` | Computes the bitwise XOR of a scalar and each element of a vector. |
| `bXor2` | Computes the bitwise XOR of the corresponding elements of two vectors. |
| `bXor3` | Computes the bitwise XOR of the elements of two vectors and stores the results in a third vector. |
| `bZero` | Initializes a vector to zero. |
| `Conj` | Conjugates a complex value. |
| `CrossCorr` | Estimates the cross-correlation of two vectors and stores the result in a third vector. |
| `Div` | Divides two complex values. |
| `DotProd`, `DotProdExt` | Computes a dot product of two vectors. |
| `DownSample` | Down-samples a signal, conceptually decreasing its sampling rate by an integer factor. |
| `Max, MaxExt` | Returns the maximum value of a vector. |
| `Mean` | Computes the mean (average) of a vector. |

continued ☛

**Table 1-6 Arithmetic, Logical, and Vector Manipulation Functions (nsp_UsesVector)** (continued)

| Function Name | Description |
|---|---|
| `Min, MinExt` | Returns the minimum value of a vector. |
| `Mpy` | Multiplies two complex values. |
| `Norm, NormExt` | Computes the C, $L_1$, or $L_2$ norm of a vector. |
| `Sum` | Computes the sum of vector elements. |
| `SampInit` | Initializes resampling parameters structure. |
| `Samp` | Performs resampling of the input signal using the multi-rate FIR filter. |
| `SampFree` | Frees memory allocated to resampling data |
| `StdDev` | Computes the variance (standard deviation) of a vector. |
| `Sub` | Subtracts two complex values. |
| `UpSample` | Up-samples a signal, conceptually increasing its sampling rate by an integer factor. |

**Table 1-7 Vector Data Conversion Functions (nsp_UsesConversion)**

Vector Data Conversion Functions

| Function Name | Description |
|---|---|
| `b2RealToCplx` | Returns a complex vector constructed from the real and imaginary parts of an input vector. |
| `bALawToLin` | Converts 8-bit A-law encoded samples to linear samples. |
| `bCartToPolar` | Converts the elements of a complex vector to a polar coordinate form. |
| `bCplxTo2Real` | Returns the real and imaginary parts of a complex vector in two respective vectors. |
| `bFixToFloat` | Converts the fixed-point data of a vector to floating-point and stores the result in a second vector. |
| `bFloatToFix` | Converts the floating-point data of a vector to fixed-point and stores the result in a second vector. |
| `bFloatToInt` | Converts the floating-point data of a vector to integer format and stores the result in a second vector. |

continued ☞

**Table 1-7 Vector Data Conversion Functions (nsp_UsesConversion)** (continued)

| Function Name | Description |
|---|---|
| bFloatToS7Fix | Converts the floating-point data of a vector to fixed-point and stores the result in a second vector, assuming a fixed-point format of S.7. |
| bFloatToS15Fix | Converts the floating-point data of a vector outfoxed-point and stores the result in a second vector, assuming a fixed-point format of S.15. |
| bFloatToS1516Fix | Converts the floating-point data of a vector to fixed-point and stores the result in a second vector, assuming a fixed-point format of S15.16. |
| bFloatToS31Fix | Converts the floating-point data of a vector to fixed-point and stores the result in a second vector, assuming a fixed-point format of S.31. |
| bImag | Returns the imaginary part of a complex vector in a second vector. |
| bIntToFloat | Converts an integer vector to floating-point format and stores the result in a second vector. |
| bLinToALaw | Encodes the linear samples in a vector using the 8-bit A-law format and stores the result in a second vector. |
| bLinToMuLaw | Encodes the linear samples in a vector using the 8-bit μ-law format and stores the result in a second vector. |
| bMag | Computes the magnitudes of elements of a complex vector and stores the result in a second vector. |
| bMuLawToLin | Converts samples from the 8-bit μ-law encoded format to linear samples. |
| bPhase | Returns the phase angles of elements of a complex vector in a second vector. |
| bPolarToCart | Converts the polar form magnitude/phase pairs stored in individual vectors into a complex vector and stores the result in one vector. |
| bPowerSpectr | Returns the power spectrum of a complex vector in a second vector. |

continued ☛

**Table 1-7 Vector Data Conversion Functions (nsp_UsesConversion)** (continued)

| Function Name | Description |
|---|---|
| bReal | Returns the real part of a complex vector in a second vector. |
| brCartToPolar | Converts the complex real/imaginary (cartesian coordinate X/Y) pairs of individual input vectors to polar coordinate form. The function stores the magnitude (radius) component of each element in one vector and the phase (angle) component of each element in another vector. |
| brMag | Computes the magnitudes of elements of the complex vector whose real and imaginary components are specified in individual vectors. Stores the result in a third vector. |
| brPhase | Computes the phase angles of elements of the complex input vector whose real and imaginary components are specified in real and imaginary vectors, respectively. The function stores the resulting phase angles in a third vector. |
| brPolarToCart | Converts the polar form magnitude/phase pairs stored in the individual vectors into a complex vector. The function stores the real component of the result in a third vector and the imaginary component in a fourth vector. |
| brPowerSpectr | Computes the power spectrum of a complex vector whose real and imaginary components are two vectors. Stores the results in a third vector. |
| bS7FixToFloat | Converts the fixed-point data of a vector to floating-point and stores the result in a second vector, assuming a fixed-point format of S.7. |
| bS15FixToFloat | Converts the fixed-point data of a vector to floating-point and stores the result in a second vector, assuming a fixed-point format of S.15. |
| bS1516FixToFloat | Converts the fixed-point data of a vector to floating-point and stores the result in a second vector, assuming a fixed-point format of S15.16. |
| bS31FixToFloat | Converts the fixed-point data of a vector to floating-point and stores the result in a second vector, assuming a fixed-point format of S.31. |

**Table 1-8 Sample-Generating Functions (nsp_UsesSampleGen)**

Sample-Generating Functions

| Function Name | Description |
|---|---|
| `bRandGaus` | Computes pseudo-random samples with a Gaussian distribution and stores them in a vector. |
| `bRandUni` | Computes pseudo-random samples with a uniform distribution and stores them in a vector. |
| `bTone` | Produces a user-specified number of consecutive samples of a sinusoid. |
| `bTrngl` | Produces a user-specified number of consecutive samples of a triangle. |
| `RandGaus` | Computes the next pseudo-random sample with a Gaussian distribution. |
| `RandGausInit` | Initializes a state data structure required to generate pseudo-random samples with a Gaussian distribution. |
| `RandUni` | Computes the next pseudo-random sample with a uniform distribution. |
| `RandUniInit` | Initializes a state required to generate a structure of pseudo-random samples with a uniform distribution. |
| `Tone` | Produces the next sample of a sinusoid. |
| `ToneInit` | Initializes a sinusoid with a given frequency, phase, and magnitude. |
| `Trngl` | Produces the next sample of a triangle. |
| `TrnglInit` | Initializes a triangle with a given frequency, phase, and magnitude. |

### Table 1-9 Windowing Functions (nsp_UsesWin)

Windowing Functions

| Function Name | Description |
|---|---|
| WinBartlett | Multiplies a vector by a Bartlett windowing function in-place. |
| WinBartlett2 | Multiplies a vector by a Bartlett windowing function not-in-place. |
| WinBlackman | Multiplies a vector by a Blackman windowing function with a user-specified adjustable parameter in-place. |
| WinBlackman2 | Multiplies a vector by a Blackman windowing function with a user-specified adjustable parameter not-in-place. |
| WinBlackmanStd | Multiplies a vector by a Blackman windowing function in-place. |
| WinBlackmanStd2 | Multiplies a vector by a Blackman windowing function not-in-place. |
| WinBlackmanOpt | Multiplies a vector by a Blackman windowing function with a 30-dB roll-off in-place. |
| WinBlackmanOpt2 | Multiplies a vector by a Blackman windowing function with a 30-dB roll-off not-in-place. |
| WinHamming | Multiplies a vector by a Hamming windowing function in-place. |
| WinHamming2 | Multiplies a vector by a Hamming windowing function not-in-place. |
| WinHann | Multiplies a vector by a Hann windowing function in-place. |
| WinHann2 | Multiplies a vector by a Hann windowing function not-in-place. |
| WinKaiser | Multiplies a vector by a Kaiser windowing function in-place. |
| WinKaiser2 | Multiplies a vector by a Kaiser windowing function not-in-place. |

### Table 1-10 Convolution Functions (nsp_UsesConvolution)

Convolution Functions

| Function Name | Description |
|---|---|
| Conv | Performs finite, linear convolution of two sequences. |
| Conv2D | Performs finite, linear convolution of two two-dimensional signals. |
| Filter2D | Filters a two-dimensional signal similar to Conv2D, but with the input and output arrays of the same size. |

### Table 1-11 Discrete Fourier Transform Function (nsp_UsesTransform)

DFT Function

| Function Name | Description |
|---|---|
| Dft | Computes a discrete Fourier transform in-place. |

### Table 1-12 DFT for a Given Frequency (Goertzel) Functions (nsp_UsesTransform)

Goertzel Functions

| Function Name | Description |
|---|---|
| bGoertz | Computes the DFT for a given frequency for a block of successive signal counts. |
| Goertz | Computes the DFT for a given frequency for a single signal count. |
| GoertzInit | Initializes the data used by Goertzel functions. |
| GoertzReset | Resets the internal delay line. |

### Table 1-13 Fast Fourier Transform Functions (nsp_UsesTransform)

FFT Functions

| Function Name | Description |
|---|---|
| Ccs2Fft | Computes a forward or inverse fast Fourier transform of two conjugate-symmetric signals, in-place. The results are stored in RCCcs format. |
| Ccs2FftNip | Computes a forward or inverse fast Fourier transform of two conjugate-symmetric signals, not-in-place. The results are stored in RCCcs format. |
| CcsFft | Computes a forward or inverse fast Fourier transform of a conjugate-symmetric signal, in-place. The results are stored in RCCcs format. |
| CcsFftl | Computes a forward or inverse low-level fast Fourier transform of a conjugate-symmetric signal, in-place. The results are stored in RCPerm or RCPack format. |

continued ☞

**Table 1-13 Fast Fourier Transform Functions (nsp_UsesTransform)** (continued)

| Function Name | Description |
|---|---|
| CcsFftlNip | Computes a forward or inverse low-level fast Fourier transform of a conjugate-symmetric signal, not-in-place. The results are stored in RCPerm or RCPack format. |
| CcsFftNip | Computes a forward or inverse fast Fourier transform of a conjugate-symmetric signal, not-in-place. The results are stored in RCCcs format. |
| Fft | Computes a complex fast Fourier transform in-place. |
| FftNip | Computes a complex fast Fourier transform not-in-place. |
| MpyRCPack2 | Multiplies two vectors stored in RCPack format and stores the results in RCPack format. |
| MpyRCPack3 | Multiplies two vectors stored in RCPack format, and stores the results in a third vector in RCPack format. |
| MpyRCPerm2 | Multiplies two vectors stored in RCPerm format and stores the results in RCPerm format. |
| MpyRCPerm3 | Multiplies two vectors stored in RCPerm format, and stores the results in a third vector in RCPerm format. |
| Real2Fft | Computes a forward or inverse fast Fourier transform of two real signals, in-place. The results are stored in RCCcs format. |
| RealFftNip | Computes a forward or inverse fast Fourier transform of two real signals, not-in-place. The results are stored in RCCcs format. |
| RealFft | Computes a forward or inverse fast Fourier transform of a real signal, in-place. The results are stored in RCCcs format. |
| RealFftl | Computes a forward or inverse low-level fast Fourier transform of a real signal, in-place. The results are stored in RCPerm or RCPack format. |

continued ☛

**Table 1-13 Fast Fourier Transform Functions (nsp_UsesTransform)** (continued)

| Function Name | Description |
|---|---|
| `RealFftlNip` | Computes a forward or inverse low-level fast Fourier transform of a real signal, not-in-place. The results are stored in `RCPerm` or `RCPack` format. |
| `RealFftNip` | Computes a forward or inverse fast Fourier transform of a real signal, not-in-place. The results are stored in `RCCcs` format. |
| `rFft` | Computes a complex fast Fourier transform in-place and places the real and imaginary parts into separate arrays. |
| `rFftNip` | Computes a complex fast Fourier transform not-in-place. On both input and output, the real and imaginary parts are placed in separate arrays. |

**Table 1-14 Low-Level Finite Impulse Response Filter Functions (nsp_UsesFir)**

Low-Level FIR Filter Functions

| Function Name | Description |
|---|---|
| `bFirl` | Filters a block of samples through a low-level, finite impulse response filter. |
| `Firl` | Filters a single sample through a low-level, finite impulse response filter. |
| `FirlGetDlyl` | Gets the delay line values for a low-level, finite impulse response filter. |
| `FirlGetTaps` | Gets the taps coefficients for a low-level, finite impulse response filter. |
| `FirlInit` | Initializes a low-level, single-rate finite impulse response filter. |
| `FirlInitDlyl` | Initializes a delay line for a low-level, finite impulse response filter. |
| `FirlInitMr` | Initializes a low-level, multi-rate finite impulse response filter. |
| `FirlSetDlyl` | Sets the delay line values for a low-level, finite impulse response filter. |
| `FirlSetTaps` | Sets the taps coefficients for a low-level, finite impulse response filter. |

**Table 1-15 Finite Impulse Response Filter Functions (nsp_UsesFir)**

FIR Filter Functions

| Function Name | Description |
|---|---|
| `bFir` | Filters a block of samples through a finite impulse response filter. |
| `Fir` | Filters a single sample through a finite impulse response filter. |
| `FirFree` | Frees dynamic memory associated with finite impulse response filters. |
| `FirGetDlyl` | Gets the delay line values for a finite impulse response filter. |
| `FirGetTaps` | Gets the taps coefficients for a finite impulse response filter. |
| `FirInit` | Initializes a single-rate finite impulse response filter. |
| `FirInitMr` | Initializes a multi-rate finite impulse response filter. |
| `FirSetDlyl` | Sets the delay line values for a finite impulse response filter. |
| `FirSetTaps` | Sets the taps coefficients for a finite impulse response filter. |
| `FirLowpass` | Computes the taps for a lowpass FIR filter. |
| `FirHighpass` | Computes the taps for a highpass FIR filter. |
| `FirBandpass` | Computes the taps for a bandpass FIR filter. |
| `FirBandstop` | Computes the taps for a bandstop FIR filter. |

**Table 1-16 Low-Level Least Mean Squares Adaptation Filter Functions (nsp_UsesLms)**

Low-Level LMS Filter Functions

| Function Name | Description |
|---|---|
| `bLmsl` | Filters samples through a low-level, multi-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `bLmslNa` | Filters samples through a low-level, multi-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm, but without adapting the filter for a secondary signal. |
| `Lmsl` | Filters samples through a low-level, single-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslGetDlyl` | Gets the delay line values for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslGetLeak` | Gets the leak values for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslGetStep` | Gets the step values for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslGetTaps` | Gets the taps coefficients for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslInit` | Initializes a low-level, single-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslInitDlyl` | Initializes a delay line for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslInitMr` | Initializes a low-level, multi-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslNa` | Filters samples through a low-level, single-rate, adaptive FIR filter that uses the LMS algorithm, but without adapting the filter for a secondary signal. |

continued ☞

**Table 1-16 Low-Level Least Mean Squares Adaptation Filter Functions (nsp_UsesLms)** (continued)

| Function Name | Description |
|---|---|
| `LmslSetDlyl` | Sets the delay line values for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslSetLeak` | Sets the leak values for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslSetStep` | Sets the step values for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslSetTaps` | Sets the taps coefficients for a low-level, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |

**Table 1-17 Least Mean Squares Adaptation Filter Functions (nsp_UsesLms)**

LMS Filter Functions

| Function Name | Description |
|---|---|
| `bLms` | Filters samples through a multi-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `bLmsDes` | Filters a block of samples through a single-rate, or multi-rate adaptive FIR filter that uses the least mean squares (LMS) algorithm. The function uses a desired-output signal for adaptation instead of an error signal. |
| `Lms` | Filters a single sample through a single-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmsDes` | Filters a single sample through a single-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. The function uses a desired-output signal for adaptation instead of an error signal. |

continued ☞

**Table 1-17 Least Mean Squares Adaptation Filter Functions (nsp_UsesLms)** (continued)

| Function Name | Description |
|---|---|
| `LmsFree` | Frees dynamic memory associated with an adaptive FIR filter that uses the LMS algorithm. |
| `LmsGetDlyl` | Gets the delay line values for an adaptive FIR filter that uses the LMS algorithm. |
| `LmsGetErrVal` | Gets the error signal for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. The error signal must be computed from the desired signal by the Signal Processing Library. |
| `LmsGetLeak` | Gets the leak values for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmsGetStep` | Gets the step values for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmsGetTaps` | Gets the taps coefficients for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmsInit` | Initializes a single-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmsInitMr` | Initializes a multi-rate, adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmsSetDlyl` | Sets the delay line values for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmsSetErrVal` | Sets the error signal for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. The error signal must be computed from the desired signal by the Signal Processing Library. |
| `LmsSetLeak` | Sets the leak values for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmsSetStep` | Sets the step values for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. |
| `LmslSetTaps` | Sets the taps coefficients for an adaptive FIR filter that uses the least mean squares (LMS) algorithm. |

### Table 1-18 Low-Level Infinite Impulse Response Filter Functions (nsp_UsesIir)

Low-Level IIR Filter Functions

| Function Name | Description |
|---|---|
| `bIirl` | Filters a block of samples through a low-level, infinite impulse response filter. |
| `Iirl` | Filters a single sample through a low-level, infinite impulse response filter. |
| `IirlInit` | Initializes a low-level, infinite impulse response filter of a specified order. |
| `IirlInitGain` | Initializes an integer flavor of a low-level IIR filter for an input signal of a limited bit range. |
| `IirlInitBq` | Initializes a low-level, infinite impulse response (IIR) filter to reference a cascade of biquads (second-order IIR sections). |
| `IirlInitDlyl` | Initializes the delay line for a low-level, infinite impulse response (IIR) filter. |

### Table 1-19 Infinite Impulse Response Filter Functions (nsp_UsesIir)

IIR Filter Functions

| Function Name | Description |
|---|---|
| `bIir` | Filters a block of samples through an infinite impulse response filter. |
| `Iir` | Filters a single sample through an infinite impulse response filter. |
| `IirFree` | Frees dynamically allocated memory associated with an infinite impulse response filter. |
| `IirInit` | Initializes an infinite impulse response filter of a specified order. |
| `IirInitBq` | Initializes an infinite impulse response (IIR) filter to reference a cascade of biquads (second-order IIR sections). |

### Table 1-20 Median Filter Functions (nsp_UsesMedian)

Median Filter Functions

| Function Name | Description |
|---|---|
| `bMedianFilter1` | Computes median values for each input vector element in-place. |
| `bMedianFilter2` | Computes median values for each input vector element and writes the result to the output vector. |

### Table 1-21 Library Information Function (nsp_UsesLibVersion)

Library Version Function

| Function Name | Description |
|---|---|
| `GetLibVersion` | Returns information about the Signal Processing Library version. |

### Table 1-22 Memory Reclaim Functions (nsp_UsesTransform)

Memory Reclaim Functions

| Function Name | Description |
|---|---|
| `FreeBitRevTbls` | Frees dynamic memory for tables of bit-reversed indices. |
| `FreeTwdTbls` | Frees memory associated with all twiddle tables of a particular type. |

### Table 1-23 Wavelet Functions (nsp_UsesWavelet)

Wavelet Functions

| Function Name | Description |
|---|---|
| `WtDecompose` | Decomposes signals into wavelet series. |
| `WtFree` | Frees the memory used by the wavelet functions for internal purposes. |
| `WtGetState` | Returns wavelet parameters of the NSPWtState structure. |
| `WtInit`, `WtInitLen`, `WtInitUserFilter` | Initialize the NSPWtState structure. |
| `WtReconstruct` | Reconstructs signals from wavelet decomposition. |
| `WtSetState` | Sets wavelet parameters of the NSPWtState structure. |

### Table 1-24 Discrete Cosine Transform Function (nsp_UsesTransform)

DCT Function

| Function Name | Description |
|---|---|
| `Dct` | Performs the discrete cosine transform (DCT). |

# *Error Handling* 2

Library function lists

This chapter describes the error handling facility supplied with the Intel® Signal Processing Library. The library functions report a variety of errors including bad arguments (`NULL` pointers and out-of-range parameters) and out of memory conditions. When a function detects an error, instead of returning a status code, the function signals an error by calling `nspSetErrStatus()`. This allows the error handling mechanism to be handled separately from the normal flow of the signal processing code. The signal processing code is thus cleaner and more compact as shown in this example.

```
outputSample = nspdFir(&firSt, inputSample);
if(nspGetErrStatus()<0)
// do error checking
```

The error handling system is hidden within the function `nspdFir()`. Thus, this statement is uncluttered by error handling code and results in a statement which closely resembles a mathematical formula.

The errors that a function may signal are implementation-dependent. Your application should assume that every library function call may result in some error condition. The Signal Processing Library performs extensive error checks (for example, `NULL` pointers, out-of-range parameters, corrupted states) for every library function.

Error macros are provided to simplify the coding for error checking and reporting. You can modify the way your application handles errors by calling `nspRedirectError()` with a pointer to your own error handling function. For more information, see "Adding Your Own Error Handler"

later in this chapter. For even more flexibility, you can replace the whole error handling facility with your own code. The source code of the default error handling facility is provided.

The Signal Processing Library does not process numerical exceptions (for example, overflow, underflow, and division by zero). The underlying floating point library or processor has the responsibility for catching and reporting these exceptions. A floating-point library is needed if a processor that handles floating-point is not present. You can attach an exception handler using an underlying floating-point library for your application, if your system supports such a library.

## Error Functions

The following sections describe the error-handling functions in the Signal Processing Library.

## Error

*Performs basic error handling.*

```
void nspError(NSPStatus status, const char *func,
      const char *context);
```

### Discussion

The `nspError()` function should be called whenever any of the library's functions encounters an error. The actual error reporting will be handled differently, depending on whether the program is running in Windows mode or in console mode. Within each invocation mode, you can set the error mode flag to alter the behavior of the `nspError()` function. See page 2-4, "SetErrMode," (for `nspSetErrMode()`) for more information on the defined error modes.

To simplify the coding for error checking and reporting, the error handling system supplied by the Signal Processing Library supports a set of error macros. See “Error Macros” for a detailed description of the error handling macros.

The `nspError()` function calls the default error reporting function. You can change the default error reporting function by calling `nspRedirectError()`. For more information, see page 2-6, “RedirectError,” (for `nspRedirectError()`).

## GetErrStatus, SetErrStatus

*Gets and sets the error codes which describe the type of error being reported.*

```
typedef int NSPStatus;
NSPStatus nspGetErrStatus();
void nspSetErrStatus(NSPStatus status);
```

| | |
|---|---|
| *`status`* | Code that indicates the type of error (see Table 2-1, “nspError() Status Codes”). |

### Discussion

The `nspGetErrStatus()` and `nspSetErrStatus()` functions get and set the error status codes which describe the type of error being reported. See “Status Codes” for descriptions of each of the error status codes.

# GetErrMode
# SetErrMode

*Gets and sets the error modes which describe how an error is processed.*

```
#define NSP_ErrModeLeaf   0
#define NSP_ErrModeParent 1
#define NSP_ErrModeSilent 2
int nspGetErrMode();
void nspSetErrMode(int errMode);
```

| | |
|---|---|
| *errMode* | Indicates how errors will be processed. The possible values for *errMode* are `NSP_ErrModeLeaf`, `NSP_ErrModeParent`, or `NSP_ErrModeSilent`. |

## Discussion

**NOTE.** *This section describes how the default error handler handles errors for applications which run in console mode. If your application has a custom error handler, errors will be processed differently than described below.*

The `nspSetErrMode()` function sets the error modes which describe how errors are processed. The defined error modes are `NSP_ErrModeLeaf`, `NSP_ErrModeParent`, and `NSP_ErrModeSilent`.

If you specify `NSP_ErrModeLeaf`, errors are processed in the “leaves” of the function call tree. The `nspError()` function (in console mode) prints an error message describing *status*, *func*, and *context*. It then terminates the program.

If you specify `NSP_ErrModeParent`, errors are processed in the “parents” of the function call tree. When `nspError()` is called as the result of detecting an error, an error message will print but the program will not terminate. Each time a function calls another function, it must check to see if an error has occurred. When an error occurs, the function should call `nspError()` specifying `NSP_StsBackTrace`, and then return. The macro `NSP_ERRCHK()` may be used to perform both the error check and back trace call. This passes the error "up" the function call tree until eventually some parent function (possibly `main()`) detects the error and terminates the program.

`NSP_ErrModeSilent` is similar to `NSP_ErrModeParent`, except that error messages are not printed.

`NSP_ErrModeLeaf` is the default, and is the simplest method of processing errors. `NSP_ErrModeParent` requires more programming effort, but provides more detailed information about where and why an error occurred. All of the functions in the library support both options (that is, they use `NSP_ERRCHK()` after function calls). If an application uses the `NSP_ErrModeParent` option, it is essential that it check for errors after all library functions that it calls.

The status code of the last detected error is stored into the global variable `NspLastStatus` and can be returned by calling `nspGetErrStatus()`. The value of this variable may be used by the application during the back trace process to determine what type of error initiated the back trace.

## ErrorStr

*Translates an error or status code into a textual description.*

```
const char* nspErrorStr(NSPStatus status);
```

| | |
|---|---|
| *status* | Code that indicates the type of error (see Table 2-1, "nspError() Status Codes"). |

### Discussion

The function `nspErrorStr()` returns a short string describing *`status`*. Use this function to produce error messages for users. The returned pointer is a pointer to an internal static buffer that may be over-written on the next call to `nspErrorStr()`.

## RedirectError

*Assigns a new error handler to call when an error occurs.*

```
NSPErrorCallBack nspRedirectError(NSPErrorCallBack func);
```

| | |
|---|---|
| *`func`* | Pointer to the function that will be called when an error occurs. |

### Discussion

The `nspRedirectError()` function assigns a new function to be called when an error occurs in the SP Library. If *`func`* is `NULL`, `nspRedirectError()`installs the Signal Processing Library's default error handler.

The return value of `nspRedirectError()` is a pointer to the previously assigned error handling function.

For the definition of the function typedef `NSPErrorCallBack`, see the include file `nsperror.h`. See "Adding Your Own Error Handler" for more information on the `nspRedirectError()` function.

## Error Macros

The error macros associated with the `nspError()` function are described below.

```
#define NSP_ERROR(status,func,context) \
      nspError((status),(func),(context));
```

```
#define NSP_ERRCHK(func,context)\
      ( (nspGetErrStatus()>=0) ? NSP_StsOk \
            : NSP_ERROR(NSP_StsBackTrace,(func),(context)) )
#define NSP_ASSERT(expr,func,context)\
      ( (expr) ? NSP_StsOk\
            : NSP_ERROR(NSP_StsInternal,(func),(context)) )
#define NSP_RSTERR()    (nspSetErrStatus(NSP_StsOk))
```

| | |
|---|---|
| *context* | Provides additional information about the context in which the error occurred. If the value of *context* is NULL or empty, this string will not appear in the error message. |
| *expr* | An expression that checks for an error condition and returns FALSE if an error occurred. |
| *func* | Name of the function where the error occurred. |
| *status* | Code that indicates the type of error (see Table 2-1, "nspError() Status Codes." |

### Discussion

The NSP_ASSERT() macro checks for the error condition *expr* and sets the error status NSP_StsInternal if the error occurred.

The NSP_ERRCHK() macro checks to see if an error has occurred by checking the error status. If an error has occurred, NSP_ERRCHK() creates an error back trace message and returns a non-zero value. This macro should normally be used after any call to a function that might have signaled an error.

The NSP_ERROR() macro simply calls the nspError() function by default. This macro is used by other error macros. By changing NSP_ERROR() you can modify the error reporting behavior without changing a single line of source code.

The NSP_RSTERR() macro resets the error status to NSP_StsOk, thus clearing any error condition. This macro should be used by an application when it decides to ignore an error condition.

## Status Codes

The status codes used by the Signal Processing Library are described in Table 2-1. Status codes are integers, not an enumerated type. This allows an application to extend the set of status codes beyond those used by the library itself. Negative codes indicate errors, while non-negative codes indicate success.

**Table 2-1 nspError() Status Codes**

| Status Code | Value | Description |
|---|---|---|
| `NSP_StsOk` | 0 | No error. The `nspError()` function will do nothing if called with this status code. |
| `NSP_StsBackTrace` | -1 | Implements a backtrace of the function calls that lead to an error. If `NSP_ERRCHK()` detects that a function call resulted in an error, it calls `NSP_ERROR()` with this status code to provide further context information for the user. |
| `NSP_StsError` | -2 | An error of unknown origin, or of an origin not correctly described by the other error codes. |
| `NSP_StsInternal` | -3 | An internal “consistency” error, often the result of a corrupted state structure. These errors are typically the result of a failed assertion. |
| `NSP_StsNoMem` | -4 | A function attempted to allocate memory using `malloc()` or a related function and was unsuccessful. The message *`context`* indicates the intended use of the memory. |
| `NSP_StsBadArg` | -5 | One of the arguments passed to the function is invalid. The message *`context`* indicates which argument and why. |

continued ☞

**Table 2-1 nspError() Status Codes** (continued)

| Status Code | Value | Description |
|---|---|---|
| NSP_StsBadFunc | -6 | The function is not supported by the implementation, or the particular operation implied by the given arguments is not supported. |
| NSP_StsNoConv | -7 | An iterative convergence algorithm failed to converge within a reasonable number of iterations. |

**Application Notes:** The global variable NspLastStatus records the status of the last error reported. Its value is initially NSP_StsOk. The value of NspLastStatus is not explicitly set by the library function detecting an error. Instead, it is set by nspSetErrStatus().

If the application decides to ignore an error, it should reset NspLastStatus back to NSP_StsOk (see NSP_RSTERR() under "Error Macros"). An application-supplied error handling function must update NspLastStatus correctly; otherwise the Signal Processing Library might fail. This is because the macro NSP_ERRCHK(), which is used internally to the library, refers to the value of this variable.

## Error Handling Example

The following example describes the default error handling for a console application. In the example program, test.c, assume that the function libFuncB() represents a library function such as nsp?Fft(), and the function libFuncD() represents a function that is called internally to the library such as nsp?GetFftTwdTbl(). In this scenario, main() and appFuncA() represent application code.

The value of the error mode is set to NSP_ErrModeParent. The NSP_ErrModeParent option produces a more detailed account of the error conditions.

### Example 2-1 Error Functions

```
/* application main function */
main() {
      nspSetErrMode(NSP_ErrModeParent);
      appFuncA(5, 45, 1.0);
      if (NSP_ERRCHK("main","compute something"))
            exit(1);
      return 0;
}

/* application subroutine */
void appFuncA(int order1, int order2, double a) {
      libFuncB(a, order1);
      if (NSP_ERRCHK("appFuncA","compute using order1")) return;
      libFuncB(a, order2);
      if (NSP_ERRCHK("appFuncA","compute using order2"))  return;
      /* do some more work */
}
```

continued ☞

**Example 2-1 Error Functions** (continued)

```
/* library function (e.g., nsp?Fft()) */
void libFuncB(double a, int order) {
      double *vec;

      if (order > 31) {
            NSP_ERROR(NSP_StsBadArg, "libFuncB",
            "order must be less than or equal to 31");
            return;
      }
      if ((vec = libFuncD(a, order)) == NULL) {
            NSP_ERRCHK("libFuncB", "compute using a");
            return;
      }
      /* code to do some real work goes here */
      free(vec);
}
/* library function called internally (e.g.,nsp?GetFftTwdTbl()) */
double *libFuncD(double a, int order) {
      double *vec;

      if ((vec=(double*)malloc(order*sizeof(double))) == NULL) {
            NSP_ERROR(NSP_StsNoMem, "libFuncD",
            "allocating a vector of doubles");
            return NULL;
      }

      /* do something with vec */
      return vec;
}
```

When the program is run, it produces the output illustrated in Example 2-2.

**Example 2-2 Output for the Error Function Program (NSP_ErrModeParent)**

```
Intel Signal Processing Library Error: Invalid argument in function
libFuncB: order must be less than or equal to 31
      called from function appFuncA: compute using order2
      called from function main: compute something
```

If the program had run with the `NSP_ErrModeLeaf` option instead of `NSP_ErrModeParent`, only the first line of the above output would have been produced before the program terminated.

If the program in Example 2-1 had run out of heap memory while using the `NSP_ErrModeParent` option, then the output illustrated in Example 2-3 would be produced.

**Example 2-3 Output for the Error Function Program (NSP_ErrModeParent)**

```
Intel Signal Processing Library Error: Out of memory in function
libFuncD:
allocating a vector of doubles
      called from function libFuncB: compute using a
      called from function appFuncA: compute using order1
      called from function main[]: compute something
```

Again, if the program had been run with the `NSP_ErrModeLeaf` option instead of `NSP_ErrModeParent`, only the first line would have been produced.

## Adding Your Own Error Handler

The Signal Processing Library allows you to define your own error handler. User-defined error handlers are useful if you want your application to send error messages to a destination other than the standard error output stream. For example, you can choose to send error messages to a dialog box if your application is running under a Windows system or you can choose to send error messages to a special log file.

There are two methods of adding your own error handler. In the first method, you can replace the `nspError()` function or the complete error handling library with your own code. Note that this method can only be used at link time.

In the second method, you can use the `nspRedirectError()` function to replace the error handler at run time. The steps below describe how to create your own error handler and how to use the `nspRedirectError()` function to redirect error reporting.

1. Define a function with the function prototype, `NSPErrorCallBack`, as defined by the Signal Procesing Library.
2. Your application should then call the `nspRedirectError()` function to redirect error reporting for your own function. All subsequent calls to `nspError()` will call your own error handler.
3. To redirect the error handling back to the default handler, simply call `nspRedirectError()` with a `NULL` pointer.

Example 2-4 illustrates a user-defined error handler function, *`ownError()`*, which simply prints an error message constructed from its arguments and exits.

**Example 2-4 A Simple Error Handler**

```
NSPStatus ownError(NSPStatus status, const char *func,
 const char *context, const char *file, int line)
{
 fprintf(stderr, "SP Library error: %s, ", nspErrorStr(status));
 fprintf(stderr, "function %s, ", func ? func : "<unknown>");
 if (line > 0) fprintf(stderr, "line %d, ", line);
 if (file != NULL) fprintf(stderr, "file %s, ", file);
 if (context) fprintf(stderr, "context %s\n", context);
 NspSetErrStatus(status);
 exit(1);
}

main ()
{
    extern NSPErrorCallBack ownError;

    /* Redirect errors to your own error handler */
    nspRedirectError(ownError);

    /* Redirect errors back to the default error handler */
    nspRedirectError(NULL);
}
```

# Arithmetic and Vector Manipulation Functions

# 3

The functions described in this chapter perform memory allocation operations, complex-valued arithmetic, vector initialization, vector arithmetic, and the following vector manipulation functions: measure, conjugation, resampling, and correlation.

## Memory Allocation Functions

Library function lists

This section describes the Signal Processing Library 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 items allocated, *length*.

## Malloc

*Allocates a 32-byte aligned memory block for data of different types.*

```
void* nspMalloc (int length);
float* nspsMalloc(int length);
      /* real values; single precision */
double* nspdMalloc(int length);
      /* real values; double precision */
SCplx* nspcMalloc(int length);
      /* complex values; single precision */
```

```
DCplx* nspzMalloc(int length);
      /* complex values; double precision */
short* nspwMalloc(int length);
      /* real values; short integer */
WCplx* nspvMalloc(int length);
      /* complex values; short integer */
int* nspiMalloc(int length);
      /* real values; integer */
ICplx* nspjMalloc(int length);
      /* complex values; integer */
```

| | |
|---|---|
| `length` | Number of data items to allocate. |

### Discussion

The `nsp?Malloc()` functions allocate memory block aligned to a 32-byte boundary for a required number of data items. The "`?`" placeholder in the function name indicates the type of data items for which the memory allocation is performed. It can be either `s`, `c`, `d`, `z`, `w`, `v`, or `i` (integer real), and `j` (integer complex values).

The functions return a pointer to an aligned memory block. If no memory is available in the system, then the `NULL` pointer is returned.

To free this memory, use `nspFree()`.

## Free

*Frees a memory block previously allocated by one of the `nsp?Malloc` functions*

```
void nspFree( void* ptr );
```

| | |
|---|---|
| `ptr` | The pointer to a memory block to be freed. |

### Discussion

The `nspFree()` function deallocates the aligned memory block that was previously allocated by one of the `nsp?Malloc()` functions.

# Arithmetic Functions

This section describes the Signal Processing Library functions that perform complex-valued arithmetic.

## Set

*Initializes a complex value to a specified value.*

```
SCplx nspcSet(const float re, const float im);
      /* complex values; single precision */
DCplx nspzSet(const double re, const double im);
      /* complex values; double precision */
WCplx nspvSet(const short int re, const short int im);
      /* complex values; short integer */
```

| | |
|---|---|
| `re` | Real part of the complex value. |
| `im` | Imaginary part of the complex value. |

### Discussion

The function `nsp?Set()` initializes a complex value with (*re*, *im*).

## Add

*Adds two complex values.*

```
SCplx nspcAdd(const SCplx a, const SCplx b);
      /* complex values; single precision */
DCplx nspzAdd(const DCplx a, const DCplx b);
      /* complex values; double precision */
```

```
WCplx nspvAdd(const WCplx a, const WCplx b, int ScaleMode,
      int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| `a`, `b` | Complex values to be added. |
| `ScaleMode`, `ScaleFactor` | Refer to “Scaling Arguments” in Chapter 1. |

### Discussion

The `nsp?Add()` function adds two complex values (`a` + `b`).

## Conj

*Conjugates a complex value.*

```
SCplx nspcConj(const SCplx a);
      /* complex values; single precision */
DCplx nspzConj(const DCplx a);
      /* complex values; double precision */
WCplx nspvConj(const WCplx a);
      /* complex values; short integer */
```

### Discussion

The `nsp?Conj()` function conjugates a complex value (`a`*).

## Div

*Divides two complex values.*

```
SCplx nspcDiv(const SCplx a, const SCplx b);
      /* complex values; single precision */
```

```
DCplx nspzDiv(const DCplx a, const DCplx b);
      /* complex values; double precision */
WCplx nspvDiv(const WCplx a, const WCplx b);
      /* complex values; short integer */
```

| | |
|---|---|
| *a*, *b* | Complex values: *a* is a dividend, *b* is a divisor. |

### Discussion

The `nsp?Div()` function divides two complex values (*a* / *b*).

## Mpy

*Multiplies two complex values.*

```
SCplx nspcMpy(const SCplx a, const SCplx b);
      /* complex values; single precision */
DCplx nspzMpy(const DCplx a, const DCplx b);
      /* complex values; double precision */
WCplx nspvMpy(const WCplx a, const WCplx b, int ScaleMode,
      int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *a*, *b* | Complex values to be multiplied. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?Mpy()` function multiplies two complex values (*a* * *b*).

## Sub

*Subtracts two complex values.*

```
SCplx nspcSub(const SCplx a, const SCplx b);
      /* complex values; single precision */
DCplx nspzSub(const DCplx a, const DCplx b);
      /* complex values; double precision */
WCplx nspvSub(const WCplx a, const WCplx b, int ScaleMode,
      int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| `a`, `b` | Complex values: `a` is a minuend, `b` is a subtrahend. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?Sub()` function subtracts two complex values (`a` - `b`).

## Vector Initialization Functions

The functions described in this section initialize the values of the elements of a vector. A vector's elements can be initialized to zero or to another specified value. They can also be initialized to the value of a second vector.

## bCopy

*Initializes a vector with the contents of a second vector.*

```
void nspsbCopy(const float *src, float *dst, int n);
      /* real values; single precision */
```

```
void nspcbCopy(const SCplx *src, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbCopy(const double *src, double *dst, int n);
      /* real values; double precision */
void nspzbCopy(const DCplx *src, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbCopy(const short *src, short *dst, int n);
      /* real values; short integer */
void nspvbCopy(const WCplx *src, WCplx *dst, int n);
      /* complex values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector to be initialized. |
| *n* | The number of elements to copy. |
| *src* | Pointer to the source vector used to initialize *dst[n]*. |

### Discussion

The function `nsp?bCopy()` copies the first *n* elements from a source vector *src[n]* into a destination vector *dst[n]*.

## bSet

*Initializes a vector to a specified value.*

```
void nspsbSet(float val, float *dst, int n);
      /* real values; single precision */
void nspcbSet(float re, float im, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbSet(double val, double *dst, int n);
      /* real values; double precision */
void nspzbSet(double re, double im, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbSet(short val, short *dst, int n);
      /* real values; short integer */
void nspvbSet(short re, short im, WCplx *dst, int n);
      /* complex values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector to be initialized. |
| *n* | The number of elements to initialize. |
| *re*, *im* | The complex value (*re* + j*im*) used to initialize the vector *dst[n]*. |
| *val* | The real value used to initialize the vector *dst[n]*. |

### Discussion

The function `nsp?bSet()` initializes the first *n* elements of the vector *dst[n]* to contain the same value: either *val* (if *dst[n]* is a real vector) or *re* + j*im* (if *dst[n]* is a complex vector).

## bZero

*Initializes a vector to zero.*

```
void nspsbZero(float *dst, int n);
      /* real values; single precision */
void nspcbZero(SCplx *dst, int n);
      /* complex values; single precision */
void nspdbZero(double *dst, int n);
      /* real values; double precision */
void nspzbZero(DCplx *dst, int n);
      /* complex values; double precision */
void nspwbZero(short *dst, int n);
      /* real values; short integer */
void nspvbZero(WCplx *dst, int n);
      /* complex values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector to be initialized to zero. |
| *n* | The number of elements to initialize. |

### Discussion

The `nsp?bZero()` function initializes the first *n* elements of the vector *dst[n]* to 0.

## Vector Arithmetic Functions

This section describes the Signal Processing Library functions which perform vector arithmetic operations between the vectors. The arithmetic functions include basic, element-wise arithmetic operations between vectors as well as more complex calculations such as limiting vector elements by a specified threshold or computing absolute values, square and square root, natural logarithm and exponential of vector elements.

The library provides two versions of each function. One version performs the operation "in-place," while the other stores the results of the operation in a third vector.

## bAdd1

*Adds a value to each element of a vector.*

```
void nspsbAdd1(const float val, float *dst, int n);
      /* real values; single precision */
void nspcbAdd1(const SCplx val, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbAdd1(const double val, double *dst, int n);
      /* real values; double precision */
void nspzbAdd1(const DCplx val, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbAdd1(const short val, short *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbAdd1(const WCplx val, WCplx *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *val* | The value used to increment each element of the vector *dst[n]*. |
| *dst* | Pointer to the vector *dst[n]*. |
| *n* | The number of values in the vector dst. |
| *ScaleMode*, *ScaleFactor* | Refer to “Scaling Arguments” in Chapter 1. |

### Discussion

The nsp?bAdd1() function adds a value *val* to each element of a destination vector *dst[n]* in-place.

## bAdd2

*Adds the elements of two vectors.*

```
void nspsbAdd2(const float *src, float *dst, int n);
      /* real values; single precision */
void nspcbAdd2(const SCplx *src, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbAdd2(const double *src, double *dst, int n);
      /* real values; double precision */
void nspzbAdd2(const DCplx *src, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbAdd2(const short *src, short *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbAdd2(const WCplx *src, WCplx *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector *dst[n]*. The vector *dst[n]* stores the result of the addition *src[n]* + *dst[n]*. |
| *n* | The number of values in the vectors. |

| | |
|---|---|
| *src* | Pointer to the vector to be added to *dst[n]*. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bAdd2()` function adds the elements of a source vector *src[n]* to the elements of a destination vector *dst[n]*, and stores the result in *dst[n]*. The vectors *src[n]* and *dst[n]* must be of equal length. If they are not, the function will return unpredictable results.

## bAdd3

*Adds the elements of two vectors and stores the result in a third vector.*

```
void nspsbAdd3(const float *srcA, const float *srcB, float *dst,
      int n); /* real values; single precision */
void nspcbAdd3(const SCplx *srcA, const SCplx *srcB, SCplx *dst,
      int n); /* complex values; single precision */
void nspdbAdd3(const double *srcA, const double *srcB, double *dst,
      int n); /* real values; double precision */
void nspzbAdd3(const DCplx *srcA, const DCplx *srcB, DCplx *dst,
      int n); /* complex values; double precision */
void nspwbAdd3(const short *srcA, const short *srcB, short *dst,
      int n, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbAdd3(const WCplx *srcA, const WCplx *srcB, WCplx *dst,
      int n, int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector *dst[n]*. This vector stores the result of the addition *srcA[n]* + *srcB[n]*. |
| *n* | The number of values in the vectors. |

| | |
|---|---|
| *srcA, srcB* | Pointers to the vectors whose elements are to be added together. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bAdd3()` function adds the elements of a source vector *srcA[n]* to the elements of vector *srcB[n]*, and stores the result in *dst[n]*. The vectors *srcA[n]*, *srcB[n]*, and *dst[n]* must be of equal length. If they are not, the function will return unpredictable results.

## bMpy1

*Multiplies each element of a vector by a value.*

```
void nspsbMpy1(const float val, float *dst, int n);
      /* real values; single precision */
void nspcbMpy1(const SCplx val, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbMpy1(const double val, double *dst, int n);
      /* real values; double precision */
void nspzbMpy1(const DCplx val, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbMpy1(const short val, short *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbMpy1(const WCplx val, WCplx *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *val* | The value used to multiply each element of the vector *dst[n]*. |
| *dst* | Pointer to the vector *dst[n]*. |

| | |
|---|---|
| *n* | The number of values in the vector `dst`. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The function `nsp?bMpy1()` multiplies each element of the destination vector `dst[n]` by the value `val` in-place.

## bMpy2

*Multiplies the elements of two vectors.*

```
void nspsbMpy2(const float *src, float *dst, int n);
      /* real values; single precision */
void nspcbMpy2(const SCplx *src, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbMpy2(const double *src, double *dst, int n);
      /* real values; double precision */
void nspzbMpy2(const DCplx *src, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbMpy2(const short *src, short *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbMpy2(const WCplx *src, WCplx *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector `dst[n]`. This vector stores the result of the multiplication (`src[n]` * `dst[n]`). |
| *n* | The number of values in the vectors. |
| *src* | Pointer to the vector to be multiplied with `dst[n]`. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The function nsp?bMpy2() multiplies the elements of the vector *src[n]* by the elements of the vector *dst[n]*, and stores the result in *dst[n]*. The vectors *src[n]* and *dst[n]* must be of equal length. If they are not, the function will return unpredictable results.

## bMpy3

*Multiplies two vectors and stores the result in a third vector.*

```
void nspsbMpy3(const float *srcA, const float *srcB, float *dst,
      int n);   /* real values; single precision */
void nspcbMpy3(const SCplx *srcA, const SCplx *srcB, SCplx *dst,
      int n);   /* complex values; single precision */
void nspdbMpy3(const double *srcA, const double *srcB, double *dst,
      int n);   /* real values; double precision */
void nspzbMpy3(const DCplx *srcA, const DCplx *srcB, DCplx *dst,
      int n);   /* complex values; double precision */
void nspwbMpy3(const short *srcA, const short *srcB, short *dst,
      int n, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbMpy3(const WCplx *srcA, const WCplx *srcB, WCplx *dst,
      int n, int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| dst | Pointer to the vector *dst[n]*. This vector stores the result of the multiplication (*srcA[n]* * *srcB[n]*). |
| *n* | The number of values in the vectors. |
| *srcA, srcB* | Pointers to the vectors whose elements are to be multiplied together. |
| *ScaleMode*, *ScaleFactor* | Refer to “Scaling Arguments” in Chapter 1. |

### Discussion

The `nsp?bMpy3()` function multiplies the elements of a vector *srcA[n]* to the elements of a vector *srcB[n]*, and stores the result in *dst[n]*. The vectors *srcA[n]*, *srcB[n]*, and *dst[n]* must be of equal length. If they are not, the function will return unpredictable results.

## bSub1

*Subtracts a value from each element of a vector.*

```
void nspsbSub1(const float val, float *dst, int n);
      /* real values; single precision */
void nspcbSub1(const SCplx val, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbSub1(const double val, double *dst, int n);
      /* real values; double precision */
void nspzbSub1(const DCplx val, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbSub1(const short val, short *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbSub1(const WCplx val, WCplx *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *val* | The value used to decrement each element of the vector *dst[n]*. |
| *dst* | Pointer to the vector *dst[n]*. |
| *n* | The number of values in the vector *dst*. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bSub1()` function subtracts a value *val* from each element of a destination vector *dst[n]* in-place.

## bSub2

*Subtracts the elements of two vectors.*

```
void nspsbSub2(const float *val, float *dst, int n);
      /* real values; single precision */
void nspcbSub2(const SCplx *val, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbSub2(const double *val, double *dst, int n);
      /* real values; double precision */
void nspzbSub2(const DCplx *val, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbSub2(const short *val, short *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbSub2(const WCplx *val, WCplx *dst, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *val* | Pointer to the vector to be subtracted from *dst[n]*. |
| *dst* | Pointer to the vector *dst[n]*. The vector *dst[n]* stores the result of the subtraction *dst[n]* - *val[n]*. |
| *n* | The number of values in the vectors. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bSub2()` function subtracts the elements of a vector *val[n]* from the elements of a destination vector *dst[n]*, and stores the result in *dst[n]*. The vectors *val[n]* and *dst[n]* must be of equal length. If they are not, the function will return unpredictable results.

# bSub3

*Subtracts the elements of two vectors and stores the results in a third vector.*

```
void nspsbSub3(const float *src, const float *val, float *dst,
      int n);      /* real values; single precision */
void nspcbSub3(const SCplx *src, const SCplx *val, SCplx *dst,
      int n);      /* complex values; single precision */
void nspdbSub3(const double *src, const double *val, double *dst,
      int n);   /* real values; double precision */
void nspzbSub3(const DCplx *src, const DCplx *val, DCplx *dst,
      int n);      /* complex values; double precision */
void nspwbSub3(const short *src, const short *val, short *dst,
      int n, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbSub3(const WCplx *src, const WCplx *val, WCplx *dst,
      int n, int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector whose elements are to be decreased by the elements of *val[n]*. |
| *val* | Pointer to the vector whose elements are subtracted from *src[n]*. |
| *dst* | Pointer to the vector *dst[n]*. This vector stores the result of the subtraction *src[n]* – *val[n]*. |
| *n* | The number of values in the vectors. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The nsp?bSub3() function subtracts the elements of a vector *val[n]* from the elements of a source vector *src[n]*, and stores the result in *dst[n]*. The vectors *src[n]*, *val[n]*, and *dst[n]* must be of equal length. If they are not, the function will return unpredictable results.

## bNormalize

*Subtracts a constant from vector elements and divides the result by another constant.*

```
void nspsbNormalize(const float *src, float *dst, int n, float
      offset, float factor); /* real values; single precision */
void nspcbNormalize(const SCplxt *src, SCplx *dst, int n, SCplx
      offset, float factor); /* complex values; single precision */
void nspdbNormalize(const double *src, double *dst, int n, double
      offset, float factor); /* real values; double precision */
void nspzbNormalize(const DCplx *src, DCplx *dst, int n, DCplx
      offset, float factor); /* complex values; double precision */
void nspwbNormalize(const short *src, short *dst, int n, short
      offset, float factor); /* real values; short */
void nspvbNormalize(const WCplxt *src, WCplxt *dst, int n,
      WCplx offset, float factor); /* complex values; short */
```

| | |
|---|---|
| *src* | Pointer to the input vector *a[n]*. |
| *dst* | Pointer to the output vector *b[n]*. |
| *n* | The number of elements in each of these vectors. |
| *offset* | The constant subtracted from input vector elements. |
| *factor* | The constant by which the vector elements are divided. |

### Discussion

The `nsp?bNormalize()` function subtracts the *`offset`* constant from the elements of the input vector *`a`* and divides the result by *`factor`*.

The function returns a vector *`b[n]`* with the elements

*`b[k]`* = (*`a[k]`* - *`offset`*)/*`factor`*.

## DotProd

*Computes the dot product of two vectors.*

```
float nspsDotProd(const float *vec1, const float *vec2, int n);
      /* real values; single precision */
double nspdDotProd(const double *vec1, const double *vec2, int n);
      /* real values; double precision */
SCplx nspcDotProd(const SCplx *vec1, const Scplx *vec2, int n);
      /* complex values; single precision */
DCplx nspzDotProd(const DCplx *vec1, const Dcplx *vec2, int n);
      /* complex values; double precision */
SCplx nspscDotProd(const float *vec1, const Scplx *vec2, int n);
      /* real and complex values; single precision */
SCplx nspcsDotProd(const SCplx *vec1, const float *vec2, int n);
      /* complex and real values; single precision */
DCplx nspdzDotProd(const double *vec1, const Dcplx *vec2, int n);
      /* real and complex values; double precision */
DCplx nspzdDotProd(const DCplx *vec1, const double *vec2, int n);
      /* complex and real values; double precision */
short nspwDotProd(const short *vec1, const short *vec2, int n,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
WCplx nspvDotProd(const WCplx *vec1, const WCplx *vec2, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
WCplx nspwvDotProd(const short *vec1, const WCplx *vec2, int n,
      int ScaleMode, int *ScaleFactor);
      /* real and complex values; short integer */
```

```
WCplx nspvwDotProd(const WCplx *vec1, const short *vec2, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex and real values; short integer */
```

| | |
|---|---|
| *vec1* | Pointer to the first vector to compute the dot product of two vectors. |
| *vec2* | Pointer to the second vector to compute the dot product of two vectors. |
| *n* | The number of elements in the vectors. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?DotProd()` function computes the dot product (scalar value) of two vectors, *vec1[n]* and *vec2[n]*. When computing the dot product, complex flavors of the `nsp?DotProd()` function do not conjugate the second operand. The vectors *vec1[n]* and *vec2[n]* must be of equal length. If they are not, the function will return unpredictable results.

## DotProdExt

*Computes the dot product of two vectors in higher precision than the input arguments.*

```
int nspwDotProdExt(const short *vec1, const short *vec2, int n,
   int ScaleMode, int *ScaleFactor);
      /* integer dot product; short integer arguments */
ICplx nspvDotProdExt(const WCplx *vec1, const WCplx *vec2, int n,
   int ScaleMode, int *ScaleFactor);
      /* complex integer product; complex short integer arguments */
ICplx nspwvDotProdExt(const short *vec1, const WCplx *vec2, int n,
   int ScaleMode, int *ScaleFactor);
      /* first vector: real values; second vector: complex values */
```

```
ICplx nspvwDotProdExt(const WCplx *vec1, const short *vec2, int n,
int ScaleMode, int *ScaleFactor);
      /* first vector: complex values; second vector: real values */
```

| | |
|---|---|
| *vec1* | Pointer to the first vector to compute the dot product of two vectors. |
| *vec2* | Pointer to the second vector to compute the dot product of two vectors. |
| *n* | The number of elements in the vectors. |
| *ScaleMode*, *ScaleFactor* | See *Discussion* below. |

## Discussion

The `nsp?DotProdExt()` function computes the dot product of two vectors, *vec1*`[`*n*`]` and *vec2*`[`*n*`]`. When computing the dot product, complex flavors of the `nsp?DotProdExt()` function do not conjugate the second operand. The vectors *vec1*`[`*n*`]` and *vec2*`[`*n*`]` must be of equal length. If they are not, the function will return unpredictable results.

**CAUTION.** *In this function, the meaning of the arguments* *scaleMode* *and* *scaleFactor* *is different from that for the other functions. Namely, the function* `nsp?DotProdExt()` *always handles overflow by truncating the most significant bits of the result and preserving the sign bit, no matter what the current overflow control option (*`NSP_SATURATE` *or* `NSP_OVERFLOW`*).*

If you specify the `NSP_AUTO_SCALE` scaling mode, the function `nsp?DotProdExt()` will perform no scaling; on exit, the function will set *scaleFactor* to zero. The other scaling modes have the same meaning as in all other functions: with `NSP_NO_SCALE`, the *scaleFactor* argument is ignored; with `NSP_FIXED_SCALE`, the function multiplies the output value by $2^{-scaleFactor}$.

## bThresh1

*Performs the threshold operation on the elements of a vector in-place by limiting the element values by* `thresh`.

```
void nspsbThresh1(float *vec, int n, float thresh, int relOp);
      /* real values; single precision */
void nspcbThresh1(SCplx *vec, int n, float thresh, int relOP);
      /* complex input vector; real threshold; single precision */
void nspdbThresh1(double *vec, int n, double thresh, int relOp);
      /* real values; double precision */
void nspzbThresh1(DCplx *vec, int n, double thresh, int relOp);
      /* complex input vector; real threshold; double precision */
void nspwbThresh1(short *vec, int n, short thresh, int relOp);
      /* real values; short integer */
void nspvbThresh1(WCplx *vec, int n, short thresh, int relOp);
      /* real values; short integer */
```

| | |
|---|---|
| `vec` | Pointer to the vector on whose elements the threshold operation is performed. |
| `n` | The number of elements in the vector. |
| `thresh` | A value used to limit each element of `vec[n]`. This argument must always be real. For complex flavors, it must be positive and represent magnitude. |
| `relOP` | The values of this argument specify which relational operator to use and whether `thresh` is an upper or lower bound for the input. The `relOP` must have one of the following values: |

`NSP_GT` Specifies the "greater than" operator and `thresh` is an upper bound.

`NSP_LT` Specifies the "less than" operator and `thresh` is a lower bound.

### Discussion

The `nsp?bThresh1()` function performs the threshold operation on the input vector *`vec[n]`* in-place by limiting the input vector by the threshold value *`thresh`*. The *`relOP`* argument specifies which relational operator to use: “greater than” or “less than,” and determines whether *`thresh`* is an upper or lower bound for the input, respectively.

For example, the formula for `nsp?bThresh1()` called with the `NSP_GT` flag is:

$$vec[k] = \begin{cases} vec[k], & thresh > vec[k] \\ thresh, & otherwise \end{cases}$$

**Application Notes:** For `s`, `d`, `c`, and `z` flavors of `nsp?bThresh1()`, the *`thresh`* argument is always real, even for the complex flavors. For `w` and `v` flavors, the *`thresh`* argument is always integer, even for the complex flavor of this function.

For all complex flavors, *`thresh`* 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.

## bThresh2

*Performs the threshold operation on a vector by limiting the vector element values by `thresh` and places the results in a second vector.*

```
void nspsbThresh2(const float *src, float *dst, int n, float thresh,
      int relOp);  /* real values; single precision */
void nspcbThresh2(const SCplx *src, Scplx *dst, int n, float thresh,
      int relOP);
      /* complex vectors; real threshold; single precision */
void nspdbThresh2(const double *src, double *dst, int n,
      double thresh, int relOp);
      /* real values; double precision */
```

```
void nspzbThresh2(const DCplx *src, Dcplx *dst, int n,
      double thresh, int relOp);
      /* complex vectors; real threshold; double precision */
void nspwbThresh2(const short *src, short *dst, int n, short thresh,
      int relOp);  /* real values; short integer */
void nspvbThresh2(const WCplx *src, WCplx *dst, int n, short thresh,
      int relOp);  /* complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[n]*. |
| *dst* | Pointer to the vector *dst[n]*. |
| *n* | The number of elements in the vectors. |
| *thresh* | A value used to limit each element of *src[n]*. This argument must always be real. For complex flavors, it must be positive and represent magnitude. |
| *relOP* | The values of this argument specify which relational operator to use and whether *thresh* is an upper or lower bound for the input, accordingly. The *relOP* must have one of the following values:<br>NSP_GT Specifies the “greater than” operator and *thresh* is an upper bound.<br>NSP_LT Specifies the “less than” operator and *thresh* is a lower bound. |

## Discussion

The nsp?bThresh2() function performs the threshold operation on the input vector *src[n]* in-place. The function limits the input vector by the threshold value *thresh*. The *relOP* argument specifies which relational operator to use: “greater than” or “less than,” and determines whether *thresh* is an upper or lower bound for the input, respectively.

For example, the formula for the real versions of nsp?bThresh2() called with the NSP_GT flag is:

$$dst[k] = \begin{cases} src[k], & thresh > src[k] \\ thresh, & otherwise \end{cases}$$

**Application Note:** For `s`, `d`, `c`, and `z` flavors of `nsp?bThresh1()`, the *`thresh`* argument is always real, even for the complex flavors. For `w` and `v` flavors, the *`thresh`* argument is always integer, even for the complex flavor of this function.

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

## bInvThresh1

*Computes the inverse of vector elements in-place after limiting their magnitudes by the lower bound of `thresh`.*

```
void nspsbInvThresh1(float *vec, int n, float thresh);
      /* real values; single precision */
void nspcbInvThresh1(SCplx *vec, int n, float thresh);
      /* complex input vector; real threshold; single precision */
void nspdbInvThresh1(double *vec, int n, double thresh);
      /* real values; double precision */
void nspzbInvThresh1(DCplx *vec, int n, double thresh);
      /* complex input vector; real threshold; double precision */
```

| | |
|---|---|
| *`vec`* | Pointer to the vector *`vec[n]`*. |
| *`n`* | The number of elements in the vector. |
| *`thresh`* | A value, the lower bound of which is used to limit each element of *`vec[n]`*. This argument must always be real and positive. |

### Discussion

The `nsp?bInvThresh1()` function computes the inverse of elements of the *`n`*-length input vector *`vec[n]`* in-place. The computation occurs after first limiting the magnitude of each element by the lower bound of *`thresh`*. The limiting operation is performed to avoid division by zero. Since *thresh* represents a magnitude, it is always real and must always be positive. For complex versions, the magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

**Application Note:** This function should skip the limiting step if *thresh* is zero. In this case, if the function encounters zero-valued vector elements, the value of the corresponding elements in the result is set to `HUGE_VAL`, and a division-by-zero error is flagged with a call to `nspError()` after computation is complete.

## bInvThresh2

*Computes the inverse of vector elements after limiting their magnitudes by the lower bound of* `thresh` *and places the results in a second vector.*

```
void nspsbInvThresh2(const float *src, float *dst, int n, float
      thresh); /* real values; single precision */
void nspcbInvThresh2(const SCplx *src, SCplx *dst, int n, float
      thresh);
      /* complex vectors; real threshold; single precision */
void nspdbInvThresh2(const double *src, double *dst, int n, double
      thresh); /* real values; double precision */
void nspzbInvThresh2(const DCplx *vec, DCplx *dst, int n, double
      thresh);
      /* complex vectors; real threshold; double precision */
```

| | |
|---|---|
| *src* | Pointer to the input vector *src[n]*. |
| *dst* | Pointer to the output vector *dst[n]*. |
| *n* | The number of elements in the vectors. |
| *thresh* | A value, the lower bound of which is used to limit each element of *src[n]*. This argument must always be real and positive. |

### Discussion

The `nsp?bInvThresh2()` function computes the inverse of elements of the *n*-length input vector *src[n]* and stores the results in the output vector *dst[n]*. The computation occurs after first limiting the magnitude of each element by the lower bound *thresh*. The limiting operation is performed to avoid division by zero. Since *thresh* represents a magnitude, it is always real and must always be positive. For complex versions, the magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

**Application Note:** This function should skip the limiting step if *thresh* is zero. In this case, if the function encounters zero-valued vector elements, the value of the corresponding elements in the result is set to `HUGE_VAL`, and a division-by-zero error is flagged with a call to `nspError()` after computation is complete.

## bAbs1

*Computes the absolute values of vector elements in-place.*

```
void nspsbAbs1(float *vec, int n)
      /* real values; single precision */
void nspdbAbs1(double *vec, int n)
      /* real values; double precision */
void nspwbAbs1(short *vec, int n)
      /* real values; short integer */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |
| *n* | The number of elements in the vector. |

### Discussion

The `nsp?bAbs1()` function computes the absolute values of the elements of the *n*-length vector in-place.

## bAbs2

*Computes the absolute values of vector elements and stores the results in a second vector.*

```
void nspsbAbs2(const float *src, float *dst, int n)
      /* real values; single precision */
void nspdbAbs2(const double *src, double *dst, int n)
      /* real values; double precision */
void nspwbAbs2(const short *src, short *dst, int n)
      /* real values; short integer */
```

| | |
|---|---|
| `src` | Pointer to the vector `src[n]`. |
| `dst` | Pointer to the vector `dst[n]`. |
| `n` | The number of elements in the vectors. |

### Discussion

The `nsp?bAbs2()` function computes the absolute values of elements of the *n*-length input vector `src[n]` and stores the results in the output vector `dst[n]`.

## bSqr1

*Computes a square of each element of a vector in-place.*

```
void nspsbSqr1(float *vec, int n);
      /* real values; single precision */
void nspcbSqr1(SCplx *vec, int n);
      /* complex values; single precision */
void nspdbSqr1(double *vec, int n);
      /* real values; double precision */
```

```
void nspzbSqr1(DCplx *vec, int n);
      /* complex values; double precision */
void nspwbSqr1(short *vec, int n, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
void nspvbSqr1(WCplx *vec, int n, int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |
| *n* | The number of elements in the vector. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bSqr1()` function computes the square of each element in the *n*-length vector *vec[n]* in-place. The computation is performed as follows:

$vec[k] = vec[k]^2, \ 0 \le k < n$

## bSqr2

*Computes a square of each element of a vector and stores the result in a second vector.*

```
void nspsbSqr2(const float *src, float *dst int n);
      /* real values; single precision */
void nspcbSqr2(const SCplx *src, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbSqr2(const double *src, double *dst, int n);
      /* real values; double precision */
void nspzbSqr2(const DCplx *src, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbSqr2(const short *src, short *dst int n, int ScaleMode,
      int *ScaleFactor);
      /* real values; short integer */
```

```
void nspvbSqr2(const WCplx *src, WCplx *dst int n, int ScaleMode,
      int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[n]*. |
| *dst* | Pointer to the vector *dst[n]*. |
| *n* | The number of elements in the vectors. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bSqr2()` function computes the square of each element in the *n*-length vector *src[n]* and stores the results in the vector *dst[n]*. The computation is performed as follows:

*dst[k]* = *src[k]*$^2$, $0 \le k < n$

## bSqrt1

*Computes a square root of each element of a vector in-place.*

```
void nspsbSqrt1(float *vec, int n);
      /* real values; single precision */
void nspcbSqrt1(SCplx *vec, int n);
      /* complex values; single precision */
void nspdbSqrt1(double *vec, int n);
      /* real values; double precision */
void nspzbSqrt1(DCplx *vec, int n);
      /* complex values; double precision */
void nspwbSqrt1(short *vec, int n);
      /* real values; short integer */
void nspvbSqrt1(WCplx *vec, int n);
      /* complex values; short integer */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |
| *n* | The number of elements in the vector. |

### Discussion

The `nsp?bSqrt1()` function computes the square root of each element in the *n*-length vector *vec[n]* in-place. The computation is performed as follows:

$$vec[k] = \sqrt{vec[k]}, \quad 0 \le k < n$$

**Application Note:** If the real version of the `nsp?bSqrt1()` function encounters a negative value in the input, the value of the corresponding element in the output vector is undefined, and the error condition is signaled with a call to `nspError()` after all elements have been computed. The complex versions of the `nsp?bSqrt1()` function compute the square roots of the complex numbers with the positive real parts.

## bSqrt2

*Computes a square root of each element of a vector and stores the result in a second vector.*

```
void nspsbSqrt2(const float *src, float *dst int n);
      /* real values; single precision */
void nspcbSqrt2(const SCplx *src, SCplx *dst, int n);
      /* complex values; single precision */
void nspdbSqrt2(const double *src, double *dst, int n);
      /* real values; double precision */
void nspzbSqrt2(const DCplx *src, DCplx *dst, int n);
      /* complex values; double precision */
void nspwbSqrt2(const short *src, short *dst int n);
      /* real values; short integer */
void nspvbSqrt2(const WCplx *src, WCplx *dst int n);
      /* complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[n]*. |
| *dst* | Pointer to the vector *dst[n]*. |
| *n* | The number of elements in the vectors. |

### Discussion

The `nsp?bSqrt2()` function computes the square root of each element in the *n*-length vector *src[n]* and stores the results in the vector *dst[n]*. The computation is performed as follows:

$$dst[k] = \sqrt{src[k]}, \quad 0 \le k < n$$

**Application Note:** If the real version of the `nsp?bSqrt2()` function encounters a negative value in the input, the value of the corresponding element in the output vector is undefined, and the error condition is signalled with a call to `nspError()` after all elements have been computed. The complex versions of the `nsp?bSqrt2()` function compute the square roots of the complex numbers with the positive real parts.

## bExp1

*Computes* `e` *to the power of each element of a vector in-place.*

```
void nspsbExp1 (float *vec, int n);
      /* real values; single precision */
void nspdbExp1(double *vec, int n);
      /* real values; double precision */
void nspwbExp1 (short *vec, int n, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |

| | |
|---|---|
| *n* | The number of elements in the vector. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bExp1()` function computes `e` to the power of each element of the *n*-length vector *vec[n]* in-place.

$vec[k] = e^{vec[k]}, \quad 0 \le k < n$

**Application Note:** For the `nspwbExp1()`, `nspwbExp2()`, `nspwbLn1()`, `nspwbLn2()` functions, the result is rounded to the nearest integer after scaling.

**CAUTION.** *Due to the nature of these functions, considerable overflows occur during intermediate calculations. To ensure accuracy, autoscaling is recommended.*

## bExp2

*Computes* `e` *to the power of each element of a vector and stores the results in a second vector.*

```
void nspsbExp2 (const float *src, float *dst, int n);
      /* real values; single precision */
void nspsdbExp2(const float *src, double *dst, int n);
  /* real values; single precision input, double precision output */
void nspdbExp2(const double *src, double *dst, int n);
      /* real values; double precision */
void nspwbExp2 (const short *src, short *dst, int n, int ScaleMode,
      int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| `src` | Pointer to the vector `src[n]`. |
| `dst` | Pointer to the vector `dst[n]`. |
| `n` | The number of elements in the vectors. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bExp2()` function computes e to the power of each element of the *n*-length input vector, `src[n]`, and stores the results in a second vector, `dst[n]`.

$dst[k] = e^{src[k]}, \quad 0 \le k < n$

**Application Note:** See "Application Note" for "bExp1", page 3-33.

## bLn1

*Computes the natural logarithm of each element of a vector in-place.*

```
void nspsbLn1 (float *vec, int n);
      /* real values; single precision */
void nspdbLn1(double *vec, int n);
      /* real values; double precision */
void nspwbLn1 (short *vec, int n);
      /* real values; short integer */
```

| | |
|---|---|
| `n` | The number of elements in the vector. |

### Discussion

The `nsp?bLn1()` function computes the natural logarithm of each element of the *n*-length vector `vec[n]` in-place.

$vec[k] = \log_e (vec[k]), \quad 0 \le k < n$

The function returns -Inf (negative infinity) for zero-valued input vector elements, and NaN for negative elements.

## bLn2

*Computes the natural logarithm of each element of a vector and stores the results in a second vector.*

```
void nspsbLn2 (const float *src, float *dst, int n);
      /* real values; single precision */
void nspdbLn2(const double *src, double *dst, int n);
      /* real values; double precision */
void nspdsbLn2(const double *src, float *dst, int n);
 /* real values; double precision input, single precision output */
void nspwbLn2 (const short *src, short *dst, int n);
      /* real values; short integer */
```

| | |
|---|---|
| `src` | Pointer to the vector `scr[n]`. |
| `dst` | Pointer to the vector `dst[n]`. |
| `n` | The number of elements in the vectors. |

### Discussion

The `nsp?bLn2()` function computes the natural logarithm of each element of the *n*-length input vector, `src[n]`, and stores the results in a second vector, `dst[n]`.

$dst[k] = \log_e (src[k]), \quad 0 \le k < n$

The function returns -Inf (negative infinity) for zero-valued input vector elements, and NaN for negative elements.

## bLg1

*Computes the decimal logarithm of each element of a vector in-place.*

```
void nspsbLg1 (float *vec, int n);
      /* real values; single precision */
```

```
void nspdbLg1(double *vec, int n);
      /* real values; double precision */
void nspcbLg1 (SCplx *vec, int n);
      /* complex values; single precision */
```

*n* The number of elements in the vector.

### Discussion

The `nsp?bLg1()` function computes the decimal logarithm of each element of the *n*-length vector `vec[n]` in-place.

$vec[k] = \log_{10}(vec[k]), \quad 0 \le k < n$

The function flavors working on real data return -Inf (negative infinity) for zero-valued input vector elements, and NaN for negative elements.

## bLg2

*Computes the decimal logarithm of each element of a vector and stores the results in a second vector.*

```
void nspsbLg2 (const float *src, float *dst, int n);
      /* real values; single precision */
void nspdbLg2(const double *src, double *dst, int n);
      /* real values; double precision */
void nspcbLg2 (const SCplx *src, SCplx *dst, int n);
      /* complex values; single precision */
void nspczbLg(const SCplx *src, DCplx *dst, int n);
 /* complex values; single precision input, double precision output */
```

*src* Pointer to the vector `scr[n]`.

*dst* Pointer to the vector `dst[n]`.

*n* The number of elements in the vectors.

### Discussion

The `nsp?bLg2()` function computes the natural logarithm of each element of the *n*-length input vector, *src[n]*, and stores the results in a second vector, *dst[n]*.

*dst[k]* = $\log_{10}$ *(src[k])*, $0 \le k < n$

The function flavors working on real data return -Inf (negative infinity) for zero-valued input vector elements, and NaN for negative elements.

## bArctan1

*Computes the arctangent of each element of a vector in-place.*

```
void nspsbArctan1 (float *vec, int n);
      /* real values; single precision */
void nspdbArctan1 (double *vec, int n);
      /* real values; double precision */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |
| *n* | The number of elements in the vector. |

### Discussion

The `nsp?bArctan1()` function computes the arctangent of each element of the *n*-length vector *vec[n]* in-place.

*vec[k]* = arctg (*vec[k]*), $0 \le k < n$

The values are returned in radians and are in the range [-π/2, π/2].

## bArctan2

*Computes the arctangent of each element of a vector and stores the results in a second vector.*

```
void nspsbArctan2 (const float *src, float *dst, int n);
      /* real values; single precision */
void nspdbArctan2(const double *src, double *dst, int n);
      /* real values; double precision */
```

| | |
|---|---|
| `src` | Pointer to the vector `scr[n]`. |
| `dst` | Pointer to the vector `dst[n]`. |
| `n` | The number of elements in the vectors. |

### Discussion

The `nsp?bArctan2()` function computes the arctangent of each element of the *n*-length input vector, `src[n]`, and stores the results in a second vector, `dst[n]`.

$dst[k] = \text{arctg}\,(src[k]), \quad 0 \le k < n$

The values are returned in radians and are in the range $[-\pi/2, \pi/2]$.

## Logical and Shift Functions

This section describes the Signal Processing Library functions which perform logical and shift operations on vectors. Unlike arithmetic functions, the logical and shift functions are available only for short integer vectors.

For binary logical operations AND, OR and XOR, the library provides several functions:
`bAnd1()`, `bOr1()`, `bXor1()` for vector-scalar operations
`bAnd2()`, `bOr2()`, `bXor2()` for in-place vector-vector operations
`bAnd3()`, `bOr3()`, `bXor3()` for not-in-place vector-vector operations.

# bAnd1

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

```
void nspwbAnd1 (short val, short *dst, int n);
      /* real values; short integer */
```

| | |
|---|---|
| `val` | The input scalar value. |
| `dst` | Pointer to the vector `dst[n]`. |
| `n` | The number of elements in the vector `dst`. |

## Discussion

This function computes the bitwise AND of a scalar value val and each element of a destination vector `dst[n]` in-place.

# bAnd2

*Computes the bitwise AND of two vectors.*

```
void nspwbAnd2 (const short *src, short *dst, int n);
      /* real values; short integer */
```

| | |
|---|---|
| `src` | Pointer to the vector `src[n]`. |
| `dst` | Pointer to the vector `dst[n]`. |
| `n` | The number of elements in each of the above vectors. |

## Discussion

This function computes the bitwise AND of the corresponding elements of the vectors `src` and `dst` and overwrites the results on the vector `dst`.

## bAnd3

*Computes the bitwise AND of two vectors and stores the results in a third vector.*

```
void nspwbAnd3 (const short *srcA, const short *srcB, short *dst,
                int n);
      /* real values; short integer */
```

| | |
|---|---|
| srcA, srcB | Pointers to the two input vectors. |
| dst | Pointer to the output vector. |
| n | The number of elements in each of the above vectors. |

### Discussion

This function computes the bitwise AND of the corresponding elements of `srcA` and `srcB` and stores the results in the vector `dst`.

## bOr1

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

```
void nspwbOr1 (short val, short *dst, int n);
      /* real values; short integer */
```

| | |
|---|---|
| val | The input scalar value. |
| dst | Pointer to the vector `dst[n]`. |
| n | The number of elements in the vector `dst`. |

### Discussion

This function computes the bitwise OR of a scalar value `val` and each element of a destination vector `dst[n]` in-place.

## bOr2

*Computes the bitwise OR of two vectors.*

```
void nspwbOr2 (const short *src, short *dst, int n);
      /* real values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector `src[n]`. |
| *dst* | Pointer to the vector `dst[n]`. |
| *n* | The number of elements in each of the above vectors. |

### Discussion

This function computes the bitwise OR of the corresponding elements of the vectors `src` and `dst` and overwrites the results on the vector `dst`.

## bOr3

*Computes the bitwise OR of two vectors and stores the results in a third vector.*

```
void nspwbOr3 (const short *srcA, const short *srcB, short *dst,
                int n);
      /* real values; short integer */
```

| | |
|---|---|
| *srcA, srcB* | Pointers to the two input vectors. |
| *dst* | Pointer to the output vector. |
| *n* | The number of elements in each of the above vectors. |

### Discussion

This function computes the bitwise OR of the corresponding elements of `srcA` and `srcB` and stores the results in the vector `dst`.

## bXor1

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

```
void nspwbXor1 (short val, short *dst, int n);
      /* real values; short integer */
```

| | |
|---|---|
| *val* | The input scalar value. |
| *dst* | Pointer to the vector *dst*[*n*]. |
| *n* | The number of elements in the vector *dst*. |

### Discussion

This function computes the bitwise XOR of a scalar value *val* and each element of a destination vector *dst*[*n*] in-place.

## bXor2

*Computes the bitwise XOR of two vectors.*

```
void nspwbXor2 (const short *src, short *dst, int n);
      /* real values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector *src*[*n*]. |
| *dst* | Pointer to the vector *dst*[*n*]. |
| *n* | The number of elements in each of the above vectors. |

### Discussion

This function computes the bitwise XOR of the corresponding elements of the vectors *src* and *dst* and overwrites the results on the vector *dst*.

## bXor3

*Computes the bitwise XOR of two vectors and stores the results in a third vector.*

```
void nspwbXor3 (const short *srcA, const short *srcB, short *dst,
                int n);
      /* real values; short integer */
```

| | |
|---|---|
| `srcA, srcB` | Pointers to the two input vectors. |
| `dst` | Pointer to the output vector. |
| `n` | The number of elements in each of the above vectors. |

### Discussion

This function computes the bitwise XOR of the corresponding elements of `srcA` and `srcB` and stores the results in the vector `dst`.

## bNot

*Computes the bitwise NOT of the input vector elements in-place.*

```
void nspwbNot (short *dst, int n);
      /* real values; short integer */
```

| | |
|---|---|
| `dst` | Pointer to the vector `dst[n]`. |
| `n` | The number of elements in the vector `dst`. |

### Discussion

This function computes the bitwise NOT of the elements of the vector `dst`, overwriting the results on this vector.

## bShiftL

*Shifts bits in vector elements to the left.*

```
void nspwbShiftL (short *dst, int n, int nShift);
      /* real values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector `dst[n]`. |
| *n* | The number of elements in the vector `dst`. |
| *nShift* | The number of bits by which the function shifts each element of the vector `dst`. |

### Discussion

This function shifts each element of the vector `dst` by `nShift` bits to the left and overwrites the results on `dst`.

## bShiftR

*Shifts bits in vector elements to the right.*

```
void nspwbShiftR (short *dst, int n, int nShift);
      /* real values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector `dst[n]`. |
| *n* | The number of elements in the vector `dst`. |
| *nShift* | The number of bits by which the function shifts each element of the vector `dst`. |

### Discussion

This function shifts each element of the vector `dst` by `nShift` bits to the right and overwrites the results on `dst`.

# Vector Measure Functions

This section describes the Signal Processing Library functions that compute the vector measure values: maximum, minimum, mean, and standard deviation.

## Max

*Returns the maximum value of a vector.*

```
float nspsMax(const float *vec, int n);
      /* real values; single precision */
double nspdMax(const double *vec, int n);
      /* real values, double precision */
short nspwMax(const short *vec, int n);
      /* real values; short integer */
```

| | |
|---|---|
| `vec` | Pointer to the vector `vec[n]`. |
| `n` | The number of elements in the vector. |

### Discussion

The `nsp?Max()` function returns the maximum value of the `n`-length input vector `vec[n]`.

## MaxExt

*Returns the maximum value of a vector and the index of the maximum element.*

```
float nspsMaxExt(const float *vec, int n, int *index);
      /* real values; single precision */
double nspdMaxExt(const double *vec, int n, int *index);
      /* real values, double precision */
```

```
short nspwMaxExt(const short *vec, int n, int *index);
      /* real values; short integer */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |
| *n* | The number of elements in the vector. |
| *index* | On exit, contains the index of the maximum element. |

### Discussion

The `nsp?MaxExt()` function returns the maximum value of the input vector `vec[n]` and stores in *index* the index of the maximum element.

## Min

*Returns the minimum value of a vector.*

```
float nspsMin(const float *vec, int n);
      /* real values; single precision */
double nspdMin(const double *vec, int n);
      /* real values, double precision */
short nspwMin(const short *vec, int n);
      /* real values; short integer */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |
| *n* | The number of elements in the vector. |

### Discussion

The `nsp?Min()` function returns the minimum value of the *n*-length input vector *vec[n]*.

## MinExt

*Returns the minimum value of a vector and the index of the minimum element.*

```
float nspsMinExt(const float *vec, int n, int *index);
      /* real values; single precision */
double nspdMinExt(const double *vec, int n, int *index);
      /* real values, double precision */
short nspwMinExt(const short *vec, int n, int *index);
      /* real values; short integer */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |
| *n* | The number of elements in the vector. |
| *index* | On exit, contains the index of the minimum element. |

### Discussion

The `nsp?MinExt()` function returns the minimum value of the input vector *vec*[*n*] and stores in *index* the index of the minimum element.

## Sum

*Computes the sum of vector elements.*

```
float nspsSum(const float *src, int n);
      /* real values; single precision */
double nspdSum(const double *src, int n);
      /* real values, double precision */
int nspwSum(const short *src, int n, int ScaleMode, int* ScaleFactor);
      /* real values; short integer */
NSPStatus nspcSum(const SCplx *src, int n, SCplx *sum);
      /* complex values; single precision */
NSPStatus nspzSum(const DCplx *src, int n, DCplx *sum);
      /* complex values; double precision */
```

```
NSPStatus nspvSum(const WCplx *src, int n, ICplx *sum,
    int ScaleMode, int* ScaleFactor);
      /* short integer complex values */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[n]*. |
| *n* | The number of elements in the vector |
| *sum* | The sum of vector elements. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?Sum()` function computes the sum of input vector elements. Function flavors that operate on short integer data perform scaling of the output result according to *ScaleMode* and *ScaleFactor*.

## Mean

*Computes the mean value of a vector.*

```
float nspsMean(const float *vec, int n);
      /* real values; single precision */
double nspdMean(const double *vec, int n);
      /* real values, double precision */
short nspwMean(const short *vec, int n);
      /* real values; short integer */
```

| | |
|---|---|
| *vec* | Pointer to the vector *vec[n]*. |
| *n* | The number of elements in the vector. |

### Discussion

The `nsp?Mean()` function computes the mean (average) of the *n*-length input vector *vec[n]*. The mean of *vec* is defined by the formula:

$$mean = \frac{1}{n}\sum_{k=0}^{n-1} vec[k]$$

# StdDev

*Computes the standard deviation value of a vector.*

```
float nspsStdDev(const float *vec, int n);
      /* real values; single precision */
double nspdStdDev(const double *vec, int n);
      /* real values; double precision */
short nspwStdDev(const short *vec, int n, int ScaleMode,
      int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| `vec` | Pointer to the vector `vec[n]`. |
| `n` | The number of elements in the vector. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

The `nsp?StdDev()` function computes the standard deviation of the `n`-length input vector, `vec[n]`. The standard deviation of `vec[n]` is defined by the formula:

$$stdDev = \left( \frac{1}{n-1} \sum_{k=0}^{n-1} (vec[k] - mean)^2 \right)^{1/2}.$$

## Norm

*Computes the norm of a vector or of two vectors' difference.*

```
float nspsNorm(const float *srcA,const float *srcB,int n,int flag);
      /* real values; single precision */
float nspcNorm(const SCplxt *srcA,const SCplx *srcB,int n,int flag);
      /* complex values; single precision */
double nspdNorm(const double *srcA, const double *srcB, int n,
      int flag);
      /* real values; double precision */
double nspzNorm(const DCplxt *srcA, const DCplx *srcB, int n,
      int flag);
      /* complex values; double precision */
float nspwNorm(const short *srcA,const short *srcB,int n,int flag);
      /* real values; short */
float nspvNorm(const WCplxt *srcA,const WCplx *srcB,int n,int flag);
      /* complex values; short */
```

| | |
|---|---|
| *srcA, srcB* | Pointers to the input vectors *a[n]* and *b[n]*. |
| *n* | The number of elements in the input vectors. |
| *flag* | Specifies the norm to compute; must be one of the following: NSP_C, NSP_L1 or NSP_L2, or the bitwise logical OR of NSP_RELATIVE and one of the above three values. |

### Discussion

The nsp?Norm() function computes the $L_1$, $L_2$, or $C$ norm of the input vectors' difference: $||a-b||$. You specify the norm by the flag NSP_C, NSP_L1, or NSP_L2. If the *srcB* pointer is NULL, the function computes the norm of *a*.

The $L_1$ norm of *a*-*b* is defined by the formula:

$$\|a-b\|_{L_1} = \sum_{k=0}^{n-1} |a[k]-b[k]|.$$

The $L_2$ norm of $a-b$ is defined by the formula:

$$\|a-b\|_{L_2} = \left(\sum_{k=0}^{n-1} |a[k]-b[k]|^2\right)^{1/2}.$$

The $C$ norm of $a-b$ is defined by the formula:

$$\|a-b\|_C = \max_k |a[k]-b[k]|.$$

If *`flag`* is a bitwise OR of `NSP_RELATIVE` and one of the above values, the computed norm is divided by the norm of $a$, and the function returns the "relative error" $\|a-b\| / \|a\|$.

## NormExt

*Computes the norm of a vector or of two vectors' difference; converts the result to integer data type.*

```
int nspwNormExt(const short *srcA, const short *srcB, int n,
      int flag, int scaleMode, int *scaleFactor
      /* real values; short */
int nspvNormExt(const WCplx *srcA, const WCplx *srcB, int n,
      int flag, int scaleMode, int *scaleFactor);
      /* complex values; short */
```

| | |
|---|---|
| *`srcA, srcB`* | Pointers to the input vectors *`a[n]`* and *`b[n]`*. |
| *`n`* | The number of elements in the input vectors. |

| | |
|---|---|
| *flag* | Specifies the norm to compute; must be one of the following: NSP_C, NSP_L1 or NSP_L2, or the bitwise logical OR of NSP_RELATIVE and one of the above three values. For more information about the flag values, see nsp?Norm() on the previous page. |
| *scaleMode*, *scaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

Similar to nsp?Norm(), the function nsp?NormExt() computes the $L_1$, $L_2$, or $C$ norm of the vector *srcA* or of the difference of the vectors *srcA* and *srcB*. Then the function scales the result to integer data type, as specified by the *scaleMode* and *scaleFactor* arguments.

## Vector Conjugation Functions

This section describes the Signal Processing Library functions which perform complex conjugation of vectors. Some of the functions, in addition to performing complex conjugation, extend the length of the output vector. Others store the results of the complex conjugation in reverse order.

The vector conjugation functions are often useful when working with the fast Fourier transform of real signals. Because the fast Fourier transform of a real signal is complex conjugate-symmetric, the FFT function needs to generate only the first (*N*/2) + 1 output samples. This enables rapid calculation of the FFT of real-valued signals. You can calculate the remainder of the samples simply by conjugating these first samples. The functions described in this section can be used for this purpose, especially nsp?bConjExtend1() and nsp?bConjExtend2(). For more information, see page 7-38, "RealFft" (for nsp?RealFft()) and Example 7-10 in Chapter 7.

## bConj1

*Computes the complex conjugate of a vector.*

```
void nspcbConj1(SCplx *vec, int n);
      /* complex values; single precision */
void nspzbConj1(DCplx *vec, int n);
      /* complex values; double precision */
void nspvbConj1(WCplx *vec, int n);
      /* complex values; short integer */
```

| | |
|---|---|
| `vec` | Pointer to the vector whose complex conjugate is to be computed. |
| `n` | The number of values in the vector `vec[n]`. |

### Discussion

The function `nsp?bConj1()` conjugates the *n*-length array `vec[n]` in-place. The vector conjugation is defined as follows:

$vec[k] = vec[k]^*, \quad 0 \le k < n$

## bConj2

*Computes the complex conjugate of a vector and stores the result in a second vector.*

```
void nspcbConj2(const SCplx *src, SCplx *dst, int n);
      /* complex values; single precision */
void nspzbConj2(const DCplx *src, DCplx *dst, int n);
      /* complex values; double precision */
void nspvbConj2(const WCplx *src, WCplx *dst, int n);
      /* complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector whose complex conjugate is to be computed. |
| *dst* | Pointer to the vector which stores the complex conjugate of the vector *src[n]*. |
| *n* | The number of values in the vectors. |

### Discussion

The function `nsp?bConj2()` computes the element-wise conjugation of the vector *src[n]* and stores the result in the vector *dst[n]*. The element-wise conjugation of the vector is defined as follows:

*dst[k]* = *src[k]**, $0 \le k < n$

The vectors *dst[n]* and *src[n]* must be of equal length.

## bConjExtend1

*Computes the conjugate-symmetric extension of a vector in-place.*

```
void nspcbConjExtend1(SCplx *vec, int n);
      /* complex values; single precision */
void nspzbConjExtend1(DCplx *vec, int n);
      /* complex values; double precision */
void nspvbConjExtend1(WCplx *vec, int n);
      /* complex values; short integer */
```

| | |
|---|---|
| *n* | The number of values in the input vector *vec*. The output *vec* has 2*n* - 2 elements. |
| *vec* | Pointer to the vector whose conjugate-symmetric extension is to be computed and stored in-place. |

### Discussion

The function `nsp?bConjExtend1()` computes the conjugate-symmetric extension of the vector *vec[n]* in-place. The conjugate-symmetric extension is defined as follows:

$$vec[k] = \begin{cases} vec[k] & \text{for } 0 \le k < n \\ vec[2n-k-2]^* & \text{for } n \le k < 2n-2 \end{cases}$$

The length of the output vector is $2n$ - 2; $vec[0]$ and $vec[n-1]$ should be real, but this is neither verified nor enforced by this function.

The `nsp?bConjExtend1()` function can be used to extend the length of the output arrays produced by the FFT of a real signal. For more information, see page 7-38, "RealFft" (for `nsp?RealFft()`) in Chapter 7.

## bConjExtend2

*Computes the conjugate-symmetric extension of a vector and stores the result in a second vector.*

```
void nspcbConjExtend2(const SCplx *src, SCplx *dst, int n);
            /* complex values; single precision */
void nspzbConjExtend2(const DCplx *src, DCplx *dst, int n);
            /* complex values; double precision */
void nspvbConjExtend2(const WCplx *src, WCplx *dst, int n);
            /* complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector whose conjugate-symmetric extension is to be computed. |
| *dst* | Pointer to the vector which stores the conjugate-symmetric extension of the vector *src[n]*. |
| *n* | The number of values in the vector *src[n]*. |

### Discussion

The function `nsp?bConjExtend2()` computes the conjugate-symmetric extension of the $n$-length vector $src[n]$. The result is stored in the vector $dst[n]$. The conjugate-symmetric extension is defined as follows:

$$dst[k] = \begin{cases} src[k] & \text{for } 0 \le k < n \\ src[2n - k - 2]^* & \text{for } n \le k < 2n - 2 \end{cases}$$

The length of the output vector $dst$ is $2n - 2$;
$src[0]$ and $src[n - 1]$ should be real, but this is neither verified nor enforced by this function.

The `nsp?bConjExtend2()` function can be used to extend the length of the output arrays produced by the FFT of a real signal. For more information, see page 7-38, “RealFft” (for `nsp?RealFft()`) in Chapter 7.

## bConjFlip2

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

```
void nspcbConjFlip2(const SCplx *src, SCplx *dst, int n);
      /* complex values; single precision */
void nspzbConjFlip2(const DCplx *src, DCplx *dst, int n);
      /* complex values; double precision */
void nspvbConjFlip2(const WCplx *src, WCplx *dst, int n);
      /* complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector whose complex conjugate is to be computed and stored in reverse order. |
| *dst* | Pointer to the vector which stores the complex conjugate of the vector *src[n]* in reverse order. |
| *n* | The number of values in the vectors. |

### Discussion

The `nsp?bConjFlip2()` function computes the conjugate of the vector *src[n]* and stores the result, in reverse order, in the vector *dst[n]*. The vectors *dst[n]* and *src[n]* must be of equal length. The complex conjugate, stored in reverse order, is defined as follows:

$dst[k] = src[n - k - 1]^*, 0 \le k < n$

If the memory locations of *src[n]* and *dst[n]* overlap, this function will fail.

The `nsp?bConjFlip2()` function is useful when working with the FFT of a real signal. For more information, see page 7-38, "RealFft" (for `nsp?RealFft()`) in Chapter 7.

## Resampling Functions

The functions described in this section manipulate signal samples. Resampling 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).
- Perform signal resampling by using the multi-rate finite impulse response (MR FIR) filter.

The up-sample and down-sample functions are used by some filter functions described in Chapter 8.

## UpSample

*Up-samples a signal, conceptually increasing its sampling rate by an integer factor.*

```
void nspsUpSample(const float *src, int srcLen, float *dst,
      int *dstLen, int factor, int *phase);
      /* real values; single precision */
void nspcUpSample(const SCplx *src, int srcLen, SCplx *dst, int
*dstLen, int factor, int *phase);
      /* complex values; single precision */
void nspdUpSample(const double *src, int srcLen, double *dst, int
*dstLen, int factor, int *phase);
      /* real values; double precision */
void nspzUpSample(const DCplx *src, int srcLen, DCplx *dst, int
*dstLen, int factor, int *phase);
      /* complex values; double precision */
void nspwUpSample(const short *src, int srcLen, short *dst, int
*dstLen, int factor, int *phase);
      /* real values; short integer */
void nspvUpSample(const WCplx *src, int srcLen, WCplx *dst, int
*dstLen, int factor, int *phase);
      /* complex values; short integer */
```

| | |
|---|---|
| `dst` | Pointer to the output array. |
| `dstLen` | An output parameter: the number of samples in the `dst` array. It is equal to the product (`srcLen` * `factor`). |
| `factor` | The factor by which the signal is up-sampled. That is, `factor` -1 zeros are inserted after each sample of `src[n]`. |
| `phase` | A parameter which determines where each sample from `src[n]` lies within each output block of `factor` samples. The value of `phase` is required to be |

$0 \le phase < factor$. The value of this parameter can be used for the next up-sampling with the same `factor` and next `src[n]`.

`src` Pointer to the input array (the signal to be up-sampled).

`srcLen` The number of samples in the `src` array.

## Discussion

The `nsp?UpSample()` function up-samples the array `src[n]` by factor `factor` with phase `phase`, and stores the result in the array `dst[n]`.

Up-sampling inserts `factor` - 1 zeros between each sample of `src[n]`. The `phase` argument determines where each sample from the `src[n]` array lies within each output block of `factor` samples. It is required that $0 \le phase < factor$.

The values of the output array `dst[n]` are defined as follows:

$$dst[k] = \begin{cases} src\left[\dfrac{k - phase}{factor}\right], & 0 \le k < factor \times srcLen \\ 0, & otherwise \end{cases}$$

so that:

$$dst[(factor \times k) + phase] = src[k], \quad 0 \le k < srcLen$$

Conceptually, $phase = 0$ places each source sample at the oldest time-slot (closest to the start of the array) within each block, while $phase = factor - 1$ places each source sample at the newest time-slot (closest to the end of the array) within each block.

To better understand the `phase` argument, consider the continuous-time analogs of the `src[n]` and `dst[n]` signals. Assume `src[n]` was sampled every $T$ seconds. After up-sampling, `dst[n]` is sampled every $T/factor$ seconds. Assuming that both $src[0]$ and $dst[0]$ correspond to time $t = 0$ then:

$$s(t) = \sum_{n=0}^{srcLen-1} \delta(t - nT) \cdot src[n]$$

$$d(t) = \sum_{n=0}^{(srcLen \times factor)-1} \delta\left(t - \frac{nT}{factor}\right) \cdot dst[n]$$

Thus,

$$d(t) = s\left(t - \frac{phase \times T}{factor}\right)$$

with the two signals identical when `phase` = 0. With this interpretation, a `phase` > 0 results in a non-causal operation, but the non-causality is less than $T$ seconds.

For example, if `factor` = 3 and the source array `src(x)` is defined as

`src(x)` = {$x_1$, $x_2$, $x_3$},

then for `phase` = 0, the destination array `dst(x)` is defined as

`dst(x)` = {$x_1$, 0, 0, $x_2$, 0, 0, $x_3$, 0, 0},

and for `phase` = 2, the destination array `dst(x)` is defined as

`dst(x)` = {0, 0, $x_1$, 0, 0, $x_2$, 0, 0, $x_3$}.

Interpolation and up-sampling are closely related. Here, up-sampling refers to inserting zero samples, while interpolation refers to up-sampling followed by filtering. The filtering is intended to give the inserted samples a value close to the values of their neighboring samples in the original signal.

**Application Notes:** The conventions for the `phase` arguments to the `nsp?UpSample()` and `nsp?DownSample()` functions are chosen so that up-sampling followed by down-sampling with the same `phase` and `factor` arguments result in the original signal. Up-sampling followed by down-sampling with equal `factor` arguments but unequal `phase` arguments result in a zero signal.

## DownSample

*Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.*

```
void nspsDownSample(const float *src, int srcLen, float *dst,
      int *dstLen, int factor, int *phase);
      /* real values; single precision */
```

```
void nspcDownSample(const SCplx *src, int srcLen, SCplx *dst,
      int *dstLen, int factor, int *phase);
      /* complex values; single precision */
void nspdDownSample(const double *src, int srcLen, double *dst,
      int *dstLen, int factor, int *phase);
      /* real values; double precision */
void nspzDownSample(const DCplx *src, int srcLen, DCplx *dst,
      int *dstLen, int factor, int *phase);
      /* complex values; double precision */
void nspwDownSample(const short *src, int srcLen, short *dst,
      int *dstLen, int factor, int *phase);
      /* real values; short integer */
void nspvDownSample(const WCplx *src, int srcLen, WCplx *dst,
      int *dstLen, int factor, int *phase);
      /* complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the input array holding the signal samples to be down-sampled. |
| *srcLen* | The number of samples in the input array *src[n]*. |
| *dst* | Pointer to the array that holds the output of the `nsp?DownSample()` function. |
| *dstLen* | The number of samples in the *dst* array. |
| *factor* | The factor by which the signal is down-sampled. That is, *factor* - 1 samples are discarded from every block of *factor* samples in *src[n]*. |
| *phase* | The **input** value of *phase* determines which of the samples within each block is not discarded. It is required to satisfy the condition 0 ≤ *phase* < *factor*.<br>The function adjusts the **output** *phase* if *srcLen* is not a multiple of *factor*. The output value of *phase* is the input *phase* + *dstLen***factor* - *srcLen*.<br>This is to allow for the continuous down-sampling of several arrays. |

### Discussion

The `nsp?DownSample()` function down-samples the *srcLen* length array *src[n]* by factor *factor* with phase *phase*, storing the result in the array *dst[n]*.

Down-sampling discards *factor* - 1 samples from *src[n]*, copying one sample from each block of *factor* samples from *src[n]* to *dst[n]*. The *phase* argument determines which of the samples in each block is not discarded. It is required that $0 \le phase < factor$. The values in the output array *dst[n]* are defined as follows:

$$dst[k] = src[(factor \times k) + phase], \quad 0 \le k < \frac{srcLen}{factor}$$

Conceptually, `phase` = 0 extracts the oldest sample within each block (closest to the start of the array), and *phase* = *factor* -1 extracts the newest sample within each block (closest to the end of the array).

Down-sampling and decimation are closely related. Here, down-sampling refers to discarding samples, while decimation refers to filtering followed by down-sampling. The filtering is intended to prevent aliasing distortion in the subsequent down-sampling.

**Application Notes:** The conventions for the input *phase* arguments to the `nsp?UpSample()` and `nsp?DownSample()` functions are chosen so that up-sampling followed by down-sampling with the same input *phase* and *factor* arguments result in the original signal. Up-sampling followed by down-sampling with equal *factor* arguments but unequal input *phase* arguments result in a zero signal.

## Resampling with filtering

These functions use multi-rate FIR filters to resample the input signal. The resampling parameters passed to the functions include the input vector length, the cut-off frequency of the lowpass filter, and the required output vector length. The resampling functions use these data to determine the parameters of the appropriate multi-rate FIR filter, which is then invoked by the corresponding library MR FIR function call to filter the input signal and produce the resampled output vector of specified length. The FIR filter parameters that are determined include integer *`upFactor`* and *`downFactor`* values, and the set of FIR filter taps corresponding to the specified cut-off frequency of the lowpass filter.

## SampInit

*Initializes resampling parameters structure.*

```
NSPStatus nspsSampInit (NSPSampState* sampSt, float* factorRange,
    float* freq, int nFactors, int nTaps)
NSPStatus nspdSampInit (NSPSampState* sampSt, float* factorRange,
    float* freq, int nFactors, int nTaps)
```

| | |
|---|---|
| *`sampSt`* | The pointer to a `NSPSampState` structure which will contain the resampling parameters data. |
| *`factorRange`* | The vector of predefined resampling factor quotients. Each value is equal to the quotient of the input vector length divided by the output vector length. |
| *`freq`* | The vector of cut-off frequencies for the lowpass filters |
| *`nFactors`* | The length of frequency and resampling factor vectors. |
| *`nTaps`* | The number of filter taps. |

### Discussion

The function `nsp?SampInit()` copies the resampling data contained in the *`factorRange`* and *`freq`* vectors into the structure *`sampSt`* which stores the resampling parameters. Then, the `nspdFirLowpass` function with Hamming smoothing window is invoked multiple times to compute sets of *`ntaps`* coefficients of the lowpass filter for each of the cut-off frequency values in *`freq`*. The filter coefficients are normalized. Pointers to the created work arrays and their lengths are written to the *`sampSt`* structure.

The function returns the following codes:

| | |
|---|---|
| NSP_StsOk | Indicates no error. It means that the resampling parameters structure was successfully initialized. |
| NSP_StsNullPtr | Indicates an error condition, if a NULL pointer to a structure or an array is specified. |
| NSP_StsBadSizeValue | Indicates an error condition, if *Nfactors* is less than or equal to zero, or *nTaps* is less than 5. |
| NSP_StsBadFreq | Indicates an error condition, if *freq* contains a value outside the range [0, 0.5]. |
| NSP_StsBadFact | Indicates an error condition, if *factorRange* contains zero or negative value. |
| NSP_StsNoMem | Indicates an error condition, if no memory for the work array allocation is available. |

## Samp

*Performs resampling of the input signal using the multi-rate FIR filter.*

```
NSPStatus nspsSamp(NSPSampState* sampSt,const float* src,
    int srcLen, float* dst, int dstLen)
NSPStatus nspdSamp(NSPSampState* sampSt,const double* src,
    int srcLen, double* dst, int dstLen)
```

| | |
|---|---|
| *sampSt* | The pointer to NSPSampState structure which contains the resampling parameters. |
| *src* | The input signal vector. |
| *srcLen* | Length of the input vector. |
| *dst* | The output signal vector. |
| *dstLen* | Length of the output vector. |

### Discussion

The function nsp?Samp() scans the resampling factors array and selects the value nearest to the given *srcLen/dstLen* ratio. This value is used to select the corresponding set of taps to be used in multi-rate FIR filtering. The values of

*upFactor* and *downFactor* for MR FIR filter are chosen to be mutually prime. The filter initialization takes place and after that the bFir filtering function is invoked.
The values of the input and output vector lengths are stored in the SampSt structure. If these values are not changed until the next call to the nsp?Samp() function, then the FIR filter selection and initialization steps are skipped and the previously initialized filter is used.

The function returns the following codes:

| | |
|---|---|
| NSP_StsOk | Indicates no error. It means that the output vector values were computed successfully. |
| NSP_StsNullPtr | Indicates an error condition, if a NULL pointer to a structure or a vector is specified. |
| NSP_StsBadSizeValue | Indicates an error condition, if *srcLen* or *dstLen* is less than or equal to zero. |
| NSP_StsNoMem | Indicates an error condition, if no memory for the FIR filter initialization is available. |

## SampFree

*Frees work array memory which is pointed to in the resampling data structure sampSt.*

```
void nspSampFree(NSPSampState* sampSt)
```

| | |
|---|---|
| *sampSt* | The pointer to a NSPSampState structure which contains the resampling parameters. |

### Discussion

Use the function nspSampFree to free the memory blocks that were allocated for the work arrays by the nsp?SampInit() functions.

The example code below shows how to use functions that perform resampling of the input signal combined with filtering.

### Example 3-1 Using the Resampling Functions

```
/* Use resampling functions to obtain an output data vector of fixed
length from input data vectors of different length. Antialiasing is
applied. */
#define nsp_UsesVector
#define nsp_UsesTransform
#include "nsp.h"
#define NR  2
int main( void ) {
   const int order = 8;        /* order of FFT transform */
   const int nFactors = NR;      /* num of factors and freq values */
   const int inLen1 = 512;       /* length of first input vector */
   const int inLen2 = 384;      /* length of second input vector */
   const int outLen = 1<<order;   /* length of the output vector */
   const int nTaps = 63;          /* number of filter coefficients */
   float freqs[NR] = {0.1f, 0.2f};  /* cut off frequencies array */
   float factors[NR] = {            /* resampling factors array */
      (float)inLen1/outLen, (float)inLen2/outLen };
   NSPSampState sampSt;             /* resampling state structure */
   /* allocate memory for the arrays */
   float *vin1 = nspsMalloc( inLen1 );
   float *vin2 = nspsMalloc( inLen2 );
   float *vout = nspsMalloc( outLen );
   /* Input data are generated here */
   /* initialize resampling structure */
if(NSP_StsOk == nspsSampInit( &sampSt, factors, freqs, nFactors, nTaps))
   {                 /* decimate with antialiasing */
      nspsSamp( &sampSt, vin1, inLen1, vout, outLen );
      /* use data vector of fixed length */
      nspsRealFft( vout, order, NSP_Forw );
      nspsSamp( &sampSt, vin2, inLen2, vout, outLen );
      nspsRealFft( vout, order, NSP_Forw );
   }
   /* free resampling structure and deallocate memory */
   nspsRealFft( vout, order, NSP_Free );
   nspSampFree( &sampSt );
   nspFree( vout );
   nspFree( vin2 );
   nspFree( vin1 );
   return NSP_StsOk == nspGetErrStatus();
}
```

# Vector Correlation Functions

This section describes the Signal Processing Library functions which perform correlation of a vector or two vectors. The `nsp?AutoCorr` functions estimate the normal, biased, and unbiased auto-correlation of a vector. The `nsp?CrossCorr` function estimates the cross-correlation of two vectors.

## AutoCorr

*Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.*

```
void nspsAutoCorr(const float *src, int len, float *dst,
      int nLags, int flag);
      /* real values; single precision; */
void nspcAutoCorr(const SCplx *src, int len, SCplx *dst,
      int nLags, int flag);
      /* complex values; single precision; */
void nspdAutoCorr(const double *src, int len, double *dst,
      int nLags, int flag); /* real values; double precision; */
void nspzAutoCorr(const DCplx *src, int len, DCplx *dst,
      int nLags, int flag); /* complex values; double precision; */
void nspwAutoCorr(const short *src, int len, short *dst,
      int nLags, int flag, int ScaleMode, int *ScaleFactor);
      /* real values; short integer; */
void nspvAutoCorr(const WCplx *src, int len, WCplx *dst,
      int nLags, int flag, int ScaleMode, int *ScaleFactor);
      /* complex values; short integer; */
```

| | |
|---|---|
| *src* | Pointer to the vector to be estimated for an auto-correlation. |
| *len* | The number of values in the *src* vector. |
| *dst* | Pointer to the vector which stores the estimated auto-correlation results of the vector *src[len]*. |
| *nLags* | The number of lags to compute, starting with a lag of zero. The lags are stored in the *dst[len]* vector. |

| | |
|---|---|
| *flag* | Indicates the kind of auto-correlation to be computed: normal, biased, or unbiased. |
| *scaleMode*, *scaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

The `nsp?AutoCorr()` function estimates normal, biased, or unbiased auto-correlation of the *len*-length vector *src[len]* and stores the results in the vector *dst[len]*. The *flag* argument indicates what kind of auto-correlation is to be computed. Table 3-1 lists the flag argument values.

**Table 3-1 Value for the flag Argument for Auto-Correlation Function**

| Value | Description |
|---|---|
| `NSP_Normal` | Specifies that the normal auto-correlation to be computed. |
| `NSP_Biased` | Specifies that the biased auto-correlation to be computed. |
| `NSP_UnBiased` | Specifies that the unbiased auto-correlation to be computed. |

The auto-correlation is defined by the following equations:

$$dst[n] = \sum_{k=0}^{len-1} src[k]^* \cdot src(k+n), \quad 0 \le n < nLag \qquad \text{(normal)}$$

$$dst[n] = \frac{1}{len} \sum_{k=0}^{len-1} src[k]^* \cdot src(k+n), \quad 0 \le n < nLag \qquad \text{(biased)}$$

$$dst[n] = \frac{1}{len-n} \sum_{k=0}^{len-1} src[k]^* \cdot src(k+n), \quad 0 \le n < nLag \qquad \text{(unbiased)}$$

$$src(k) = \begin{cases} src[k], & 0 \le k < len \\ 0, & otherwise \end{cases}$$

**Application Note:** The auto-correlation estimates are computed only for positive lags, since the auto-correlation for a negative lag value is the complex conjugate of the auto-correlation for the equivalent positive lag.

### Related Topics

| | |
|---|---|
| `CrossCorr` | Estimates the cross-correlation of two vectors. |

## CrossCorr

*Estimates the cross-correlation of two vectors.*

```
void nspsCrossCorr(const float *srcA, int lenA, const float *srcB,
      int lenB, float *dst, int loLag, int hiLag);
      /* real values; single precision */
void nspcCrossCorr(const SCplx *srcA, int lenA, const SCplx *srcB,
      int lenB, SCplx *dst, int loLag, int hiLag);
      /* complex values; single precision */
void nspscCrossCorr(const float *srcA, int lenA, const SCplx *srcB,
      int lenB, SCplx *dst, int loLag, int hiLag);
      /* real and complex input vectors; complex output;
         single precision */
void nspcsCrossCorr(const SCplx *srcA, int lenA, const float *srcB,
      int lenB, SCplx *dst, int loLag, int hiLag);
      /* complex and real input vectors; complex output;
         single precision */
void nspdCrossCorr(const double *srcA, int lenA, const double *srcB,
      int lenB, double *dst, int loLag, int hiLag);
      /* real values; double precision */
void nspzCrossCorr(const DCplx *srcA, int lenA, const DCplx *srcB,
      int lenB, DCplx *dst, int loLag, int hiLag);
      /* complex values; double precision */
void nspdzCrossCorr(const double *srcA, int lenA, const DCplx *srcB,
      int lenB, DCplx *dst, int loLag, int hiLag);
      /* real and complex input vectors; complex output;
         double precision */
```

```
void nspzdCrossCorr(const DCplx *srcA, int lenA, const double *srcB,
       int lenB, DCplx *dst, int loLag, int hiLag);
       /* complex and real input vectors; complex output;
          double precision */
void nspwCrossCorr(const short *srcA, int lenA, const short *srcB,
       int lenB, short *dst, int loLag, int hiLag, int ScaleMode,
       int *ScaleFactor);
       /* real values; short integer */
void nspvCrossCorr(const WCplx *srcA, int lenA, const WCplx *srcB,
       int lenB, WCplx *dst, int loLag, int hiLag, int ScaleMode,
       int *ScaleFactor);
       /* complex values; short integer */
void nspwvCrossCorr(const short *srcA, int lenA, const WCplx *srcB,
       int lenB, WCplx *dst, int loLag, int hiLag, int ScaleMode,
       int *ScaleFactor);
       /* real inputs; complex output; short integer */
void nspvwCrossCorr(const WCplx *srcA, int lenA, const short *srcB,
       int lenB, WCplx *dst, int loLag, int hiLag, int ScaleMode,
       int *ScaleFactor);
       /* complex inputs; real output; short integer */
```

| | |
|---|---|
| *srcA* | Pointer to the vector *srcA[lenA]*. |
| *lenA* | The number of values in the *srcA* vector. |
| *srcB* | Pointer to the vector *srcB[lenB]*. |
| *lenB* | The number of values in the *srcB* vector. |
| *dst* | Pointer to the vector which stores the results of the estimated cross-correlation of the vectors *srcA[lenA]* and *srcB[lenB]*. |
| *loLag* | The bottom of the range of lags at which the correlation estimates should be computed. |
| *hilag* | The top of the range of lags at which the correlation estimates should be computed. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

The `nsp?CrossCorr()` function estimates the cross-correlation of the *lenA*-length vector *srcA* and the *lenB*-length vector *srcB* and stores the results in the array *dst[n]*. The resulting array *dst[n]* is defined by the equation:

$$dst[n] = \sum_{k=0}^{lenA-1} srcA[k]^{*} \cdot srcB(k + n + loLag),\ 0 \le n \le hiLag - loLag$$

$$srcB(k) = \begin{cases} srcB[k], & 0 \le k < lenB \\ 0, & otherwise \end{cases}$$

**Application Note:** The number of result elements is *hiLag* - *loLag* + 1, ranging from the estimate at a lag of *loLag* in *dst*[0] to the estimate at a lag of *hiLag* in *dst*[*hiLag*-*loLag*].

## Related Topics

| | |
|---|---|
| AutoCorr | Estimates normal, biased, or unbiased auto-correlation of a vector and stores the result in a second vector. |

*This page is left blank for double-sided printing*

*This page is left blank for double-sided printing*

# Vector Data Conversion Functions

# 4

Library function lists

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

- Extracting components from and constructing a complex vector
- Floating-point to integer and integer to floating point
- Floating-point to fixed-point and fixed-point to floating-point
- Floating-point to fixed-point and fixed-point to floating-point Optimized for specific fixed-point formats
- Cartesian to polar and polar to Cartesian coordinate conversion
- 8-bit µ-law or 8-bit A-law encoded format to linear or vice versa conversions of signal samples (companding functions).

## Complex Vector Structure Functions

This section describes the Signal Processing Library functions which extract real and imaginary components from a complex vector or construct a complex vector using its real and imaginary components.

The `nsp?bReal()`, `nsp?bImag()` functions return the real and imaginary parts of a complex vector in two separate vectors.

The `nsp?2RealToCplx()`, `nsp?CplxTo2Real()` functions construct a complex vector from real and imaginary components stored in two respective vectors.

The `nsp?bMag()`, `nsp?brMag()`, `nsp?bPhase()`, `nsp?brPhase()` functions compute the magnitude and phase of a complex vector elements.

## bReal

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

```
void nspcbReal(const SCplx *src, float *dst, int n);
      /* complex input; real output; single precision */
void nspzbReal(const DCplx *src, double *dst, int n);
      /* complex input; real output; double precision */
void nspvbReal(const WCplx *src, short *dst, int n);
      /* complex input; real output; short integer */
```

| | |
|---|---|
| `src` | Pointer to the vector `src[n]`. |
| `dst` | Pointer to the vector `dst[n]`. |
| `n` | The number of values in the vectors. |

### Discussion

The `nsp?bReal()` function returns the real part of the complex input vector `src[n]` in the vector `dst[n]`.

## bImag

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

```
void nspcbImag(const SCplx *src, float *dst, int n);
      /* complex input; real output; single precision */
void nspzbImag(const DCplx *src, double *dst, int n);
      /* complex input; real output; double precision */
void nspvbImag(const WCplx *src, short *dst, int n);
      /* complex input; real output; short integer */
```

| | |
|---|---|
| `src` | Pointer to the vector `src[n]`. |

*dst* Pointer to the vector *dst[n]*.

*n* The number of values in the vectors.

### Discussion

The `nsp?bImag()` function returns the imaginary part of the complex input vector *src[n]* in the vector *dst[n]*.

## b2RealToCplx

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

```
void nspcb2RealToCplx(const float *srcReal, const float *srcImag,
      SCplx *dst, int n);
      /* real inputs; complex output; single precision */
void nspzb2RealToCplx(const double *srcReal, const double *srcImag,
      DCplx *dst, int n);
      /* real inputs; complex output; double precision */
void nspvb2RealToCplx(const short *srcReal, const short *srcImag,
      WCplx *dst, int n);
      /* real inputs; complex output; short integer */
```

*srcReal* Pointer to the vector *srcReal[n]*.

*srcImag* Pointer to the vector *srcImag[n]*.

*dst* Pointer to the vector *dst[n]*.

*n* The number of values in the vectors.

### Discussion

The `nsp?b2RealToCplx()` function returns the complex vector *dst* constructed from the real and imaginary parts of the input vectors *srcReal[n]* and *srcImag[n]*. If *srcReal* is `NULL`, the real component of the vector is set to zero. If *srcImag* is `NULL`, the imaginary component of the vector is set to zero.

## bCplxTo2Real

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

```
void nspcbCplxTo2Real(const SCplx *src, float *dstReal,
      float *dstImag, int n);
      /* complex input; real outputs; single precision */
void nspzbCplxTo2Real(const DCplx *src, float *dstReal,
      float *dstImag, int n);
      /* complex input; real outputs; double precision */
void nspvbCplxTo2Real(const WCplx *src, short *dstReal,
      short *dstImag,  int n);
      /* complex input; real outputs; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[n]*. |
| *dstReal* | Pointer to the vector *dstReal[n]*. |
| *dstImag* | Pointer to the vector *dstImag[n]*. |
| *n* | The number of values in the vectors. |

### Discussion

The `nsp?bCplxTo2Real()` function returns the real and imaginary parts of the complex input vector *src[n]* in two output vectors *dstReal[n]* and *dstImag[n]*.

## bMag

*Returns the magnitudes of elements of a complex vector in a second vector.*

```
void nspcbMag(const SCplx *src, float *mag, int n);
      /* complex input; real output; single precision */
```

```
void nspzbMag(const DCplx *src, double *mag, int n);
      /* complex input; real output; double precision */
void nspvbMag(const WCplx *src, short *mag, int n, int ScaleMode,
      int *ScaleFactor);
      /* complex input; real output; short integer */
```

| | |
|---|---|
| `src` | Pointer to the vector `src[n]`. |
| `mag` | Pointer to the vector `mag[n]`. |
| `n` | The number of values in the vectors. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bMag()` function returns the magnitudes of elements of the complex input vector `src[n]` in the vector `mag[n]`.

## brMag

*Computes the magnitudes of elements of a complex vector whose real and imaginary components are specified in two vectors and stores the results in a third vector.*

```
void nspsbrMag(const float *srcReal, const float *srcImag,
      float *mag, int n); /* real values; single precision */
void nspdbrMag(const double *srcReal, const double *srcImag,
      double *mag, int n); /* real values; double precision */
void nspwbrMag(const short *srcReal, const short *srcImag,
      short *mag, int n, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| `srcReal` | Pointer to the vector `srcReal[n]`. |
| `srcImag` | Pointer to the vector `srcImag[n]`. |

| | |
|---|---|
| `mag` | Pointer to the vector `mag[n]`. |
| `n` | The number of values in the vectors. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?brMag()` computes the magnitudes of elements of the complex input vector whose real and imaginary components are specified in the vectors `srcReal[n]` and `SrcImag[n]`, respectively. It returns the results in the vector `mag[n]`.

## bPhase

*Returns the phase angles of elements of complex input vector in a second vector.*

```
void nspcbPhase(const SCplx *src, float *phase, int n);
      /* complex input; real output; single precision */
void nspzbPhase(const DCplx *src, double *phase, int n);
      /* complex input; real output; double precision */
void nspvbPhase(const WCplx *src, short *phase, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex input; real output; short integer */
```

| | |
|---|---|
| `src` | Pointer to the vector `src[n]`. |
| `phase` | Pointer to the vector `phase[n]`. |
| `n` | The number of values in the vectors. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?bPhase()` function returns the phase angles of elements of the complex input vector *src[n]* in the array *phase[n]*. Phase values are returned in radians and are in the range (-π, π].

## brPhase

*Computes the phase angles of elements of a complex vector whose real and imaginary components are specified in two vectors and stores the results in a third vector.*

```
void nspsbrPhase(const float *srcReal, const float *srcImag,
      float *phase, int n);
      /* real values; single precision */
void nspdbrPhase(const double *srcReal, const double *srcImag,
      double *phase, int n);
      /* real values; double precision */
void nspwbrPhase(const short *srcReal, const short *srcImag,
      short *phase, int n, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| *srcReal* | Pointer to the vector *srcReal[n]*. |
| *srcImag* | Pointer to the vector *srcImag[n]*. |
| *phase* | Pointer to the vector *phase[n]*. |
| *n* | The number of values in the vectors. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The nsp?brPhase() function computes the phase angles of elements of the complex input vector whose real and imaginary components are specified in the vectors *srcReal[n]* and *srcImag[n]*, respectively. It returns the result in the vector *phase[n]*. Phase values are returned in radians and are in the range (-π, π].

## bPowerSpectr

*Returns the power spectrum of a complex vector in a second vector.*

```
void nspcbPowerSpectr(const SCplx *src, float *spectr, int n);
      /* complex input; real output; single precision */
void nspzbPowerSpectr(const DCplx *src, double *spectr, int n);
      /* complex input; real output; double precision */
void nspvbPowerSpectr(const WCplx *src, short *spectr, int n,
      int ScaleMode, int *ScaleFactor);
      /* complex input; real output; short integer */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[n]*. |
| *spectr* | Pointer to the vector *spectr[n]*. |
| *n* | The number of values in the vectors. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The nsp?bPowerSpectr() function returns the power spectrum of the complex input vector *src[n]* in the vector *spectr[n]*. The spectrum elements are squares of the magnitudes of the complex input vector elements:

$$spectr[k] = (\mathrm{Re}\ src[k])^2 + (\mathrm{Im}\ src[k])^2 .$$

The magnitudes are returned by the bMag function.

# brPowerSpectr

*Computes the power spectrum of a complex vector whose real and imaginary components are specified in two vectors and stores the results in a third vector.*

```
void nspsbrPowerSpectr(const float *srcReal, const float *srcImag,
      float *spectr, int n); /* real values; single precision */
void nspdbrPowerSpectr(const double *srcReal, const double *srcImag,
      double *spectr, int n); /* real values; double precision */
void nspwbrPowerSpectr(const short *srcReal, const short *srcImag,
      short *spectr, int n, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| `srcReal` | Pointer to the vector `srcReal[n]`. |
| `srcImag` | Pointer to the vector `srcImag[n]`. |
| `spectr` | Pointer to the vector `spectr[n]`. |
| `n` | The number of values in the vectors. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

The `nsp?brPowerSpectr()` computes the power spectrum of the complex input vector whose real and imaginary components are specified in the vectors `srcReal[n]` and `SrcImag[n]`, respectively. It returns the results in the vector `spectr[n]`. The power spectrum elements are squares of the magnitudes of the complex input vector elements:

$$\texttt{spectr[k]} = (\texttt{srcReal[k]})^2 + (\texttt{srcImag[k]})^2 .$$

The magnitudes are returned by the brMag function.

# Data Type Conversion Functions

This section describes the Signal Processing Library functions that perform floating-point to integer (and reverse) and floating-point to fixed-point (and reverse) data type conversion for vectors.

## Flags Argument

The data type conversion functions require you to specify the conditions of conversion of the floating-point data to the resulting integer or fixed-point data. Specify these conditions in the `flags` argument.

The `flags` argument is evaluated as the bitwise-OR of the values you supply. The values you can use for the `flags` argument are listed in Table 4-1.

**Table 4-1 Value for the flags Argument for Data Type Conversion Functions**

| Value | Description |
|---|---|
| `NSP_Round` | Specifies that floating-point values must be rounded to the nearest integer. |
| `NSP_TruncZero` | Specifies that floating-point values must be truncated toward zero. |
| `NSP_TruncNeg` | Specifies that floating-point values must be truncated toward negative infinity. |
| `NSP_Unsigned` | Specifies that integer or fixed-point values are unsigned.<br>If this flag is not present, signed integer or fixed-point format is assumed. |

continued ☞

**Table 4-1 Value for the flags Argument for Data Type Conversion Functions**

| Value | Description |
|---|---|
| `NSP_Clip` | Specifies that floating-point values outside the allowable integer or fixed-point range are “clipped.” The allowable range is derived from the number of bits for integer data or the numbers of integer and fractional bits for the fixed-point data and the NSP_Unsigned flag described above. Clipping the floating-point values means that they are saturated to the maximum (or minimum) possible integer or fixed-point value. If this flag is not present, the values returned for floating-point numbers outside the allowable range are undefined. |
| `NSP_OvfErr` | Specifies that an overflow error should be signaled with a call to nspError() after conversion is complete, if floating-point values outside the allowable integer or fixed-point range are encountered. Note that the error is detected regardless of whether the offending values are clipped with the NSP_Clip flag (see the `NSP_Clip` value discussion above). |

Table 4-4 lists the allowable integer and fixed-point value ranges corresponding to the floating-point values to be converted. The ***`fractBits`*** variable used in Table 4-4 corresponds to the number of fractional bits.

The table presents approximate values. The precise values depend on the ***`flags`*** settings and may differ by a value that corresponds to 1 or 1/2 of the lowest bit.

**Table 4-2 Allowable Integer and Fixed-Point Value Ranges for Floating-Point Conversion**

| Function | Word Size | Minimum Value | Maximum Value |
|---|---|---|---|
| `bFloatToInt` | 8<br>16<br>32 | `MINCHAR`<br>`MINSHRT`<br>`MINLONG` | `MAXCHAR`<br>`MAXSHRT`<br>`MAXLONG` |
| `bFloatToFix` | 8<br><br>16<br>32 | `MINCHAR`/<br>pow(2, *`fractBits`*)<br>`MINSHRT`/<br>pow(2, *`fractBits`*)<br>`MINLONG`/<br>pow(2, *`fractBits`*) | `MAXCHAR`/<br>pow(2, *`fractBits`*)<br>`MAXSHRT`/<br>pow(2, *`fractBits`*)<br>`MAXLONG`/<br>pow(2, *`fractBits`*) |

## bFloatToInt

*Converts the floating-point data of a vector to integer data and stores the results in a second vector.*

```
void nspsbFloatToInt(const float *src, void *dst, int len,
      int wordSize, int flags);
      /* real values; single precision */
void nspdbFloatToInt(const double *src, void *dst, int len,
      int wordSize, int flags);
      /* real values; double precision */
```

| | |
|---|---|
| *`src`* | Pointer to the vector *`src[len]`*. |
| *`len`* | The number of values in the *`src[len]`* vector. |

| | |
|---|---|
| *dst* | Pointer to the vector which stores the results of the conversion to integer data. The type of the vector *dst[len]* is void to support different integer word sizes. |
| *wordSize* | The size of an integer word in bits; must be 8, 16, or 32. |
| *flags* | Specifies how conversion must be performed. See Table 4-1 for the *flags* values. |

## Discussion

The nsp?bFloatToInt() function converts the floating-point data in the *src[len]* to integer data, and stores the results in the vector *dst[len]*.

The argument *flags* consists of the bitwise-OR of one or more of the flags described in Table 4-1. One of the NSP_Round, NSP_TruncZero, or NSP_TruncNeg flags must be specified.

**Application Note:** Internally, the 8-, 16-, and 32-bit conversions should each be implemented separately for maximum performance.

## Related Topic

| | |
|---|---|
| bIntToFloat | Converts the integer data of a vector to the floating-point data. Stores the results in a second vector. |
| bFloatToFix | Converts the floating-point data to fixed-point data. If in the floating-point value the number of fractional bits is equal to 0, this function produces the same result as bFloatToInt. |

## bIntToFloat

*Converts the integer data of a vector to floating-point data and stores the results in a second vector.*

```
void nspsbIntToFloat(const void *src, float *dst, int len,
      int wordSize, int flags);
      /* real values; single precision */
void nspdbIntToFloat(const void *src, double *dst, int len,
      int wordSize, int flags);
      /* real values; double precision */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[len]*. The type of the vector *src[len]* is `void` to support different integer word sizes. |
| *len* | The number of values in the *src[len]* vector. |
| *dst* | Pointer to the vector which stores the results of the conversion to the floating-point data. |
| *wordSize* | The size of an integer in bits; must be 8, 16, or 32. |
| *flags* | A flag which can be `NSP_Unsigned` or `Nsp_Noflags`: |
| | `NSP_Unsigned` Specifies that integer values are unsigned. If this flag is not present, signed integer format is assumed. |
| | `NSP_Noflags` Self-explanatory. |

### Discussion

The `nsp?IntToFloat()` function converts the integer data in the vector *src[n]* to the floating-point data and stores the results in the vector *dst[len]*.

**Application Note:** Internally, the 8-, 16-, and 32-bit conversions should each be implemented separately for maximum performance.

### Related Topic

| | |
|---|---|
| `bFloatToInt` | Converts the floating-point data in a vector to integer and stores the results in a second vector. |
| `bFixToFloat` | Converts the fixed-point data to floating-point data. If in the floating-point value the number of fractional bits is equal to 0, this function produces the same result as `bIntToFloat`. |

## bFloatToFix

*Converts the floating-point data of a vector to fixed-point data and stores the results in a second vector.*

```
void nspsbFloatToFix(const float *src, void *dst, int len,
      int wordSize, int fractBits, int flags);
      /* real values; single precision */
void nspdbFloatToFix(const double *src, void *dst, int len,
      int wordSize, int fractBits, int flags);
      /* real values; double precision */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[len]*. |
| *len* | The number of values in the *src[len]* vector. |
| *dst* | Pointer to the vector which stores the results of the conversion to fixed-point data. The type of the vector *dst[len]* is `void` to support different fixed-point word sizes. |
| *wordSize* | The size of a fixed-point word in bits; must be 8, 16, or 32. |
| *fractBits* | The number of fractional bits in the desired fixed-point format. It can have a maximum value of *wordSize* for unsigned fixed-point format or *wordSize* -1 for signed fixed-point format (see the description of the value |

| | |
|---|---|
| | `NSP_Unsigned` in the *`flags`* argument discussion) and a minimum value of zero. When *`fractBits`* is zero, the fixed-point format reduces to integer format. In this case, the `nsp?bFloatToInt()` must be used to ensure a better performance. |
| *`flags`* | Specifies how conversion must be performed. See Table 4-1 for the *`flags`* values. |

## Discussion

The `nsp?bFloatToFix()` function converts the floating-point data in the vector *`src[len]`* to the fixed-point data, and stores the results in the vector *`dst[len]`*.

The argument *`flags`* consists of the bitwise-OR of one or more of the flags described in Table 4-1. One of the `NSP_Round`, `NSP_TruncZero`, or `NSP_TruncNeg` flags must be specified.

**Application Notes:** Internally, the 8-, 16-, and 32-bit conversions should each be implemented separately for maximum performance.

For signed, purely fractional fixed-point formats, and for S15.16 format (that is, sign bit, 15 integer bits, and 16 fractional bits), special optimized functions are available and should be used instead of `nsp?bFloatToFix()` for maximum performance. See the "Related Topics" section that follows.

## Related Topics

| | |
|---|---|
| `bFixToFloat` | Converts the fixed-point data of a vector to floating-point data. Stores the results in a second vector. |
| `bFloatToS31Fix`, `bFloatToS15Fix`, `bFloatToS7Fix`, `bFloatToS1516Fix` | Perform the optimized functions used for signed, purely fractional fixed-point formats |

## bFixToFloat

*Converts the fixed-point data of a vector to the floating-point data and stores the results in a second vector.*

```
void nspsbFixToFloat(const void *src, float *dst, int len,
      int wordSize, int fractBits, int flags);
      /* real values; single precision */
void nspdbFixToFloat(const void *src, double *dst, int len,
      int wordSize, int fractBits, int flags);
      /* real values; double precision */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[len]*. The type of the vector *src[len]* is `void` to support different fixed-point word sizes. |
| *len* | The number of values in the *src[len]* vector. |
| *dst* | Pointer to the vector which stores the results of the conversion to the floating-point data. |
| *wordSize* | The bit size of a fixed-point word; must be 8, 16, or 32. |
| *fractBits* | The number of fractional bits in the desired fixed-point format. It can have a maximum value of *wordSize* for unsigned fixed-point format or *wordSize* -1 for signed fixed-point format (see the description of the value `NSP_Unsigned` in the *flags* argument discussion) and a minimum value of zero. When *fractBits* is zero, the fixed-point format reduces to integer format. In this case, the `nsp?bIntToFloat()` must be used to ensure a better performance. |
| *flags* | A flag which can be `NSP_Unsigned` or `NSP_Noflags`.<br>`NSP_Unsigned` — Specifies that fixed-point values are unsigned. If this flag is not present, signed fixed-point format is assumed. |

### Discussion

The `nsp?FixToFloat()` function converts the fixed-point data in the vector *`src[len]`* to the floating-point data and stores the results in the vector *`dst[len]`*.

**Application Notes:** Internally, the 8-, 16-, and 32-bit conversions should each be implemented separately for maximum performance.

For signed, purely fractional fixed-point formats, and for S15.16 format (that is, sign bit, 15 integer bits, and 16 fractional bits), special optimized functions are available and should be used instead of `nsp?bFixToFloat()` for maximum performance. See the “Related Topics” section that follows.

### Related Topics

| | |
|---|---|
| `bFloatToFix` | Converts the floating-point data of a vector to fixed-point data. Stores the results in a second vector. |
| `bS31FixToFloat`, `bS15FixToFloat`, `bS7FixToFloat`, `bS1516FixToFloat` | Perform the optimized functions used for signed, purely fractional fixed-point formats. |

## Optimized Data Type Conversion Functions

This section describes the Signal Processing Library functions that perform floating-point to fixed-point (and reverse) data type conversion. These conversion operations are optimized for signed, purely fractional fixed-point vector formats.

The fixed-point formats assumed for these functions are:

- S.31—a sign bit and 31 fractional bits
- S15.16—a sign bit, 15 integer bits, and 16 fractional bits
- S.15—a sign bit and 15 fractional bits
- S.7—a sign bit and 7 fractional bits

## Flags Argument

The optimized data type conversion functions require you to specify the conditions of conversion of the floating-point data to the resulting fixed-point data. Specify these conditions in the *`flags`* argument. The *`flags`* argument is evaluated as the bitwise-OR of the values you supply. The values you can use for the *`flags`* argument are listed in Table 4-3.

**Table 4-3 Value for the flags Argument for Optimized Data Type Conversion Functions**

| Value | Description |
|---|---|
| `NSP_Round` | Specifies that floating-point values must be rounded to the nearest integer. |
| `NSP_TruncZero` | Specifies that floating-point values must be truncated toward zero. |
| `NSP_TruncNeg` | Specifies that floating-point values must be truncated toward negative infinity. |
| `NSP_Clip` | Specifies that floating-point values outside the allowable fixed-point range are “clipped.” The allowable range is derived from the numbers of integer and fractional bits for the optimized fixed-point formats described above. Clipping the floating-point values means that they are saturated to the maximum (or minimum) possible fixed-point value. If this flag is not present, the values returned for floating-point numbers outside the allowable range are undefined. |
| `NSP_OvfErr` | Specifies that an overflow error should be signaled with a call to nspError() after conversion is complete, if floating-point values outside the allowable integer or fixed-point range are encountered. Note that the error is detected regardless of whether the offending values are clipped with the NSP_Clip flag (see the `NSP_Clip` value discussion above). |

Table 4-4 lists the allowable integer and fixed-point value ranges corresponding to the floating-point values to be converted.

Table 4-4 presents approximate values. The precise values depend on the *flags* settings and may differ by a value that corresponds to 1 or 1/2 of the lowest bit.

**Table 4-4 Allowable Integer and Fixed-Point Value Ranges for Floating-Point Optimized Conversion**

| Function | Minimum Value | Maximum Value |
|---|---|---|
| bFloatToS7Fix | -1.0 | 1.0 |
| bFloatToS15Fix | -1.0 | 1.0 |
| bFloatToS31Fix | -1.0 | 1.0 |
| bFloatToS1516Fix | MINSHRT | MAXSHRT |

## bFloatToS31Fix

*Converts the floating-point data of a vector to S.31 fixed-point data and stores the results in a second vector.*

```
void nspsbFloatToS31Fix(const float *src, long *dst, int len,
      int flags); /* real values; single precision */
void nspdbFloatToS31Fix(const double *src, long *dst, int len,
      int flags); /* real values; double precision */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[len]*. |
| *len* | The number of values in the *src[len]* vector. |
| *dst* | Pointer to the vector which stores the results of the conversion to fixed-point data. |
| *flags* | Specifies how conversion must be performed. See Table 4-1 for the *flags* values. |

### Discussion

The `nsp?bFloatToS31Fix()` function converts the floating-point data in the vector *`src[len]`* to the fixed-point data, and stores the results in the vector *`dst[len]`*. This function assumes a fixed-point format of S.31, that is, a sign bit and 31 fractional bits.

The argument *`flags`* consists of the bitwise-OR of one or more of the flags described in Table 4-1. One of the `NSP_Round`, `NSP_TruncZero`, or `NSP_TruncNeg` flags must be specified.

### Related Topics

| | |
|---|---|
| `bFloatToS15Fix` | Converts the floating-point data of a vector to S.15 fixed-point data. Stores the results in a second vector. |
| `bFloatToS7Fix` | Converts the floating-point data of a vector to S.7 fixed-point data. Stores the results in a second vector. |
| `bFloatToS1516Fix` | Converts the floating-point data of a vector to S15.16 fixed-point data. Stores the results in a second vector. |

## bS31FixToFloat

*Converts the S.31 fixed-point data of a vector to floating point and stores the result in a second vector.*

```
void nspsbS31FixToFloat(const long *src, float *dst, int len);
      /* real values; single precision */
void nspdbS31FixToFloat(const long *src, double *dst, int len);
      /* real values; double precision */
```

| | |
|---|---|
| *`src`* | Pointer to the vector *`src[len]`*. |
| *`len`* | The number of values in the *`src[len]`* vector. |
| *`dst`* | Pointer to the vector which stores the results of the conversion to floating-point data. |

### Discussion

The `nsp?bS31FixToFloat()` function converts the fixed-point data in the vector *`src[len]`* to the floating-point data, and stores the results in the vector *`dst[len]`*. This function assumes a fixed-point format of S.31, that is, a sign bit and 31 fractional bits.

### Related Topics

| | |
|---|---|
| `bS15FixToFloat` | Converts the S.15 fixed-point data of a vector to floating-point. Stores the results in a second vector. |
| `bS7FixToFloat` | Converts the S.7 fixed-point data of a vector to floating-point. Stores the results in a second vector. |
| `bS1516FixToFloat` | Converts the S15.16 fixed-point data of a vector to floating-point. Stores the results in a second vector. |

## bFloatToS1516Fix

*Converts the floating-point data of a vector to S15.16 fixed-point data and stores the results in a second vector.*

```
void nspsbFloatToS1516Fix(const float *src, long *dst, int len,
      int flags); /* real values; single precision */
void nspdbFloatToS1516Fix(const double *src, long *dst, int len,
      int flags); /* real values; double precision */
```

| | |
|---|---|
| *`src`* | Pointer to the vector *`src[len]`*. |
| *`len`* | The number of values in the *`src[len]`* vector. |
| *`dst`* | Pointer to the vector which stores the results of the conversion to fixed-point data. |
| *`flags`* | Specifies how conversion must be performed. See Table 4-1 for the *`flags`* values. |

### Discussion

The nsp?bFloatToS1516Fix() function converts the floating-point data in the vector *src[len]* to the fixed-point data, and stores the results in the vector *dst[len]*. This function assumes a fixed-point format of S15.16, that is, a sign bit, 15 integer bits, and 16 fractional bits.

The argument *flags* consists of the bitwise-OR of one or more of the flags described in Table 4-1. One of the NSP_Round, NSP_TruncZero, or NSP_TruncNeg flags must be specified.

### Related Topics

| | |
|---|---|
| bFloatToS31Fix | Converts the floating-point data of a vector to S.31 fixed-point data. Stores the results in a second vector. |
| bFloatToS7Fix | Converts the floating-point data of a vector to S.7 fixed-point data. Stores the results in a second vector. |
| bFloatToS15Fix | Converts the floating-point data of a vector to S.15 fixed-point data. Stores the results in a second vector. |

## bS1516FixToFloat

*Converts the S15.16 fixed-point data of a vector to floating point and stores the result in a second vector.*

```
void nspsbS1516FixToFloat(const long *src, float *dst, int len);
      /* real values; single precision */
void nspdbS1516FixToFloat(const long *src, double *dst, int len);
      /* real values; double precision */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[len]*. |
| *len* | The number of values in the *src[len]* vector. |
| *dst* | Pointer to the vector which stores the results of the conversion to floating-point data. |

### Discussion

The `nsp?bS1516FixToFloat()` function converts the fixed-point data in the vector *`src[len]`* to the floating-point data, and stores the results in the vector *`dst[len`*. This function assumes a fixed-point format of S15.16, that is, a sign bit, 15 integer bits, and 16 fractional bits.

### Related Topics

| | |
|---|---|
| `bS31FixToFloat` | Converts the S.31 fixed-point data of a vector to floating-point. Stores the results in a second vector. |
| `bS7FixToFloat` | Converts the S.7 fixed-point data of a vector to floating-point. Stores the results in a second vector. |
| `bS15FixToFloat` | Converts the S.15 fixed-point data of a vector to floating-point. Stores the results in a second vector. |

## bFloatToS15Fix

*Converts the floating-point data of a vector to S.15 fixed-point data and stores the results in a second vector.*

```
void nspsbFloatToS15Fix(const float *src, short *dst, int len,
      int flags); /* real values; single precision */
void nspdbFloatToS15Fix(const double *src, short *dst, int len,
      int flags); /* real values; double precision */
```

| | |
|---|---|
| *`src`* | Pointer to the vector *`src[len]`*. |
| *`len`* | The number of values in the *`src[len]`* vector. |
| *`dst`* | Pointer to the vector which stores the results of the conversion to fixed-point data. |
| *`flags`* | Specifies how conversion must be performed. See Table 4-1 for the *`flags`* values. |

### Discussion

The `nsp?bFloatToS15Fix()` function converts the floating-point data in the vector *`src[len]`* to the fixed-point data, and stores the results in the vector *`dst[len]`*. This function assumes a fixed-point format of S.15, that is, a sign bit and 15 fractional bits.

The argument *`flags`* consists of the bitwise-OR of one or more of the flags described in Table 4-1. One of the `NSP_Round`, `NSP_TruncZero`, or `NSP_TruncNeg` flags must be specified.

### Related Topics

| | |
|---|---|
| `bFloatToS31Fix` | Converts the floating-point data of a vector to S.31 fixed-point data. Stores the results in a second vector. |
| `bFloatToS7Fix` | Converts the floating-point data of a vector to S.7 fixed-point data. Stores the results in a second vector. |
| `bFloatToS1516Fix` | Converts the floating-point data of a vector to S15.16 fixed-point data. Stores the results in a second vector. |

## bS15FixToFloat

*Converts the S.15 fixed-point data of a vector to floating point and stores the result in a second vector.*

```
void nspsbS15FixToFloat(const short *src, float *dst, int len);
      /* real values; single precision */
void nspdbS15FixToFloat(const short *src, double *dst, int len);
      /* real values; double precision */
```

| | |
|---|---|
| *`src`* | Pointer to the vector *`src[len]`*. |
| *`len`* | The number of values in the *`src[len]`* vector. |
| *`dst`* | Pointer to the vector which stores the results of the conversion to floating-point data. |

### Discussion

The `nsp?bS15FixToFloat()` function converts the fixed-point data in the vector *`src[len]`* to the floating-point data, and stores the results in the vector *`dst[len]`*. This function assumes a fixed-point format of S.15, that is, a sign bit and 15 fractional bits.

### Related Topics

| | |
|---|---|
| `bS31FixToFloat` | Converts the S.31 fixed-point data of a vector to floating-point. Stores the results in a second vector. |
| `bS7FixToFloat` | Converts the S.7 fixed-point data of a vector to floating-point. Stores the results in a second vector. |
| `bS1516FixToFloat` | Converts the S15.16 fixed-point data of a vector to floating-point. Stores the results in a second vector. |

## bFloatToS7Fix

*Converts the floating-point data of a vector to S.7 fixed-point data and stores the results in a second vector.*

```
void nspsbFloatToS7Fix(const float *src, char *dst, int len,
      int flags); /* real values; single precision */
void nspdbFloatToS7Fix(const double *src, char *dst, int len,
      int flags); /* real values; double precision */
```

| | |
|---|---|
| *`src`* | Pointer to the vector *`src[len]`*. |
| *`len`* | The number of values in the *`src[len]`* vector. |
| *`dst`* | Pointer to the vector which stores the results of the conversion to fixed-point data. |
| *`flags`* | Specifies how conversion must be performed. See Table 4-1 for the *`flags`* values. |

### Discussion

The `nsp?bFloatToS7Fix()` function converts the floating-point data in the vector *`src[len]`* to the fixed-point data, and stores the results in the vector *`dst[len]`*. This function assumes a fixed-point format of S.7, that is, a sign bit and 7 fractional bits.

The argument *`flags`* consists of the bitwise-OR of one or more of the flags described in Table 4-1. One of the `NSP_Round`, `NSP_TruncZero`, or `NSP_TruncNeg` flags must be specified.

### Related Topics

| | |
|---|---|
| `bFloatToS15Fix` | Converts the floating-point data of a vector to S.15 fixed-point data. Stores the results in a second vector. |
| `bFloatToS31Fix` | Converts the floating-point data of a vector to S.31 fixed-point data. Stores the results in a second vector. |
| `bFloatToS1516Fix` | Converts the floating-point data of a vector to S15.16 fixed-point data. Stores the results in a second vector. |

## bS7FixToFloat

*Converts the S.7 fixed-point data of a vector to floating point and stores the result in a second vector.*

```
void nspsbS7FixToFloat(const char *src, float *dst, int len);
      /* real values; single precision */
void nspdbS7FixToFloat(const char *src, double *dst, int len);
      /* real values; double precision */
```

| | |
|---|---|
| *`src`* | Pointer to the vector *`src[len]`*. |
| *`len`* | The number of values in the *`src[len]`* vector. |
| *`dst`* | Pointer to the vector which stores the results of the conversion to floating-point data. |

### Discussion

The `nsp?bS7FixToFloat()` function converts the fixed-point data in the *`src[len]`* to the floating-point data, and stores the results in the vector *`dst[len]`*. This function assumes a fixed-point format of S.7, that is, a sign bit and 7 fractional bits.

### Related Topics

| | |
|---|---|
| `bS15FixToFloat` | Converts the S.15 fixed-point data of a vector to floating-point. Stores the results in a second vector. |
| `bS31FixToFloat` | Converts the S.31 fixed-point data of a vector to floating-point. Stores the results in a second vector. |
| `bS1516FixToFloat` | Converts the S15.16 fixed-point data of a vector to floating-point. Stores the results in a second vector. |

## Coordinate Conversion Functions

This section describes the Signal Processing Library functions that perform Cartesian to polar and polar to Cartesian coordinate conversion for vectors.

## bCartToPolar

*Converts the elements of a complex vector to polar coordinate form.*

```
void nspcbCartToPolar(const SCplx *src, float *mag, float *phase,
      int len); /* complex input; real output; single precision */

void nspzbCartToPolar(const DCplx *src, double *mag, double *phase,
      int len); /* complex input; real output; double precision */
```

| | |
|---|---|
| *src* | Pointer to the vector *src[len]*. |
| *mag* | Pointer to the vector *mag[len]* which stores the magnitude (radius) components of the elements of vector *src[len]*. |
| *phase* | Pointer to the vector *phase[len]* which stores the phase (angle) components of the elements of vector *srclen]*. Phase values are in the range (-π,π]. |
| *len* | The number of values in the vectors. |

## Discussion

The `nsp?bCartToPolar()` function converts the elements of a complex input vector *src[len]* to polar coordinate form, storing the magnitude (radius) component of each element in the vector *mag[len]* and the phase (angle) component of each element in the vector *phase[len]*.

## Related Topics

| | |
|---|---|
| `bPolarToCart` | Converts the polar form magnitude/phase pairs stored in input vectors into a complex vector. Stores the result in a third vector. |
| `brCartToPolar` | Converts the complex real/imaginary pairs of input vectors to polar coordinate form. Stores the magnitude and phase components of each element in two respective vectors. |
| `brPolarToCart` | Converts the polar form magnitude/phase pairs stored in two input vectors into a complex vector. Stores the real and imaginary components of the result in two respective vectors. |

## brCartToPolar

*Converts the complex real/imaginary pairs of input vectors to polar coordinate form.*

```
void nspsbrCartToPolar(const float *srcReal, const float *srcImag,
      float *mag, float *phase, int len);
      /* real values; single precision */
void nspdbrCartToPolar(const double *srcReal, const double *srcImag,
      double *mag, double *phase, int len);
      /* real values; double precision */
```

| | |
|---|---|
| *srcReal* | Pointer to the vector *srcReal[len]* which stores the real components of Cartesian X/Y pairs. |
| *srcImag* | Pointer to the vector *srcImag[len]* which stores the imaginary components of Cartesian X/Y pairs. |
| *mag* | Pointer to the vector *mag[len]* which stores the magnitude (radius) components of the elements in polar coordinate form. |
| *phase* | Pointer to the vector *phase[len]* which stores the phase (angle) components of the elements in polar coordinate form. Phase values are in the range $(-\pi,\pi]$. |
| *len* | The number of values in the vectors. |

### Discussion

The `nsp?brCartToPolar()` function converts the complex real/imaginary (Cartesian coordinate X/Y) pairs of the input vectors *srcReal[len]* and *srcImag[len]* to polar coordinate form, storing the magnitude (radius) component of each element in the vector *mag[len]* and the phase (angle) component of each element in the vector *phase[len]*.

### Related Topics

| | |
|---|---|
| brPolarToCart | Converts the polar coordinate form magnitude/phase pairs stored in two input vectors into a complex vector. Stores the real and imaginary components of the result in two respective vectors. |
| bCartToPolar | Converts the elements of the complex vector to polar coordinate form. Stores the magnitude and phase components of each element in two respective vectors. |
| bPolarToCart | Converts the polar coordinate form magnitude/phase pairs stored in input vectors into a complex vector. Stores the result in a third vector. |

## bPolarToCart

*Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.*

```
void nspcbPolarToCart(const float *mag, const float *phase,
      SCplx *dst, int len);
      /* real input; complex output; single precision */
void nspzbPolarToCart(const double *mag, const double *phase,
      DCplx *dst, int len);
      /* real input; complex output; double precision */
```

| | |
|---|---|
| *mag* | Pointer to the vector *mag[len]* which stores the magnitude (radius) components of the elements. |
| *phase* | Pointer to the vector *phase[len]* which stores the phase (angle) components of the elements. |
| *dst* | Pointer to the resulting vector *dst[len]* which stores the complex values consisting of magnitude (radius) and phase (angle). |
| *len* | The number of values in the vectors. |

### Discussion

The `nsp?bPolarToCart()` function converts the polar form magnitude/phase pairs stored in the input vectors *mag[len]* and *phase[len]* into a complex vector and stores the result in the vector *dst[len]*.

### Related Topics

| | |
|---|---|
| bCartToPolar | Converts the elements of the complex vector to polar coordinate form. Stores the magnitude and phase components of each element in two respective vectors. |
| brCartToPolar | Converts the complex real/imaginary pairs of input vectors to polar coordinate form. Stores the magnitude and phase components of each element in two respective vectors. |
| brPolarToCart | Converts the polar form magnitude/phase pairs stored in two input vectors into a complex vector. Stores the real and imaginary components of the result in two respective vectors. |

## brPolarToCart

*Converts the polar form magnitude/phase pairs of input vectors to Cartesian coordinate form.*

```
void nspsbrPolarToCart(const float *mag, const float *phase,
      float *dstReal, float *dstImag, int len);
      /* real values; single precision */
void nspdbrPolarToCart(const double *mag, const double *phase,
      double *dstReal, double *dstImag, int len);
      /* real values; double precision */
```

| | |
|---|---|
| *mag* | Pointer to the vector *mag[len]* which stores the magnitude (radius) components of the elements in polar coordinate form. |
| *phase* | Pointer to the vector *phase[len]* which stores the phase (angle) components of the elements in polar coordinate form. Phase values are in the range (-π,π]. |
| *dstReal* | Pointer to the vector *dstReal[len]* which stores the real components of Cartesian X/Y pairs. |
| *dstImag* | Pointer to the vector *dstImag[len]* which stores the imaginary components of Cartesian X/Y pairs. |
| *len* | The number of values in the vectors. |

## Discussion

The `nsp?brPolarToCart()` function converts the polar form magnitude/phase pairs stored in the input vectors *mag[len]* and *phase[len]* into a complex vector and stores the real component of the result in the vector *dstReal[len]* and the imaginary component in the vector *dstImag[len]*.

## Related Topics

| | |
|---|---|
| brCartToPolar | Converts the complex real/imaginary pairs of input vectors to polar coordinate form. Stores the magnitude and phase components of each element in two respective vectors. |
| bCartToPolar | Converts the elements of the complex vector to polar coordinate form. Stores the magnitude and phase components of each element in two respective vectors. |
| bPolarToCart | Converts the polar form magnitude/phase pairs stored in input vectors into a complex vector. Stores the result in a third vector. |

# Companding Functions

The functions described in this section perform an operation of data compression by using a logarithmic encoder-decoder, referred to as companding. Companding allows you to maintain a constant percentage error by logarithmically spacing the quantization levels [Rab78].

The Signal Processing Library companding functions perform the following conversion operations of signal samples:

- From 8-bit μ-law encoded format to linear or vice-versa.
- From 8-bit A-law encoded format to linear or vice-versa.

Samples encoded in μ-law or A-law format are non-uniformly quantized. The quantization functions used by these formats are designed to reduce the dependency of signal-to-noise ratio on the magnitude of the encoded signal. This is achieved by quantizing (companding) at a finer resolution near zero, and at a coarse resolution at larger positive or negative levels. The output values are normalized to be in the range of -1 to +1.

These functions perform the μ-law and A-law companding in compliance with the CCITT G.711 specification, [CCITT]. For the conversion rules and more details, refer to [CCITT].

## bMuLawToLin

*Decodes samples from 8-bit μ-law encoded format to linear samples.*

```
void nspsbMuLawToLin (const unsigned char *src, float *dst,
      int len);
      /* real values; single precision */
void nspdbMuLawToLin (const unsigned char *src, double *dst,
      int len);
      /* real values; double precision */
void nspwbMuLawToLin (const unsigned char *src, short *dst,
      int len);
      /* real values; short integer */
```

| | |
|---|---|
| `src` | Pointer to the input unsigned char vector, which stores 8-bit µ-law encoded signal samples to be decoded. |
| `dst` | Pointer to the output vector, which stores the linear sample results. |
| `len` | The number of samples in the vector `src[len]`. |

## Discussion

The `nsp?bMuLawToLin()` function decodes the 8-bit µ-law encoded samples in the input vector `src[len]` to linear samples and stores them in the vector `dst[len]`.

The formula for µ-law companding is as follows:

$$|C_{\mu}(x)| = \frac{\ln(1 + 255 \cdot |x|)}{\ln(256)} \cdot 128, \qquad -1 \le x \le 1$$

where $x$ is the linear signal sample and $C_{\mu}(x)$ is the µ-law encoded sample. The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

**Application Notes:** The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of µ-law format is usually performed using look-up Tables 2a/G.711 and 2b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

## Related Topics

| | |
|---|---|
| `bLinToMuLaw` | Encodes the linear samples using 8-bit µ-law format. |
| `bALawToLin` | Decodes the 8-bit A-law encoded samples to linear samples. |
| `bLinToALaw` | Encodes the linear samples using 8-bit A-law format. |
| `bMuLawToALaw` | Converts samples from 8-bit µ-law encoded format to 8-bit A-law encoded format. |
| `bALawToMuLaw` | Converts samples from 8-bit A-law encoded format to 8-bit µ-law encoded format. |

## bLinToMuLaw

*Encodes the linear samples using 8-bit μ-law format and stores them in a vector.*

```
void nspsbLinToMuLaw (const float *src, unsigned char *dst,
      int len);
      /* real values; single precision */
void nspdbLinToMuLaw (const double *src, unsigned char *dst,
      int len);
      /* real values; double precision */
void nspwbLinToMuLaw (const short char *src, unsigned char *dst,
      int len);
      /* real values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector that holds the output of the `nsp?bLinToMuLaw()` function. |
| *src* | Pointer to the vector that holds the signal samples (normalized to be less than 1.0) to be encoded. |
| *len* | The number of samples in the vector *src[len]*. |

### Discussion

The `nsp?bLinToMuLaw()` function encodes the linear samples in the input vector *src[len]* using 8-bit μ-law format and stores them in the vector *dst[len]*.

### Related Topics

| | |
|---|---|
| bMuLawToLin | Decodes samples from the 8-bit μ-law encoded format to linear samples. |
| bALawToLin | Decodes samples from the 8-bit A-law encoded format to linear samples. |
| bLinToALaw | Encodes the linear samples using 8-bit A-law format. |

| | |
|---|---|
| bMuLawToALaw | Converts samples from 8-bit μ-law encoded format to 8-bit A-law encoded format. |
| bALawToMuLaw | Converts samples from 8-bit A-law encoded format to 8-bit μ-law encoded format. |

## bALawToLin

*Decodes the 8-bit A-law encoded samples to linear samples.*

```
void nspsbALawToLin (const unsigned char *src, float *dst, int len);
      /* real values; single precision */
void nspdbALawToLin (const unsigned char *src, double *dst,
      int len);
      /* real values; double precision */
void nspwbALawToLin (const unsigned char *src, short *dst,
      int len);
      /* real values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector that holds the output of the `nsp?bALawToLin()` function. |
| *src* | Pointer to the vector that holds the signal samples to be converted. |
| *len* | The number of samples in the vector *src[len]*. |

### Discussion

The `nsp?bALawToLin()` function decodes the 8-bit A-law encoded samples in the input vector *src[len]* to linear samples and stores them in the vector *dst[len]*.

The formula for A-law companding is as follows:

$$|c_A(x)| = \begin{cases} \dfrac{87.56|x|}{1 + \ln 87.56} \cdot 128, & 0 \le |x| \le \dfrac{1}{87.56} \\ \dfrac{1 + \ln(87.56|x|)}{1 + \ln 87.56} \cdot 128, & \dfrac{1}{87.56} < |x| \le 1 \end{cases}$$

where $x$ is the linear signal sample and $C_A(x)$ is the A-law encoded sample. The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

**Application Notes:** The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of A-law format is usually performed using look-up Tables 1a/G.711 and 1b/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

### Related Topics

| | |
|---|---|
| `bLinToALaw` | Encodes the linear samples using 8-bit A-law format. |
| `bMuLawToLin` | Decodes the 8-bit µ-law encoded samples to linear samples. |
| `bLinToMuLaw` | Encodes the linear samples using 8-bit µ-law format. |
| `bMuLawToALaw` | Converts samples from 8-bit µ-law encoded format to 8-bit A-law encoded format. |
| `bALawToMuLaw` | Converts samples from 8-bit A-law encoded format to 8-bit µ-law encoded format. |

## bLinToALaw

*Encodes the linear samples using 8-bit A-law format and stores them in an array.*

```
void nspsbLinToALaw (const float char *src, unsigned char *dst, int
len);
      /* real values; single precision */
void nspdbLinToALaw (const double *src, unsigned char *dst,
      int len);
      /* real values; double precision */
```

```
void nspwbLinToALaw (const short *src, unsigned char *dst,
      int len);
      /* real values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector that holds the output of the `nsp?bLinToALaw()` function. |
| *src* | Pointer to the vector that holds the signal samples to be encoded. |
| *len* | The number of samples in the vector *src[len]*. |

## Discussion

The `nsp?bLinToALaw()` function encodes the linear samples in the input vector *src[len]* using 8-bit A-law format and stores them in the vector *dst[len]*.

## Related Topics

| | |
|---|---|
| bALawToLin | Decodes the 8-bit A-law encoded samples to linear samples. |
| bMuLawToLin | Decodes the 8-bit μ-law encoded samples to linear samples. |
| bLinToMuLaw | Encodes the linear samples using 8-bit μ-law format. |
| bMuLawToALaw | Converts samples from 8-bit μ-law encoded format to 8-bit A-law encoded format. |
| bALawToMuLaw | Converts samples from 8-bit A-law encoded format to 8-bit μ-law encoded format. |

## bMuLawToALaw

*Converts samples from 8-bit μ-law encoded format to 8-bit A-law encoded format.*

```
void nspbMuLawToALaw (const unsigned char *src, unsigned char *dst,
      int len);
```

| | |
|---|---|
| *src* | Pointer to the input unsigned char vector, which stores 8-bit μ-law encoded signal samples. |
| *dst* | Pointer to the output unsigned char vector, which stores the 8-bit A-law encoded samples. |
| *len* | The number of samples in the vector *src[len]*. |

### Discussion

The `nspbMuLawToALaw()` function converts signal samples from 8-bit μ-law encoded format in the input vector *src[len]* to 8-bit A-law encoded format and stores them in the vector *dst[len]*.

**Application Notes:** The conversion of μ-law format to A-law format is usually performed using look-up Table 3/G.711shown in the CCITT specification G.711. Refer to the G.711 specification for details.

### Related Topics

| | |
|---|---|
| bMuLawToLin | Decodes the 8-bit μ-law encoded samples to linear samples. |
| bLinToMuLaw | Encodes the linear samples using 8-bit μ-law format. |
| bALawToLin | Decodes the 8-bit A-law encoded samples to linear samples. |
| bLinToALaw | Encodes the linear samples using 8-bit A-law format. |
| bALawToMuLaw | Converts samples from 8-bit A-law encoded format to 8-bit  μ-law encoded format. |

## bALawToMuLaw

*Converts samples from 8-bit A-law encoded format to 8-bit µ-law encoded format.*

```
void nspbALawToMuLaw (const unsigned char *src, unsigned char *dst,
      int len);
```

| | |
|---|---|
| *src* | Pointer to the input unsigned char vector, which stores 8-bit A-law encoded signal samples. |
| *dst* | Pointer to the output unsigned char vector, which stores the 8-bit µ-law encoded samples. |
| *len* | The number of samples in the vector *src[len]*. |

### Discussion

The `nspbMuLawToALaw()` function converts signal samples from 8-bit A-law encoded format in the input vector *src[len]* to 8-bit µ-law format and stores them in the vector *dst[len]*.

**Application Notes:** The conversion of A-law format to µ-law format is usually performed using look-up Table 4/G.711 shown in the CCITT specification G.711. Refer to the G.711 specification for details.

### Related Topics

| | |
|---|---|
| bMuLawToLin | Decodes the 8-bit µ-law encoded samples to linear samples. |
| bLinToMuLaw | Encodes the linear samples using 8-bit µ-law format. |
| bALawToLin | Decodes the 8-bit A-law encoded samples to linear samples. |
| bLinToALaw | Encodes the linear samples using 8-bit A-law format. |
| bMuLawToALaw | Converts samples from 8-bit µ-law encoded format to 8-bit A-law encoded format. |

*This page is left blank for double-sided printing*

*This page is left blank for double-sided printing*

# Sample-Generating Functions

5

This chapter describes the Intel® Signal Processing Library functions which generate the tone samples, triangle samples, pseudo-random samples with uniform distribution, and pseudo-random samples with Gaussian distribution.

## Tone-Generating Functions

Library function lists

The functions described in this section generate a tone (or “sinusoid”) of a given frequency, phase, and magnitude. Tones are fundamental building blocks for analog signals. This makes sampled tones extremely useful in signal processing systems as test signals and as building blocks for more complex signals. The tone functions are preferable to the C math library’s `sin()` function for many applications because they can use knowledge retained from the computation of the previous sample to compute the next sample much faster than `sin()` or `cos()`.

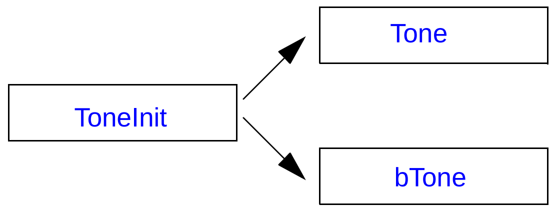
The Signal Processing Library provides functions for initializing a tone generator and functions for generating single or multiple samples from a previously initialized tone.

| | |
|---|---|
| `nsp?bTone()` | Returns a specified number of consecutive samples of the tone. |
| `nsp?Tone()` | Returns one sample of the tone each time the function is called. |
| `nsp?ToneInit()` | Initializes the `NSP?ToneState` structure with a given frequency, phase, and magnitude for the tone. |

`NSP?ToneState` Structure which contains the specified parameters for the tone.

Figure 5-1 illustrates the order of use of the tone-generating functions.

**Figure 5-1 Order of Use of the Tone-Generating Functions**




The `nsp?ToneInit()` function initializes the `NSP?ToneState` structure with a specified frequency, phase, and magnitude for the tone. The structure can then be passed to either the `nsp?bTone()` function, the `nsp?Tone()` function, or both. The `nsp?Tone()` function returns a sample of the tone each time it is called. The `nsp?bTone()` function returns a specified number of consecutive samples. These functions are described in more detail below.

Example 5-1 shows the code for generating a tone and taking its FFT.

**Example 5-1 Generating a Tone and Taking its FFT**

```
/* generate a tone
 * and take its FFT
 */
NSPZToneState ts;
DCplx     tone[256];
DCplx     tonefft[256];

nspzToneInit(0.11, NSP_DegToRad(30), 5.0, &ts);
nspzbTone(&ts, tone, 256);
nspzFftNip(tone, tonefft, 8, NSP_Forw);
```

Example 5-2 shows the code for using a tone for primary demodulation in a passband modem.

### Example 5-2 Using Generated Tones

```
/* Use a tone for primary demodulation
 * in passband modem
 */
NSPDToneState   prim_osc;

nspdToneInit(0.35, 0, 1.0, &prim_osc);
for(;;) {
      DCplx  samp;
      samp = ...; /* get analytic sample (after Hilbert filter) */
      samp = nspzMpy( nspdTone( &prim_osc), samp);
      ...
}
```

**Application Notes:** The contents of the structures `NSP?ToneState` and the particular equations used to calculate the tone are implementation-dependent. The tone is calculated using a structure that implements the following second-order transfer function:

$$X(z) = \frac{z^{-1}}{1 - \alpha z^{-1} + z^{-2}} \quad , \quad \alpha = 2\cos(2\pi \cdot rfreq)$$

This system has two complex conjugate poles on the unit circle. The angle of the poles is determined by *rfreq*. There are several possible equations to implement this system. The particular equation used is implementation-dependent because the relative speed and harmonic distortion depends on the particular processor.

Complex tones (`nspcTone()` and `nspzTone()`) are generated by using two real-valued tone oscillators that are ninety degrees out of phase.

## bTone

*Produces consecutive samples of a tone.*

```
void nspsbTone(NSPSToneState *statePtr, float *samps, int sampsLen);
      /* real values; single precision */
void nspcbTone(NSPCToneState *statePtr, SCplx *samps, int sampsLen);
      /* complex values; single precision */
void nspdbTone(NSPDToneState *statePtr, double *samps,
      int sampsLen);
      /* real values; double precision */
void nspzbTone(NSPZToneState *statePtr, DCplx *samps, int sampsLen);
      /* complex values; double precision */
void nspwbTone(NSPWToneState *statePtr, short *samps, int sampsLen);
      /* real values; short integer */
void nspvbTone(NSPVToneState *statePtr, WCplx *samps, int sampsLen);
      /* complex values; short integer */
```

| | |
|---|---|
| *samps* | Pointer to the array which stores the samples. |
| *sampsLen* | The number of samples of the tone to be computed. |
| *statePtr* | Pointer to the `NSP?ToneState` structure. |

### Discussion

The `nsp?bTone()` function references the `NSP?ToneState` structure, computes *sampsLen* samples of the tone, and stores them in the array *samps[n]*. The first call to `nsp?bTone()` returns the $n = 0$ sample of $x(n)$. For real tones, $x(n)$ is defined as:

$$x(n) = \text{mag} \cdot \cos(2\pi \cdot rfreq \cdot n + phase)$$

For complex tones, $x(n)$ is defined as:

$$x(n) = \text{mag} \cdot \cos(2\pi \cdot rfreq \cdot n + phase) + j \cdot \sin(2\pi \cdot rfreq \cdot n + phase)$$

Calls to `nsp?Tone()` and `nsp?bTone()` can be combined on the same *statePtr*.

### Related Topics

| | |
|---|---|
| `Tone` | Produces the next sample of a tone. |
| `ToneInit` | Initializes a tone with a given frequency, phase, and magnitude. |

## Tone

*Produces the next sample of a tone.*

```
float nspsTone(NSPSToneState *statePtr);
      /* real values; single precision */
SCplx nspcTone(NSPCToneState *statePtr);
      /* complex values; single precision */
double nspdTone(NSPDToneState *statePtr);
      /* real values; double precision */
DCplx nspzTone(NSPZToneState *statePtr);
      /* complex values; double precision */
short nspwTone(NSPWToneState *statePtr);
      /* real values; short integer */
WCplx nspvTone(NSPVToneState *statePtr);
      /* complex values; short integer */
```

| | |
|---|---|
| *`statePtr`* | Pointer to the `NSP?ToneState` structure. |

### Discussion

The `nsp?Tone()` function references the `NSP?ToneState` structure and returns the next sample of the tone. The first call to `nsp?Tone()` returns the $n = 0$ sample of $x(n)$. For real tones, $x(n)$ is defined as:

$$x(n) = \text{mag} \cdot \cos(2\pi \cdot rfreq \cdot n + phase)$$

For complex tones, $x(n)$ is defined as:

$$x(n) = \text{mag} \cdot \cos(2\pi \cdot rfreq \cdot n + phase) + j \cdot \sin(2\pi \cdot rfreq \cdot n + phase)$$

Calls to `nsp?Tone()` and `nsp?bTone()` can be mixed on the same *`statePtr`*.

### Related Topics

| | |
|---|---|
| `bTone` | Produces consecutive samples of a tone. |
| `ToneInit` | Initializes a tone with a given frequency, phase, and magnitude. |

## ToneInit

*Initializes a tone with a given frequency, phase, and magnitude.*

```
void nspsToneInit(float rfreq, float phase, float mag,
      NSPSToneState *statePtr);
      /* real values; single precision */
void nspcToneInit(float rfreq, float phase, float mag,
      NSPCToneState *statePtr);
      /* complex values; single precision */
void nspdToneInit(double rfreq, double phase, double mag,
      NSPDToneState *statePtr);
      /* real values; double precision */
void nspzToneInit(double rfreq, double phase, double mag,
      NSPZToneState *statePtr);
      /* complex values; double precision */
void nspwToneInit(float rfreq, float phase, short mag,
      NSPWToneState *statePtr);
      /* real values; single precision */
void nspvToneInit(float rfreq, float phase, short mag,
      NSPVToneState *statePtr);
      /* complex values; short integer */
```

| | |
|---|---|
| *mag* | The magnitude of the tone; that is, the maximum value attained by the wave. |
| *phase* | The phase of the tone relative to a cosine wave. It must be between 0.0 and $2\pi$. |
| *rfreq* | The frequency of the tone relative to the sampling frequency. It must be between 0.0 and 0.5. |

| | |
|---|---|
| *statePtr* | Pointer to the NSP?ToneState structure. |

### Discussion

The nsp?ToneInit() function initializes the given NSP?ToneState structure pointed to by *statePtr* with the specified frequency, phase, and magnitude. These parameters are used to generate the tone. The NSP?ToneState structure is later passed to the nsp?Tone() and/or nsp?bTone() functions to generate samples of the tone.

For real tones, the arguments to nsp?ToneInit() specify the following signal:

$x(n) = \text{mag} \cdot \cos(2\pi \cdot rfreq \cdot n + phase)$

For complex tones, the arguments to nsp?ToneInit() specify the following signal:

$x(n) = \text{mag} \cdot \cos(2\pi \cdot rfreq \cdot n + phase) + j \cdot \sin(2\pi \cdot rfreq \cdot n + phase)$

### Related Topics

| | |
|---|---|
| bTone | Produces consecutive samples of a tone. |
| Tone | Produces the next sample of a tone. |

## Triangle-Generating Functions

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

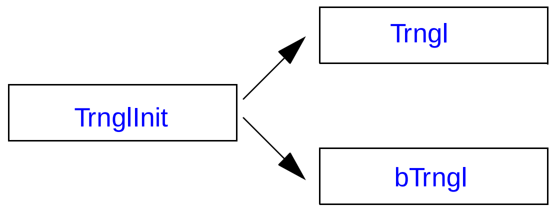
The Signal Processing Library provides functions for initializing a triangle generator and functions for generating single or multiple samples from a previously initialized triangle.

| | |
|---|---|
| nsp?bTrngl() | Returns a specified number of consecutive samples of the triangle. |
| nsp?Trngl() | Returns one sample of the triangle each time the function is called. |
| nsp?TrnglInit() | Initializes the NSP?TrnglState structure with a given frequency, phase, magnitude, and asymmetry for the triangle. |

`NSP?TrnglState` Structure which contains the specified parameters for the triangle.

Figure 6-2 illustrates the order of use of the triangle-generating functions.

**Figure 5-2 Order of Use of the Triangle-Generating Functions**




The `nsp?TrnglInit()` function initializes the `NSP?TrnglState` structure with a specified frequency, phase, magnitude, and asymmetry for the triangle. The structure can then be passed to either the `nsp?bTrngl()` function, the `nsp?Trngl()` function, or both. The `nsp?Trngl()` function returns a sample of the triangle each time it is called. The `nsp?bTrngl()` function returns a specified number of consecutive samples.

Example 5-3 shows the code for generating periodic signals with triangular waves.

**Example 5-3 Generating Triangles**

```
/* generate triangles
 * with different
 * wave forms
 */
float x[258], y[128];
NSPSTrnglState ct, saw_sheer_back, saw_sheer_fore, rect;
int i=128;
float eps=0.00001, float large_mag=1000000.;

/* initialize symmetric triangle wave */
nspsTrnglInit(0.1, 0.03, 3.0, 0.0, &ct);
nspsbTrngl(&ct, x, 128);

/* now generate a single sample
 * then continue generating the same wave */
while (i--) {
    nspsTrngl(&ct, x[129]);
}
/* generate a “saw” sheer-back wave
 * with asymmetry near -π */
nspsTrnglInit(0.09, 0.0, 1.0, eps-NSP_PI, &saw_sheer_back);
nspsbTrngl(&saw_sheer_back, y, 128);

/* generate a “saw” sheer fore-part wave
 * with asymmetry near π */
nspsTrnglInit(0.09, 0.0, 1.0, NSP_PI-eps, &saw_sheer_fore);
nspsbTrngl(&saw_sheer_fore, y, 128);

/* generate a rectangular wave from a triangle
 * using a large magnitude and asymmetry = 0 */
nspsTrnglInit(0.08, 1.5*NSP_PI, large_mag, 0.0, &rect);
nspsbTrngl(&rect, y, 128);
/* you can generate other signal shapes; for example, by using
/* Thresh and other SP functions */
```

**Application Notes:** A real periodic signal with triangular wave form `x[n]` (a real triangle, in short) of a given frequency `rfreq`, phase `phase`, magnitude `mag`, and asymmentry `h` is defined as follows:

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

A complex periodic signal with triangular wave form `x[n]` (a complex triangle, in short) of a given frequency `rfreq`, phase `phase`, magnitude `mag`, and asymmetry `h` is defined as follows:

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

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

$$H = \pi + h$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## bTrngl

*Produces consecutive samples of a triangle.*

```
void nspsbTrngl(NSPSTrnglState *statePtr, float *samps,
      int sampsLen);
      /* real values; single precision */
void nspcbTrngl(NSPCTrnglState *statePtr, SCplx *samps,
      int sampsLen);
      /* complex values; single precision */
void nspdbTrngl(NSPDTrnglState *statePtr, double *samps,
      int sampsLen);
      /* real values; double precision */
void nspzbTrngl(NSPZTrnglState *statePtr, DCplx *samps,
      int sampsLen);
      /* complex values; double precision */
void nspwbTrngl(NSPWTrnglState *statePtr, short *samps,
      int sampsLen);
      /* real values; short integer */
void nspvbTrngl(NSPVTrnglState *statePtr, WCplx *samps,
      int sampsLen);
      /* complex values; short integer */
```

| | |
|---|---|
| *samps* | Pointer to the array which stores the samples. |
| *sampsLen* | The number of samples of the triangle to be computed. |
| *statePtr* | Pointer to the `NSP?TrnglState` structure. |

### Discussion

The `nsp?bTrngl()` function references the `NSP?TrnglState` structure, computes *`sampsLen`* samples of the triangle, and stores them in the array *`samps[n]`*. The first call to `nsp?bTrngl()` returns the $n = 0$ sample of $x(n)$. For real triangle, $x(n)$ is defined as:

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

For complex triangles, $x(n)$ is defined as:

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

See page 5-10, "Application Notes," for the definition of functions $\mathbf{ct}_h()$ and $\mathbf{st}_h()$. Calls to `nsp?Trngl()` and `nsp?bTrngl()` can be combined on the same *`statePtr`*.

### Related Topics

| | |
|---|---|
| `Trngl` | Produces the next sample of a triangle. |
| `TrnglInit` | Initializes a triangle with a given frequency, phase, magnitude, and asymmetry. |

## Trngl

*Produces the next sample of a triangle.*

```
float nspsTrngl(NSPSTrnglState *statePtr);
      /* real values; single precision */
SCplx nspcTrngl(NSPCTrnglState *statePtr);
      /* complex values; single precision */
double nspdTrngl(NSPDTrnglState *statePtr);
      /* real values; double precision */
DCplx nspzTrngl(NSPZTrnglState *statePtr);
      /* complex values; double precision */
```

```
float nspwTrngl(NSPWTrnglState *statePtr);
      /* real values; short integer */
WCplx nspvTrngl(NSPVTrnglState *statePtr);
      /* complex values; short integer */
```

*statePtr* Pointer to the `NSP?TrnglState` structure.

## Discussion

The `nsp?Trngl()` function references the `NSP?TrnglState` structure and returns the next sample of the triangle. The first call to `nsp?Trngl()` returns the $n = 0$ sample of $x(n)$.

For real triangles, $x(n)$ is defined as:

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

For complex triangles, $x(n)$ is defined as:

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

See page 5-10, "Application Notes," for the definition of functions $\mathbf{ct}_h()$ and $\mathbf{st}_h()$. Calls to `nsp?Trngl()` and `nsp?bTrngl()` can be mixed on the same *statePtr*./

## Related Topics

`bTrngl` Produces consecutive samples of a triangle.

`TrnglInit` Initializes a triangle with a given frequency, phase, and magnitude.

## TrnglInit

*Initializes a triangle with a given frequency, phase, and magnitude.*

```
void nspsTrnglInit(float rfreq, float phase, float mag, float asym,
      NSPSTrnglState *statePtr);
      /* real values; single precision */
void nspcTrnglInit(float rfreq, float phase, float mag, float asym,
      NSPCTrnglState *statePtr);
      /* complex values; single precision */
void nspdTrnglInit(double rfreq, double phase, double mag,
      double asym, NSPDTrnglState *statePtr);
      /* real values; double precision */
void nspzTrnglInit(double rfreq, double phase, double mag,
      double asym, NSPZTrnglState *statePtr);
      /* complex values; double precision */
void nspwTrnglInit(float rfreq, float phase, short mag, float asym,
      NSPWTrnglState *statePtr);
      /* real values; short integer */
void nspvTrnglInit(float rfreq, float phase, short mag, float asym,
      NSPVTrnglState *statePtr);
      /* complex values; short integer */
```

| | |
|---|---|
| *rfreq* | The frequency of the triangle relative to the sampling frequency. It must be between 0.0 and 0.5. |
| *phase* | The phase of the triangle relative to a cosine triangular analog wave. It must be between 0.0 and $2\pi$. |
| *mag* | The magnitude of the triangle; that is, the maximum value attained by the wave. |
| *asym* | The asymmetry *h* of a triangle. It must be between $-\pi$ and $\pi$. If *h*=0, then the triangle is symmetric and a direct analog of a tone. |
| *statePtr* | Pointer to the `NSP?TrngleState` structure. |

### Discussion

The `nsp?TrnglInit()` function initializes the given `NSP?TrnglState` structure pointed to by *statePtr* with the specified frequency, phase, and magnitude. These parameters are used to generate the triangle. The `NSP?TrnglState` structure is later passed to the `nsp?Trngl()` and/or `nsp?bTrngl()` functions to generate samples of the triangle.

For real triangles, *x(n)* is defined as:

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

For complex triangles, *x(n)* is defined as:

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

See page 5-10, "Application Notes," for the definition of functions $\mathbf{ct}_h()$ and $\mathbf{st}_h()$. Calls to `nsp?Trngl()` and `nsp?bTrngl()` can be mixed on the same *statePtr*.

### Related Topics

| | |
|---|---|
| `bTrngl` | Produces consecutive samples of a triangle. |
| `Trngl` | Produces the next sample of a triangle. |

## Pseudo-Random Samples Generation

The Signal Processing Library provides functions for initializing a random-sample generator and functions for generating single or multiple pseudo-random samples from a previously initialized sample with uniform or Gaussian distribution.

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

## Uniform Distribution Functions

The pseudo-random samples with uniform distribution functions include:

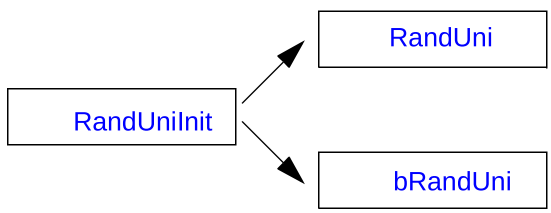
`nsp?RandUniInit()`
Initializes the `NSP?RandUniState` structure required to generate the pseudo-random samples.

`nsp?RandUni()` Returns consecutive samples, one at a time.

`nsp?bRandUni()` Computes samples and stores them in an array.

Figure 5-3 illustrates the order of use of the pseudo-random sample-generating functions with a uniform distribution.

**Figure 5-3 Order of Use of the Uniform Distribution Functions**




The `nsp?RandUniInit()` function initializes the given `NSP?RandUniState` structure, and the application can pass it to the `nsp?RandUni()` and/or `nsp?bRandUni()` functions to generate consecutive samples. Example 5-4 shows a simulation of a noisy digital transmission with 7% bit-error rate.

**Example 5-4 Simulation of a Noisy Digital Transition**

```
/* Simulate noisy digital transmission
 * with 7% bit-error rate
 */
NSPSRandUniState  rstate;
char              data;
int               i;
nspsRandUniInit(0.1, 0.0, 100.0, &rstate);
for(;;) {
     /* insert code here to put next eight bits of signal in data */
      for (i=0; i<8; i++)
            if (nspsRandUni (&rstate) < 7.0)
                 data = data ^(1<<i);
      /* each bit now has a 7% probability of being corrupted */
         }
/* dither a signal and quantize to eight bits */
NSPDRandUniState rstate;
double           samps[256];
double           dither[256];
char             output[256];
nspdRandUniInit (0.1, -0.25, 0.25, &rstate);
for (;;) {
      /* insert code here to fill samps[] with samples */
      nspdbRandUni(&rstate, dither, 256);
      nspdbAdd2(dither, samps, 256);
      nspdbFloat2Int(samps, output, 256, 8, NSP_Round | NSP_Clip);
      }
             ...
}
```

## bRandUni

*Computes pseudo-random samples with a uniform distribution and stores them in an array.*

```
void nspsbRandUni(NSPSRandUniState *statePtr, float *samps,
      int sampsLen);
      /* real values; single precision */
void nspcbRandUni(NSPCRandUniState *statePtr, SCplx *samps,
      int sampsLen);
      /* complex values; single precision */
void nspdbRandUni(NSPDRandUniState *statePtr, double *samps,
      int sampslen);
      /* real values; double precision */
void nspzbRandUni(NSPZRandUniState *statePtr, DCplx *samps,
      int sampsLen);
      /* complex values; double precision */
void nspwbRandUni(NSPWRandUniState *statePtr, short *samps,
      int sampsLen);
      /* real values; short integer */
void nspvbRandUni(NSPVRandUniState *statePtr, WCplx *samps,
      int sampsLen);
      /* complex values; short integer */
```

| | |
|---|---|
| *statePtr* | Pointer to the `NSP?RandUniState` structure. |
| *samps* | Pointer to the array containing pseudo-random samples. |
| *sampsLen* | The number of elements (samples) in the *samps* array. |

### Discussion

The `nsp?bRandUni` function computes *sampsLen* pseudo-random samples with a uniform distribution and stores them in the *samps* array.

Calls to `nsp?RandUni()` and `nsp?bRandUni()` can be mixed on the same *statePtr*.

### Related Topics

| | |
|---|---|
| RandUniInit | Initializes the state required to generate the pseudo-random samples. |
| RandUni | Returns consecutive samples, one at a time. |

## RandUni

*Returns consecutive pseudo-random samples with a uniform distribution, one at a time.*

```
float nspsRandUni(NSPSRandUniState *statePtr);
      /* real values; single precision */
SCplx nspcRandUni(NSPCRandUniState *statePtr);
      /* complex values; single precision */
double nspdRandUni(NSPDRandUniState *statePtr);
      /* real values; double precision */
DCplx nspzRandUni(NSPZRandUniState *statePtr);
      /* complex values; double precision */
short nspwRandUni(NSPWRandUniState *statePtr);
      /* real values; short integer */
WCplx nspvRandUni(NSPVRandUniState *statePtr);
      /* complex values; short integer */
```

| | |
|---|---|
| *statePtr* | Pointer to the `NSP?RandUniState` structure. |

### Discussion

The `nsp?RandUni` function returns consecutive pseudo-random samples with a uniform distribution, one at a time.

### Related Topics

| | |
|---|---|
| RandUniInit | Initializes the state required to generate the pseudo-random samples. |
| bRandUni | Computes pseudo-random samples and stores them in an array. |

## RandUniInit

*Initializes the state required to generate the pseudo-random samples with a uniform distribution.*

```
void nspsRandUniInit(float seed, float low, float high,
      NSPSRandUniState *statePtr);
      /* real values; single precision */
void nspcRandUniInit(float seed, float low, float high,
      NSPCRandUniState *statePtr);
      /* complex values; single precision */
void nspdRandUniInit(double seed, double low, double high,
      NSPDRandUniState *statePtr);
      /* real values; double precision */
void nspzRandUniInit(double seed, double low, double high,
      NSPZRandUniState *statePtr);
      /* complex values; double precision */
void nspwRandUniInit(short seed, short low, short high,
      NSPWRandUniState *statePtr);
      /* real values; short integer */
void nspvRandUniInit(short seed, short low, short high,
      NSPVRandUniState *statePtr);
      /* complex values; short integer */
```

| | |
|---|---|
| *seed* | The seed value used by the pseudo-random number generation algorithm. |
| *low* | The lower bounds of the uniform distribution's range. |
| *high* | The upper bounds of the uniform distribution's range. |
| *statePtr* | Pointer to the `NSP?RandUniState` structure. |

### Discussion

The `nsp?RandUniInit` function initializes the state required to generate the pseudo-random samples with a uniform distribution. Note that floating-point *seed* values are truncated to integer type before use.

### Related Topics

| | |
|---|---|
| bRandUni | Computes pseudo-random samples and stores them in an array. |
| RandUni | Returns consecutive samples, one at a time. |

## Gaussian Distribution Functions

The pseudo-random samples with Gaussian distribution functions include:

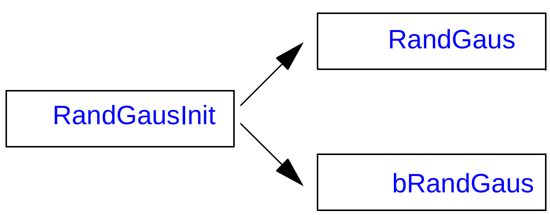
nsp?RandGausInit()
Initializes the NSP?RandGausState structure required to generate the pseudo-random samples.

nsp?RandGaus() Returns consecutive samples, one at a time.

nsp?bRandGaus() Computes samples and stores them in an array.

Figure 5-4 illustrates the order of use of the pseudo-random sample-generating functions with a Gaussian distribution.

**Figure 5-4 Order of Use of the Gaussian Distribution Functions**




The nsp?RandGausInit() function initializes the given NSP?RandGausState structure, and the application can pass it to the nsp?RandGaus() and/or nsp?bRandGaus() functions to generate consecutive samples. Calls to nsp?RandGaus() and nsp?bRandGaus() may be mixed on the same *statePtr*.

## bRandGaus

*Computes pseudo-random samples with a Gaussian distribution and stores them in an array.*

```
void nspsbRandGaus(NSPSRandGausState *statePtr, float *samps,
      int sampsLen);
      /* real values; single precision */
void nspcbRandGaus(NSPCRandGausState *statePtr, SCplx *samps,
      int sampsLen);
      /* complex values; single precision */
void nspdbRandGaus(NSPDRandGausState *statePtr, double *samps,
      int sampslen);
      /* real values; double precision */
void nspzbRandGaus(NSPZRandGausState *statePtr, DCplx *samps,
      int sampsLen);
      /* complex values; double precision */
void nspwbRandGaus(NSPWRandGausState *statePtr, short *samps,
      int sampsLen);
      /* real values; short integer */
void nspvbRandGaus(NSPVRandGausState *statePtr, WCplx *samps,
      int sampsLen);
      /* complex values; short integer */
```

| | |
|---|---|
| *statePtr* | Pointer to the `NSP?RandGausState` structure. |
| *samps* | Pointer to the array containing pseudo-random samples. |
| *sampsLen* | The number of elements (samples) in the *samps* array. |

### Discussion

The `nsp?bRandGaus` function computes *sampsLen* pseudo-random samples with a Gaussian distribution and stores them in the *samps* array.

### Related Topics

| | |
|---|---|
| `RandGausInit` | Initializes the state required to generate the pseudo-random samples. |
| `RandGaus` | Returns consecutive samples, one at a time. |

# RandGaus

*Returns consecutive pseudo-random samples with a Gaussian distribution, one at a time.*

```
float nspsRandGaus(NSPSRandGausState *statePtr);
      /* real values; single precision */
SCplx nspcRandGaus(NSPCRandGausState *statePtr);
      /* complex values; single precision */
double nspdRandGaus(NSPDRandGausState *statePtr);
      /* real values; double precision */
DCplx nspzRandGaus(NSPZRandGausState *statePtr);
      /* complex values; double precision */
short nspwRandGaus(NSPWRandGausState *statePtr);
      /* real values; short integer */
WCplx nspwRandGaus(NSPVRandGausState *statePtr);
      /* complex values; short integer */
```

| | |
|---|---|
| *statePtr* | Pointer to the `NSP?RandGausState` structure. |

## Discussion

The `nsp?RandGaus` function returns consecutive pseudo-random samples with a Gaussian distribution, one at a time.

## Related Topics

| | |
|---|---|
| `RandGausInit` | Initializes the state required to generate the pseudo-random samples. |
| `bRandGaus` | Computes pseudo-random samples and stores them in an array. |

## RandGausInit

*Initializes the state required to generate the pseudo-random samples with a Gaussian distribution.*

```
void nspsRandGausInit(float seed, float mean, float stdDev,
      NSPSRandGausState *statePtr);
      /* real values; single precision */
void nspcRandGausInit(float seed, float mean, float stdDev,
      NSPCRandGausState *statePtr);
      /* complex values; single precision */
void nspdRandGausInit(double seed, double mean, double stdDev,
      NSPDRandGausState *statePtr);
      /* real values; double precision */
void nspzRandGausInit(double seed, double mean, double stdDev,
      NSPZRandGausState *statePtr);
      /* complex values; double precision */
void nspwRandGausInit(short seed, short mean, short stdDev,
      NSPWRandGausState *statePtr);
      /* real values; short integer */
void nspvRandGausInit(short seed, short mean, short stdDev,
      NSPVRandGausState *statePtr);
      /* complex values; short integer */
```

| | |
|---|---|
| *seed* | The seed value used by the pseudo-random number generation algorithm. |
| *mean* | The mean of the Gaussian distribution. |
| *stdDev* | The standard deviation of the Gaussian distribution. |
| *statePtr* | Pointer to the `NSP?RandUniState` structure. |

### Discussion

The `nsp?RandGausInit` function initializes the state required to generate the pseudo-random samples with a Gaussian distribution. Note that floating-point *seed* values are truncated to integer type before use.

# Windowing Functions

6

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

Library function lists

The Signal Processing Library provides the following functions to generate window samples:

- Bartlett windowing function
- Blackman family of windowing functions
- Hamming windowing function
- Hann windowing function
- Kaiser windowing function

These functions generate the window samples and multiply them into an existing signal.  To obtain the window samples themselves, initialize the vector argument to the unity vector before calling the window function.

Example 6-1 shows the code for windowing a time-domain signal and taking its FFT.

### Example 6-1 Window and FFT a Single Frame of a Signal

```
/* window and FFT a single
 * frame of a signal
 */
double       xTime[128];
DCplx        xFreq[65];

/* insert code here to put time-domain samples in xTime */

nspdWinHamming(xTime, 128);
nspdRealFftNip(xTime, xFreq, 7, NSP_Forw);
/* FFT samples are now in xFreq */
```

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 (`nsp?WinBlackmanStd()`, for example) on a vector with all elements set to 1.0. Then use one of the vector multiplication functions (`nsp?bMpy2()`, for example) to multiply the window into the signal each time a new set of input samples is available. This avoids repeatedly calculating the window samples. This is illustrated in Example 6-2.

### Example 6-2 Window and FFT Many Frames of a Signal

```
/* window and FFT many
 * frames of a signal
 */
double       xTime[128], win[128];
DCplx        xFreq[65];

nspdbSet(1.0, win, 128);
nspdWinBlackmanStd(win, 128);
for (;;) {
      /* insert code here to put
       * time-domain samples in xTime
       */
      nspdbMpy2(win, xTime, 128);
      nspdRealFftNip(xTime, xFreq, 7, NSP_Forw);
      /* FFT samples are now in xFreq */
}
```

## Related Topics

For more information on windows, see: [Jac89], section 7.3, *Windows in Spectrum Analysis*; [Jac89], section 9.1, *Window-Function Technique*; and [Mit93], section 16-2, *Fourier Analysis of Finite-Time Signals*. For more information on these references, see the Bibliography at the end of this manual.

## WinBartlett, WinBartlett2

*Multiplies a vector by a Bartlett windowing function.*

```
void nspsWinBartlett(float *vec, int N);
void nspsWinBartlett2(const float *src, float *dst, int N);
      /* real values; single precision */
void nspcWinBartlett(SCplx *vec, int N);
void nspcWinBartlett2(const SCplx *src, SCplx *dst, int N);
      /* complex values; single precision */
void nspdWinBartlett(double *vec, int N);
void nspdWinBartlett2(const double *src, double *dst, int N);
      /* real values; double precision */
void nspzWinBartlett(DCplx *vec, int N);
void nspzWinBartlett2(const DCplx *src, DCplx *dst, int N);
      /* complex values; double precision */
void nspwWinBartlett(short *vec, int N);
void nspwWinBartlett2(const short *src, short *dst, int N);
      /* real values; short integer */
void nspvWinBartlett(WCplx *vec, int N);
void nspvWinBartlett2(const WCplx *src, WCplx *dst, int N);
      /* complex values; short integer */
```

| | |
|---|---|
| `vec, src` | Pointer to the vector to be multiplied by the chosen windowing function. |
| `dst` | Pointer to the output vector (for the not-in-place functions). |
| `N` | The number of samples in the input and output vectors. |

### Discussion

The `nsp?WinBartlett(), nsp?WinBartlett2()` functions multiply a vector by the Bartlett (triangle) window. The complex flavors multiply both the real and imaginary parts of the vector by the same window. The `nsp?WinBartlett()` function performs operations in-place, whereas `nsp?WinBartlett2()` uses different source and destination vectors.

The Bartlett window is defined as follows:

$$w_{bartlett}(n) = \begin{cases} \dfrac{2n}{N-1}, & 0 \le n \le \dfrac{N-1}{2} \\ 2 - \dfrac{2n}{N-1}, & \dfrac{N-1}{2} < n \le N-1 \end{cases}$$

## WinBlackman, WinBlackman2

*Multiplies a vector by a Blackman windowing function.*

```
void nspsWinBlackman(float *vec, int N, float alpha);
void nspsWinBlackman2(const float *src, float *dst, int N,
      float alpha);
void nspsWinBlackmanStd(float *vec, int N);
void nspsWinBlackmanStd2(const float *src, float *dst, int N);
void nspsWinBlackmanOpt(float *vec, int N);
void nspsWinBlackmanOpt2(const float *src, float *dst, int N);
      /* real values; single precision */

void nspcWinBlackman(SCplx *vec, int N, float alpha);
void nspcWinBlackman2(const SCplx *src, SCplx *dst, int N,
      float alpha);
void nspcWinBlackmanStd(SCplx *vec, int N);
void nspcWinBlackmanStd2(const SCplx *src, SCplx *dst, int N);
void nspcWinBlackmanOpt(SCplx *vec, int N);
void nspcWinBlackmanOpt2(const SCplx *src, SCplx *dst, int N);
      /* complex values; single precision */

void nspdWinBlackman(double *vec, int N, double alpha);
void nspdWinBlackman2(const double *src, double *dst, int N,
      double alpha);
void nspdWinBlackmanStd(double *vec, int N);
void nspdWinBlackmanStd2(const double *src, double *dst, int N);
void nspdWinBlackmanOpt(double *vec, int N);
void nspdWinBlackmanOpt2(const double *src, double *dst, int N);
      /* real values; double precision */
```

```
void nspzWinBlackman(DCplx *vec, int N, double alpha);
void nspzWinBlackman2(const DCplx *src, DCplx *dst, int N,
      double alpha);
void nspzWinBlackmanStd(DCplx *vec, int N);
void nspzWinBlackmanStd2(const DCplx *src, DCplx *dst, int N);
void nspzWinBlackmanOpt(DCplx *vec, int N);
void nspzWinBlackmanOpt2(const DCplx *src, DCplx *dst, int N);
      /* complex values; double precision */

void nspwWinBlackman(short *vec, int N, float alpha);
void nspwWinBlackman2(const short *src, short *dst, int N,
      float alpha);
void nspwWinBlackmanStd(short *vec, int N);
void nspwWinBlackmanStd2(const short *src, short *dst, int N);
void nspwWinBlackmanOpt(short *vec, int N);
void nspwWinBlackmanOpt2(const short *src, short *dst, int N);
      /* real values; short integer */

void nspvWinBlackman(WCplx *vec, int N, float alpha);
void nspvWinBlackman2(const WCplx *src, WCplx *dst, int N,
      float alpha);
void nspvWinBlackmanStd(WCplx *vec, int N);
void nspvWinBlackmanStd2(const WCplx *src, WCplx *dst, int N);
void nspvWinBlackmanOpt(WCplx *vec, int N);
void nspvWinBlackmanOpt2(const WCplx *src, WCplx *dst, int N);
      /* complex values; short integer */
```

| | |
|---|---|
| *vec, src* | Pointer to the vector to be multiplied by the chosen windowing function. |
| *dst* | Pointer to the output vector (for the not-in-place functions). |
| *N* | The number of samples in the input and output vectors. |
| *alpha* | An adjustable parameter associated with the Blackman windowing equation. |

## Discussion

The `nsp?WinBlackman()` family of functions multiply a vector by a Blackman window. The complex flavors multiply both the real and imaginary parts of the vector by the same window.
The `nsp?WinBlackman()`, `nsp?WinBlackmanStd()`, `nsp?WinBlackmanOpt()` functions perform operations in-place, whereas `nsp?WinBlackman2()`, `nsp?WinBlackmanStd2()`, `nsp?WinBlackmanOpt2()` functions use different source and destination vectors.

The functions for the Blackman family of windows are defined below.

**`nsp?WinBlackman()`, `nsp?WinBlackman2()`.** The `nsp?WinBlackman()` function allows the application to specify *`alpha`*.

$$w_{blackman}(n) = \frac{alpha + 1}{2} - 0.5\cos\left(\frac{2\pi n}{N-1}\right) - \frac{alpha}{2}\cos\left(\frac{4\pi n}{N-1}\right)$$

$$0 \le n < N$$

**`nsp?WinBlackmanStd()`, `nsp?WinBlackmanStd2()`.** The traditional, standard Blackman window is provided by the `nsp?WinBlackmanStd()` function, which simply calls `nsp?WinBlackman()` with the value of *`alpha_std`* shown below.

*`alpha_std`* = -0.16

**`nsp?WinBlackmanOpt()`, `nsp?WinBlackmanOpt2()`.** The `nsp?WinBlackmanOpt()` function provides a modified window that has a 30-dB/octave roll-off by calling `nsp?WinBlackman()` with the value of *`alpha_opt`* shown below.

$$alpha\_opt = -\left[\frac{\sin\dfrac{\pi}{N-1}}{\sin\dfrac{2\pi}{N-1}}\right]^2$$

For large *`N`*, *`alpha_opt`* converges asymptotically to *`alpha_asym`*; the application can use this value with `nsp?WinBlackman()`.

*`alpha_asym`* = -0.25

## WinHamming, WinHamming2

*Multiplies a vector by a Hamming windowing function.*

```
void nspsWinHamming(float *vec, int N);
void nspsWinHamming2(const float *src, float *dst, int N);
      /* real values; single precision */
void nspcWinHamming(SCplx *vec, int N);
void nspcWinHamming2(const SCplx *src, SCplx *dst, int N);
      /* complex values; single precision */
void nspdWinHamming(double *vec, int N);
void nspdWinHamming2(const double *src, double *dst, int N);
      /* real values; double precision */
void nspzWinHamming(DCplx *vec, int N);
void nspzWinHamming2(const DCplx *src, DCplx *dst, int N);
      /* complex values; double precision */
void nspwWinHamming(short *vec, int N);
void nspwWinHamming2(const short *src, short *dst, int N);
      /* real values; short integer */
void nspvWinHamming(WCplx *vec, int N);
void nspvWinHamming2(const WCplx *src, WCplx *dst, int N);
      /* complex values; short integer */
```

| | |
|---|---|
| *vec, src* | Pointer to the vector to be multiplied by the chosen windowing function. |
| *dst* | Pointer to the output vector (for the not-in-place functions). |
| *N* | The number of samples in the input and output vectors. |

### Discussion

The `nsp?WinHamming(), nsp?WinHamming2()` functions multiply a vector by the Hamming window. The complex flavors multiply both the real and imaginary parts of the vector by the same window.
The `nsp?WinHamming()` function performs operations in-place, whereas `nsp?WinHamming2()` uses different source and destination vectors.

The Hamming window is defined as follows:

$$w_{hamming}(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right), \quad 0 \le n < N$$

## WinHann, WinHann2

*Multiplies a vector by a Hann windowing function.*

```
void nspsWinHann(float *vec, int N);
void nspsWinHann2(const float *src, float *dst, int N);
      /* real values; single precision */
void nspcWinHann(SCplx *vec, int N);
void nspcWinHann2(const SCplx *src, SCplx *dst, int N);
      /* complex values; single precision */
void nspdWinHann(double *vec, int N);
void nspdWinHann2(const double *src, double *dst, int N);
      /* real values; double precision */
void nspzWinHann(DCplx *vec, int N);
void nspzWinHann2(const DCplx *src, DCplx *dst, int N);
      /* complex values; double precision */
void nspwWinHann(short *vec, int N);
void nspwWinHann2(const short *src, short *dst, int N);
      /* real values; short integer */
void nspvWinHann(WCplx *vec, int N);
void nspvWinHann2(const WCplx *src, WCplx *dst, int N);
      /* complex values; short integer */
```

| | |
|---|---|
| `vec, src` | Pointer to the vector to be multiplied by the chosen windowing function. |
| `dst` | Pointer to the output vector (for the not-in-place functions). |
| `N` | The number of samples in the input and output vectors. |

### Discussion

The `nsp?WinHann(), nsp?WinHann2()` functions multiply a vector by the Hann window. The complex flavors multiply both the real and imaginary parts of the vector by the same window.
The `nsp?WinHann()` function performs operations in-place, whereas `nsp?WinHann2()` uses different source and destination vectors.

The Hann window is defined as follows:

$$w_{hann}(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right), \quad 0 \le n < N$$

## WinKaiser, WinKaiser2

*Multiplies a vector by a Kaiser windowing function.*

```
void nspsWinKaiser(float *vec, int N, float beta);
void nspsWinKaiser2(const float *src, float *dst, int N, float beta);
      /* real values; single precision */
void nspcWinKaiser(SCplx *vec, int N, float beta);
void nspcWinKaiser2(const SCplx *src, SCplx *dst, int N, float beta);
      /* complex values; single precision */
void nspdWinKaiser(double *vec, int N, double beta);
void nspdWinKaiser2(const double *src, double *dst, int N,
      double beta);
      /* real values; double precision */
void nspzWinKaiser(DCplx *vec, int N, double beta);
void nspzWinKaiser2(const DCplx *src, DCplx *dst, int N, double beta);
      /* complex values; double precision */
void nspwWinKaiser(short *vec, int N, float beta);
void nspwWinKaiser2(const short *src, short *dst, int N, float beta);
      /* real values; short integer */
void nspvWinKaiser(WCplx *vec, int N, float beta);
void nspvWinKaiser2(const WCplx *src, WCplx *dst, int N, float beta);
      /* complex values; short integer */
```

| | |
|---|---|
| `vec, src` | Pointer to the vector to be multiplied by the chosen windowing function. |
| `dst` | Pointer to the output vector (for the not-in-place functions). |
| `N` | The number of samples in the input and output vectors. |
| `beta` | An adjustable parameter associated with the Kaiser windowing equation. |

## Discussion

The `nsp?WinKaiser(), nsp?WinKaiser2()` functions multiply a vector by the Kaiser window. The complex flavors multiply both the real and imaginary parts of the vector by the same window.
The `nsp?WinKaiser()` function performs operations in-place, whereas `nsp?WinKaiser2()` uses different source and destination vectors.

The Kaiser family of windows are defined as follows:

$$w_{kaiser}(n) = \frac{I_0\left(beta\sqrt{\left(\frac{N-1}{2}\right)^2 - \left(n - \left(\frac{N-1}{2}\right)\right)^2}\right)}{I_0\left(beta\left(\frac{N-1}{2}\right)\right)}, \quad 0 \le n < N$$

where $I_0()$ is the modified zero-order Bessel function of the first kind.

# Fourier and Discrete Cosine Transform Functions

# 7

Library function lists

This chapter describes the Fourier and discrete cosine transform functions in the Signal Processing Library. The library contains functions which perform the discrete Fourier transform (DFT), the fast Fourier transform (FFT), and the discrete cosine transform (DCT) of signal samples. It also includes variations of the basic functions to support different application requirements.

The basic Fourier transform functions are described in these sections:

DFT Function. This section describes the `nsp?Dft()` function. This function performs the complex Fourier transform of a finite-length signal. DFT for a Given Frequency (Goertzel) Functions. This section describes the `GoertzInit()`, `GoertzReset()`, `bGoertz()`, and `Goertz()` functions based on Goertzel algorithm. These functions compute discrete Fourier transforms for individual frequencies.

Basic FFT Functions. This section describes the `nsp?Fft()`, `nsp?FftNip()`, `nsp?rFft()`, and `nsp?rFftNip()` functions. These functions compute the complex fast Fourier transform of a signal. The fast Fourier transform produces identical results as the discrete Fourier transform (provided the length of the DFT is a power of 2) but is faster.

The variations of the basic functions described in the sections that follow are significantly faster than the standard complex FFT functions described in "Basic FFT Functions":

Low-Level FFTs of Real Signals. This section describes the `nsp?RealFftl()` and `nsp?RealFftlNip()` functions. These functions are optimized for real-valued input and provide a low-level interface to compute the FFT of real signals. The `nsp?RealFftl()` and `nsp?RealFftlNip()` functions exploit symmetry properties of the basic Fourier transform.

**Low-Level FFTs of Conjugate-Symmetric Signals**. This section describes the `nsp?CcsFftl()` and `nsp?CcsFftlNip()` functions. These functions are optimized for conjugate-symmetric input and provide a low-level interface to compute the FFT of conjugate-symmetric signals. The `nsp?CcsFftl()` and `nsp?CcsFftlNip()` functions exploit symmetry properties of the basic Fourier transform.

**FFTs of Real Signals**. This section describes the `nsp?RealFft()` and `nsp?RealFftNip()` functions. These functions are optimized for real-valued input. The `nsp?RealFft()` and `nsp?RealFftNip()` functions exploit symmetry properties of the basic Fourier transform.

**FFTs of Conjugate-Symmetric Signals**. This section describes the `nsp?CcsFft()` and `nsp?CcsFftNip()` functions. These functions are optimized for conjugate-symmetric input. The `nsp?CcsFft()` and `nsp?CcsFftNip()` functions exploit symmetry properties of the basic Fourier transform.

**FFTs of Two Real Signals**. This section describes the `nsp?Real2Fft()` and `nsp?Real2FftNip()` functions. These functions simultaneously compute two real FFTs using a single complex FFT.

**FFTs of Two Conjugate-Symmetric Signals**. This section describes the `nsp?Ccs2Fft()` and `nsp?Ccs2FftNip()` functions. These functions simultaneously compute two conjugate-symmetric FFTs using a single complex FFT.

The variations of the basic FFT include normal bit order versus bit-reversed order, real versus complex signals, and complex arrays versus paired real arrays.

**Memory Reclaim Functions**. This section describes the `nspFreeBitrevTbls()` and `nsp?FreeTwdTbls()` functions. These functions free the memory allocated for bit-reversed indices tables and for twiddle tables, respectively.

**DCT Function**. This section describes the `nsp?Dct()` function. This function computes the discrete cosine transform of signals.

The library also contains specialized functions which perform the Fourier transform of real signals and conjugate-symmetric signals. These functions are divided into three groups.

- Low-level functions store their output in `RCPack` or `RCPerm` format. See "Low-Level FFTs of Real Signals" for a description of the `nsp?RealFftl()` and `nsp?RealFftlNip()` functions. See "Low-Level FFTs of Conjugate-Symmetric Signals" for a description of the `nsp?CcsFftl()` and `nsp?CcsFftlNip()` functions. These sections also describe `RCPack` and `RCPerm` formats, as well as vector multiplication in `RCPack` or `RCPerm` format.
- Higher-level functions store their output in `RCCcs` format. See "FFTs of Real Signals" for a description of the `nsp?RealFft()` and `nsp?RealFftNip()` functions. See "FFTs of Conjugate-Symmetric Signals" for a description of the `nsp?CcsFft()` and `nsp?CcsFftNip()` functions. These sections also describe the `RCCcs` format.
- The library contains functions which simultaneously compute the Fourier transform of two real signals or two conjugate-symmetric signals using a single complex FFT. The results are stored in `RCCcs` format. See "FFTs of Two Real Signals" for a description of the `nsp?Real2Fft()` and `nsp?Real2FftNip()` functions. See "FFTs of Two Conjugate-Symmetric Signals" for a description of the `nsp?Ccs2Fft()` and `nsp?Ccs2FftNip()` functions.

Figure 7-1 contains a matrix which lists the names of the Fourier transform functions in the Intel Signal Processing Library. The functions are arranged according to input and output format. The left-most column lists the possible input format, while the header lists the possible output format. For example, if you have one real array to use as input for an FFT function, find "1 real array" in the left-most column and read horizontally. You can use either `nsp?RealFftl()` or `nsp?RealFftlNip()` to obtain one output array in `RCPerm` or `RCPack` format or you can use either `nsp?RealFft()` or `nsp?RealFftNip()` to obtain one output array in `RCCcs` format.

The arrows in the matrix indicate inverse functions. For example, the inverse function of `nsp?CcsFftl()` is `nsp?RealFftl()`; the inverse function of `nsp?Real2FftNip()` is `nsp?Ccs2FftNip()`.

For the purposes of clarity in the matrix, the `nsp?` prefix is not included in the function names.

## Figure 7-1 Fourier Transforms Arranged by Input and Output Types

| | **Output Format** | 1 complex array | 1 real and 1 imaginary array | 1 real array | 2 real arrays | 1 RCPack or RCPerm format array | 1 RCCcs format array | 2 RCCcs format arrays |
|---|---|---|---|---|---|---|---|---|
| **Input Format** | | | | | | | | |
| 1 complex array | | Dft, Goertz, Fft, FftNip | | | | | | |
| 1 real and 1 imaginary array | | | rFft, rFftNip | | | | | |
| 1 real array | | | | | | RealFftl, RealFftlNip | RealFft, RealFftNip | |
| 2 real arrays | | | | | | | | Real2Fft, Real2FftNip |
| 1 RCPack or RCPerm format array | | | | CcsFftl, CcsFftlNip | | | | |
| 1 RCCcs format array | | | | CcsFft, CcsFftNip | | | | |
| 2 RCCcs format arrays | | | | | Ccs2Fft, Ccs2FftNip | | | |

# DFT Function

This section describes the function which calculates the discrete Fourier transform of a signal.

## Dft

*Computes the forward or inverse discrete Fourier transform (DFT) of a signal.*

```
void nspcDft(const SCplx *inSamps, SCplx *outSamps, int length,
      int flags);  /*complex values; single precision */
void nspzDft(const DCplx *inSamps, DCplx *outSamps, int length,
      int flags); /*complex values; double precision */
void nspvDft(const WCplx *inSamps, WCplx *outSamps, int length,
      int flags, int ScaleMode, int *ScaleFactor);
      /*complex values; short integer */
```

| | |
|---|---|
| *flags* | Specifies how the DFT should be performed. |
| *inSamps* | Pointer to the complex-valued input array. |
| *length* | The number of samples in the arrays *inSamps[n]* and *outSamps[n]*. |
| *outSamps* | Pointer to the complex-valued output array. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?Dft()` function computes the forward and inverse discrete Fourier transform (DFT). Note that the FFT (see "Fft" in page 7-16 for a description of `nsp?Fft()`) performs the equivalent function for certain length DFTs, but is much faster.

In the following definition of the discrete Fourier transform, $N$ = `length`. Also, in the forward direction, `x(n)` is `inSamps[n]` and `X(k)` is `outSamps[k]`; in the inverse direction, `x(n)` is `outSamps[n]` and `X(k)` is `inSamps[k]`.

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right)$$

The definition of the inverse discrete Fourier transform is:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k) \cdot \exp\left(j2\pi\frac{kn}{N}\right)$$

The argument `flags` consists of the bitwise-OR of one or more of the flags described in Table 7-1.

**Table 7-1 Value for the flags Argument for the DFT Function**

| Value | Description |
|---|---|
| `NSP_DoFloatCore` | Specifies that `nspvDft()` performs float core computation for all lengths of the input array. By default, float core computation is performed when length > `MAX_Dft_Length_MMX` = 128. |
| `NSP_DoIntCore` | Specifies that `nspvDft()` performs integer core computation (using MMX™ technology-optimized algorithms) for all array lengths. |
| `NSP_Forw` | Specifies a forward DFT with the `inSamps[n]` array providing `x(n)` and the `outSamps[n]` array containing `X(k)`. |
| `NSP_Free` | Frees all internal arrays and twiddle arrays |
| `NSP_Init` | Specifies that the function should initialize the twiddle table (if required), but perform no other computation. |
| `NSP_Inv` | Specifies an inverse DFT with the `inSamps[n]` array providing `X(k)` and the `outSamps[n]` array containing `x(n)`. |
| `NSP_NoScale` | Specifies that when performing an inverse transform, the 1/$N$ normalization should not be performed. |

One and only one of the values `NSP_Forw`, `NSP_Inv`, and `NSP_Init` can be specified in the *`flags`* argument.

Example 7-1 illustrates the use of the `nsp?Dft()` function.

### Example 7-1 Using nsp?Dft() to Perform the DFT

```
/*
 * Calculate 100 point DFT of an input signal.
 * Input signal is in xTime, output is in xFreq.
 */
DCplx    xTime[100], xFreq[100];

/* insert code here to put time domain
samples in xTime */

nspzDft(xTime, xFreq, 100, NSP_Forw);
/* xFreq now has frequency-domain samples */
```

### Application Notes

Use the `nsp?Dft()` function when the number of samples (*`length`*) is not a power of 2. The DFT algorithm is generally less efficient than the FFT. If your application is concerned with speed, you should use the FFT algorithm instead.

### Related Topics

See [Mit93], section 8-2, *Fast Computation of the DFT*, for more information on the fast computation of the discrete Fourier transform.

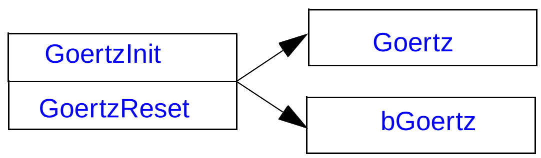
## DFT for a Given Frequency (Goertzel) Functions

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 SPL functions use a Goertzel algorithm and are more efficient when a small number of DFTs is needed. The Goertzel functions perform a primary initialization of the required data, repeated initialization to apply the algorithm for a new signal and unchanged frequency, a DFT computation for a single input signal, and a number of DFT computations for a block of input signals.

Figure 7-2 illustrates the order of use of the Goertzel functions.

**Figure 7-2 Order of Use of the Goertzel Functions**




Depending on the application, two modes of processing Goertzel signals can be implemented:

batch — The signal to be processed is finite and stored entirely in memory. Such a signal can be processed in "batch" mode, that is, all at once in a single (large) operation.

cyclic — The signal to be processed is not stored entirely in memory, either because it is too large, infinite in length, or the output is required before input is entirely known. Such a signal can be processed in "cyclic" mode, that is, in small pieces. In this case, a portion of the signal is read into memory, processed, and output. Then the process is repeated with the next portion.

The functions described in this section process a signal in the cyclic mode.

# bGoertz

*Computes the DFT for a block of successive samples for a given frequency.*

```
Scplx nspsbGoertz(NSPSGoertzState *stPtr, float *src, int len);
      /* real values; single precision */
Scplx nspcbGoertz(NSPCGoertzState *stPtr, SCplx *src, int len);
      /* complex values; single precision */
Dcplx nspdbGoertz(NSPDGoertzState *stPtr, double *src, int len);
      /* real values; double precision */
Dcplx nspzbGoertz(NSPZGoertzState *stPtr, DCplx *srcs, int len);
      /* complex values; double precision */
Wcplx nspwbGoertz(NSPWGoertzState *stPtr, short *src, int len,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
Wcplx nspvbGoertz(NSPVGoertzState *stPtr, WCplx *src, int len,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| `src` | Pointer to the array which stores the block of successive input samples. |
| `len` | The number of input samples in the `src` array. |
| `stPtr` | Pointer to the `NSP?GoertzState` structure. |
| `ScaleMode`, `ScaleFactor` | Refer to “Scaling Arguments” in Chapter 1. |

## Discussion

The `nsp?bGoertz()` function references the `NSP?GoertzState` structure for frequency, delay line and constants, and computes `len` DFTs for a block of successive input samples contained in the array `src`.

### Related Topics

| | |
|---|---|
| `Goertz` | Computes the DFT for a single signal count for a given frequency (see page 7-10). |
| `CoertzInit` | Initializes the frequency, delay line, and constants required for Goertzel functions (see page 7-11). |
| `GoertzReset` | Resets the internal delay line (see page 7-12). |

## Goertz

*Computes the DFT for a given frequency for a single signal sample.*

```
SCplx nspsGoertz(NSPSGoertzState *stPtr, float sample);
      /* real values; single precision */
DCplx nspcGoertz(NSPCGoertzState *stPtr, SCplx sample);
      /* complex values; single precision */
SCplx nspdGoertz(NSPDGoertzState *stPtr, double sample);
      /* real values; double precision */
DCplx nspzGoertz(NSPZGoertzState *stPtr, DCplx sample);
      /* complex values; double precision */
WCplx nspwGoertz(NSPWGoertzState *stPtr, short sample,
      int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
WCplx nspvGoertz(NSPVGoertzState *stPtr, WCplx sample,
      int ScaleMode, int *ScaleFactor);
      /* complex values; short integer */
```

| | |
|---|---|
| *sample* | The input sample to process. |
| *stPtr* | Pointer to the `NSP?GoertzState` structure. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

The `nsp?Goertz()` function computes a DFT for an single signal *`sample`*.

### Related Topics

| | |
|---|---|
| `bGoertz` | Computes the DFT for a block of signal counts for a given frequency (see page 7-9). |
| `GoertzInit` | Initializes the frequency, delay line, and constants required for Goertzel functions (see page 7-11). |
| `GoertzReset` | Resets the internal delay line (see page 7-12). |

## GoertzInit

*Initializes the frequency, delay line, and constants for Goertzel functions.*

```
void nspsGoertzInit(float freq, NSPSGoertzState *stPtr);
      /* real signal; single precision */
void nspcGoertzInit(float freq, NSPCGoertzState *stPtr);
      /* complex signal; single precision */
void nspdGoertzInit(double freq, NSPDGoertzState *stPtr);
      /* real signal; double precision */
void nspzGoertzInit(double freq, NSPZGoertzState *stPtr);
      /* complex signal; double precision */
void nspwGoertzInit(float freq, NSPWGoertzState *stPtr);
      /* real signal; short integer */
void nspvGoertzInit(float freq, NSPVGoertzState *stPtr);
      /* complex signal; short integer */
```

| | |
|---|---|
| *`freq`* | Normalized frequency value (0 < *`freq`* < 1.0), for which the DFT is computed. |
| *`stPtr`* | Pointer to the `NSP?GoertzState` structure. |

### Discussion

The `nsp?GoertzInit()` function initializes the `NSP?GoertzState` data structure which is used by other Goertzel functions. The function saves the frequency *`freq`* value and all required constants, and resets the internal delay line.

### Related Topics

| | |
|---|---|
| `bGoertz` | Computes the DFT for a block of successive samples for a given frequency (see page 7-9). |
| `Goertz` | Computes the DFT for a single signal and a given frequency (see page 7-10). |
| `GoertzReset` | Resets the internal delay line (see page 7-12). |

## GoertzReset

*Resets the internal delay line.*

```
void nspsGoertzReset(NSPSGoertzState *stPtr);
      /* real signal; single precision */
void nspcGoertzReset(NSPCGoertzState *stPtr);
      /* complex signal; single precision */
void nspdGoertzReset(NSPDGoertzState *stPtr);
      /* real signal; double precision */
void nspzGoertzReset(NSPZGoertzState *stPtr);
      /* complex signal; double precision */
void nspwGoertzReset(NSPWGoertzState *stPtr);
      /* real signal; short integer */
void nspvGoertzReset(NSPVGoertzState *stPtr);
      /* complex signal; short integer */
```

| | |
|---|---|
| *`stPtr`* | Pointer to the `NSP?GoertzState` structure. |

### Discussion

The `nsp?GoertzReset()` function resets the internal delay line contained in the `NSP?GoertzState` data structure. Resetting the delay line is necessary in order to repeat the algorithm execution without any change of frequency for which the DFT is to be computed.

### Related Topics

| | |
|---|---|
| `bGoertz` | Computes the DFT for a block of successive samples for a given frequency (see page 7-9). |
| `Goertz` | Computes the DFT for a single signal and a given frequency (see page 7-10). |
| `GoertzInit` | Initializes the frequency, delay line, and constants required for Goertzel functions (see page 7-11). |

**Example 7-2 Using Goertzel Functions for Selecting Magnitudes of a Given Frequency**

```
/* Compute DFT for selected frequency = 0.125 */

NSPSGoertz gs;

/* initialize and process sample-by-sample */

nspsGoertzInit(0.125, &gs);

for (n=0; n<2000; n++) {
     xval = /* insert code here to get the next input
               sample */
     x[n] = xval;
     dftval1 = nspsGoertz(&gs, xval);
}

/* re-initialize and process the whole block */

nspsGoertzReset(&gs);
dftval2 = nspsbGoertz(&gs, x, 2000);

/* dftval1 and dftval2 must be equal */
```

Example 7-2 illustrates the use of Goertzel functions for selecting the magnitudes of a given frequency when computing DFTs.

**Application Notes.** Each value of the DFT computed by the Goertzel algorithm takes 2`N`+2 real multiplications and 4`N` real additions. You can then use an FFT to compute the total DFT of the `N`log2(`N`) order of real multiplications and additions. Therefore, the Goertzel algorithm is efficient only if less then log2(`N`) input counts are necessary.

# Basic FFT Functions

The functions described in this section compute the forward or inverse complex fast Fourier transform of a signal.  The FFT produces identical results as the discrete Fourier transform (provided the length of the DFT is a power of 2), but is faster (see “Dft” in page 7-5 for a description of `nsp?Dft()`).  The length of the vector transformed by the FFT must be a power of 2.

If your application takes the FFTs of real signals or complex conjugate signals, you should consider using the FFT functions optimized for this purpose.  For example, see “RealFft” in page 7-38 and “CcsFft” in page 7-45 for a description of the `nsp?RealFft()` and `nsp?CcsFft()` functions. For more information on the FFT, see Appendix A.

## Flags Argument

The Fourier transform functions require you to specify the direction of the FFT and whether the input or output of the function is in bit-reversed order. Specify these items in the *`flags`* argument.  The *`flags`* argument is evaluated as the bitwise-OR of the values you enter.  The values you can enter for the *`flags`* argument are listed in Table 7-2.

You must specify one and only one of the `NSP_Forw`, `NSP_Inv` and `NSP_Init` values in the *`flags`* argument. You have the option of specifying none of the bit-reversal flag values, (`NSP_NoBitRev`, `NSP_InBitRev`, and `NSP_OutBitRev`) or you can specify, at most, one of them.  For example, it is not legal to specify both `NSP_InBitRev` and `NSP_OutBitRev`.  The default is no bit-reversal (`NSP_NoBitRev`).  The bit-reversal flags provide low-level access to the FFT algorithm.

**Table 7-2 Values for the flags Argument for the FFT Functions**

| Value | Description |
|---|---|
| NSP_Forw | Specifies a forward FFT with *inSamps[n]* (or *samps[n]*) providing *x(n)* on entry and *outSamps[k]* (or *samps[k]*) containing *X(k)* on exit. Note that the forward FFT is computed **without** the $1/N$ or $1/N^{1/2}$ normalization. |
| NSP_Free | Frees all internal arrays and twiddle arrays |
| NSP_InBitRev | Specifies that the input is in bit-reversed order and that the output should be generated in normal order. |
| NSP_Init | Specifies that the twiddle table and bit reversal table for this *order* FFT should be initialized; no other computation is performed and the other arguments (*inSamps*, *outSamps*, *samps*) are not referenced. Other flags are disregarded, if any. |
| NSP_Inv | Specifies an inverse FFT with *inSamps[k]* (or *samps[k]*) providing *X(k)* on entry and *outSamps[n]* (or *samps[n]*) containing *x(n)* on exit. |
| NSP_NoBitRev | Specifies that the input is in normal order and that the output should be generated in normal order. This is the default if neither NSP_InBitRev nor NSP_OutBitRev are specified. |
| NSP_NoScale | Specifies that the inverse transform will be done without the $1/N$ normalization. If this flag is not set, the $1/N$ normalization is performed. |
| NSP_OutBitRev | Specifies that the input is in normal order and that the output should be generated in bit-reversed order. |
| NSP_DoIntCore | Specifies that the integer core code will be used for any FFT order. See Application Note below. |
| NSP_DoFloatCore | Specifies that the float core code will be used for any FFT order. See Application Note below. |
| NSP_DoFastMMX | Specifies that the fast MMX™ technology algorithm will be used for any FFT order. See Application Note below. |

**Application Note:** If none of the `NSP_DoIntCore`, `NSP_DoFloatCore`, and `NSP_DoFastMMX` values is specified, float core code is used for FFT orders higher than `FFT_MMX_max_order`=7; otherwise integer code is used. The `NSP_DoIntCore` and `NSP_DoFastMMX` values are only valid for integer `w` and `v` data types. If you specify both `NSP_DoIntCore` and `NSP_DoFastMMX` values, integer core code is used for FFT orders higher than `FFT_MMX_Fast_max_order`=10; fast core calculations with MMX™ instructions will be used for order 10 or less.

## Fft, FftNip, rFft, rFftNip

*Computes the forward or inverse fast Fourier transform (FFT) of a signal.*

```
void nspcFft(SCplx *samps, int order, int flags);
void nspcrFft(float *reSamps, float *imSamps, int order, int flags);
void nspcFftNip(const SCplx *inSamps, SCplx *outSamps, int order,
      int flags);
void nspcrFftNip(const float *reInSamps, const float *imInSamps,
      float *reOutSamps, float *imOutSamps, int order, int flags);
      /*complex values; single precision */

void nspzFft(DCplx *samps, int order, int flags);
void nspzrFft(double *reSamps, double *imSamps, int order,
      int flags);
void nspzFftNip(const DCplx *inSamps, DCplx *outSamps, int order,
      int flags);
void nspzrFftNip(const double *reInSamps, const double *imInSamps,
      double *reOutSamps, double *imOutSamps, int order, int flags);
      /*complex values; double precision */

void nspvFft(WCplx *samps, int order, int flags, int ScaleMode,
      int *ScaleFactor);
void nspvrFft(short *reSamps, short *imSamps, int order, int flags,
      int ScaleMode, int *ScaleFactor);
```

```
void nspvFftNip(const WCplx *inSamps, WCplx *outSamps, int order,
      int flags, int ScaleMode, int *ScaleFactor);
void nspvrFftNip(const short *reInSamps, const short *imInSamps,
      short *reOutSamps, short *imOutSamps, int order, int flags,
      int ScaleMode, int *ScaleFactor);
      /*complex values; short integer */
```

| | |
|---|---|
| *flags* | Indicates the direction of the fast Fourier transform and whether bit-reversal is performed. The values for the *flags* argument are described in "Flags Argument." |
| *imInSamps* | Pointer to the real array which holds the imaginary part of the input to the `nsp?rFftNip()` function. The *imInSamps[n]* array must be of length $N = 2^{order}$. |
| *imOutSamps* | Pointer to the real array which holds the imaginary part of the output of the `nsp?rFftNip()` function. The *imOutSamps[n]* array must be of length $N = 2^{order}$. |
| *imSamps* | Pointer to the real array which holds the imaginary part of the input and output of the `nsp?rFft()` function. The *imSamps[n]* array must be of length $N = 2^{order}$. |
| *inSamps* | Pointer to the complex array which holds the input to the `nsp?FftNip()` function. The *inSamps[n]* array must be of length $N = 2^{order}$. |
| *order* | Base-2 logarithm of the number of samples in the FFT ($N$). |
| `outSamps` | Pointer to the complex array which holds the output from the `nsp?FftNip()` function. The *outSamps[n]* array must be of length $N = 2^{order}$. |
| *reInSamps* | Pointer to the real array which holds the real part of the input to the `nsp?rFftNip()` function. The *reInSamps[n]* array must be of length $N = 2^{order}$. |
| *reOutSamps* | Pointer to the real array which holds the real part of the output of the `nsp?rFftNip()` function. The *reOutSamps[n]* array must be of length $N = 2^{order}$. |

| | |
|---|---|
| *reSamps* | Pointer to the real array which holds the real part of the input and output of the `nsp?rFft()` function. The *reSamps[n]* array must be of length $N = 2^{order}$. |
| *samps* | Pointer to the complex array which holds the input and output samples for the `nsp?Fft()` function. The *samps[n]* array must be of length $N = 2^{order}$. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

**nsp?Fft().** The `nsp?Fft()` function computes a complex FFT in-place using the complex array *samps[n]* for input and output. This is functionally equivalent to `nsp?Dft()`, except that the DFT algorithm does not compute in-place. The length of the FFT must be a power of 2.

**nsp?FftNip().** The `nsp?FftNip()` function computes a complex FFT not-in-place, that is, it uses separate input and output arrays. The complex array *inSamps[n]* holds the input samples (time-domain for forward direction), and *outSamps[n]* holds the output samples (frequency-domain for forward direction). This is functionally equivalent to `nsp?Dft()`, except the length of the FFT must be a power of 2.

**nsp?rFft().** The `nsp?rFft()` function computes a complex FFT in-place, and places the real and imaginary parts into separate arrays. The real array *reSamps[n]* holds the real part, and the real array *imSamps[n]* holds the imaginary part. This form of the FFT is only used in special situations.

**nsp?rFftNip().** The `nsp?rFftNip()` function computes a complex FFT not-in-place. That is, on both input and output, it uses separate arrays for the real and imaginary parts. The arrays *reInSamps[n]* and *imInSamps[n]* hold the input samples, while the arrays *reOutSamps[n]* and *imOutSamps[n]* hold the output samples. This form of the FFT is only used in special situations.

Example 7-3 shows the code for standard fast Fourier transform usage.

**Example 7-3 Using nsp?FftNip() to Perform the FFT**

```
/* Calculate 128-point FFT of an input signal.
 * Input signal is in xTime, output is in xFreq.
 * "order" of FFT is 7 (log-base-2 of 128).
 */
DCplx        xTime[128], xFreq[128];

/* insert code here to put time-domain samples in xTime */

nspzFftNip(xTime, xFreq, 7, NSP_Forw);
/* xFreq now has frequency-domain samples */
```

Example 7-4 shows the code for using the FFT to low-pass filter.

**Example 7-4 Using nsp?FftNip() to Low-Pass Filter**

```
/* Low-pass filter an input signal by taking
 * its FFT, zeroing out the high frequency
 * components, and taking its inverse FFT. Input
 * signal is in "xTime" output is in "yTime."
 */
DCplx        xTime[128], xFreq[128], yTime[128];

nspzFftNip(xTime, xFreq, 7, NSP_Forw);
nspzZero(xFreq+32, 64); /* zero high frequencies */
nspzFftNip(xFreq, yTime, 7, NSP_Inv);
/* low-pass version of xTime is now in yTime */
```

Example 7-5 shows the code for using the FFT to implement the fast convolution of complex signals.

**Example 7-5 Using nsp?Fft() to Implement Fast Convolution**

```
/* Use a 256-point FFT to implement the fast
 * convolution of two complex signals. This
 * is accomplished by taking the FFTs of both
 * input signals (x and h), multiplying them
 * together in the frequency domain, and then
 * taking the inverse FFT of their product.
 */
DCplx        h[256], x[256];
/* insert code here to fill in h and x vectors */
nspzFft(h, 8, NSP_Forw|NSP_OutBitRev);
nspzFft(x, 8, NSP_Forw|NSP_OutBitRev);
nspzMpy2(h, x, 256);
nspzFft(x, 8, NSP_Inv|NSP_InBitRev);
/* x now contains the (circular) convolution of h and x */
```

### Application Notes

The algorithm for the bit-reversed, not in-place functions (that is, the `nsp?FftNip()` and `nsp?rFftNip()` functions with the `NSP_InBitRev` and `NSP_OutBitRev` flags specified) provide minimal performance advantage over the normal-ordered, not-in-place algorithm. This is because the library can optimize the normal-ordered, not-in-place algorithms by including the bit-reversal into the first FFT stage. In contrast, the bit-reversed, not-in-place combinations first copy the input array into the output array and then perform the computation in-place. Thus there is little reason for applications to use the bit-reversed forms unless bit-reversed data happens to be available.

# Low-Level FFTs of Real Signals

The functions described in this section provide a low-level interface to compute the FFT of real signals (in either the time- or frequency-domain). Real signals occur frequently in the real world. These functions exploit symmetry properties of the Fourier transform and compute the FFT of real signals much more efficiently than the FFT functions described in the previous section.

These functions are referred to as "low-level" because the results of the FFT are formatted in a somewhat complicated fashion. The results can be stored in either `RCPack` or `RCPerm` format. These formats arrange sequences of real and complex samples in ways which are more convenient for the FFT algorithms. For more information on these formats, see the sections RCPack Format and RCPerm Format later in this chapter. For the description of a higher level interface to the FFT algorithm, see "RealFft" in page 7-38 for information on `nsp?RealFft()`.

## Flags Argument

For low-level FFT functions, the *`flags`* argument must also declare if the output will be stored in `RCPack` or `RCPerm` format. This is in addition to the flag values described earlier in Flags Argument in the "Basic FFT Functions" section.

The `RCPack` and `RCPerm` flag values are described in Table 7-3. One of these flag values must be specified as one of the elements in the *`flags`* argument.

**Table 7-3 Flag Values for nsp?RealFftl() and nsp?RealFftlNip() Functions**

| Value | Description |
|---|---|
| `NSP_OutRCPack` | Specifies that the output array (*`samps[n]`* or *`outSamps[n]`*) should be arranged in `RCPack` format. |
| `NSP_OutRCPerm` | Specifies that the output array (*`samps[n]`* or *`outSamps[n]`*) should be arranged in `RCPerm` format. |

**NOTE.** *The bit-reversal flag values (*`NSP_NoBitRev`*,* `NSP_InBitRev`*, and* `NSP_OutBitRev`*) are not available for the* `nsp?RealFftl()` *and* `nsp?RealFftlNip()` *functions. This is because the algorithms used to compute real FFTs do not naturally use bit-reversed ordering.*

## Inverses of the Low-Level FFTs of Real Signals

The `nsp?RealFftl()` and `nsp?RealFftlNip()` functions do not provide their own inverses. Instead, the inverses are provided by the `nsp?CcsFftl()` and `nsp?CcsFftlNip()` functions.

For example, calling `nspdRealFftl()` with the `NSP_Forw` flag transforms a real time-domain signal into a conjugate-symmetric frequency-domain signal. The function `nspdCcsFftl()` called with the `NSP_Inv` flag can then be used to transform it back to the original, real time-domain signal. In typical signal processing, these two operations (real time-domain to conjugate-symmetric frequency-domain and back) are more frequently used than the other two operations (conjugate-symmetric time-domain to real frequency-domain forward and back). For more information about inverses of Fourier transforms see Appendix A.

## RealFftl, RealFftlNip

*Computes the forward or inverse FFT of a real signal using `RCPack` or `RCPerm` format.*

```
void nspsRealFftl(float *samps, int order, int flags);
void nspsRealFftlNip(const float *inSamps, float *outSamps,
      int order, int flags);
      /* real values, single precision */
void nspdRealFftl(double *samps, int order, int flags);
void nspdRealFftlNip(const double *inSamps, double *outSamps,
      int order, int flags);
      /* real values, double precision */
void nspwRealFftl(short *samps, int order, int flags,
      int ScaleMode, int *ScaleFactor);
void nspwRealFftlNip(const short *inSamps, short *outSamps,
      int order, int flags, int ScaleMode, int *ScaleFactor);
      /* real values, short integer */
```

| | |
|---|---|
| *flags* | Indicates the direction of the fast Fourier transform, whether bit-reversal is performed, and the packing type for the function. The argument consists of the bitwise-OR of one or more flags. One and only one of the flag values `NSP_Forw`, `NSP_Inv`, and `NSP_Init` must be specified. The `NSP_NoScale` flag is optional. The values for the *flags* argument are described in Table 7-2, the "Basic FFT Functions" section, and Table 7-3, the "Low-Level FFTs of Real Signal" section. |
| *inSamps* | Pointer to the real array which holds the input to the `nsp?RealFftlNip()` function. The *inSamps[n]* array must be of length $N = 2^{order}$. |
| *order* | The base-2 logarithm of the number of samples in the FFT ($N$). |

*outSamps* Pointer to the real array which holds the output from the nsp?RealFftlNip() function. The *outSamps[n]* array must be of length $N = 2^{order}$.

*samps* Pointer to the real array which holds the input and output samples for the nsp?RealFftl() function. The *samps[n]* array must be of length $N = 2^{order}$.

*ScaleMode*, *ScaleFactor* Refer to "Scaling Arguments" in Chapter 1.

## Discussion

**nsp?RealFftl()**. The nsp?RealFftl() function computes the FFT in-place. In the forward direction (*flags* = NSP_Forw), the array *samps[n]* contains *N* real, time-domain samples that define an *N*-length sequence *x(n)*. On exit, *samps[n]* contains *N* real values in either RCPack or RCPerm format that describe the forward FFT of *x(n)*.

In the inverse direction (*flags* = NSP_Inv), *samps[k]* contains *N* real frequency-domain samples that define a *N*-length sequence *X(k)*. On exit, *samps[k]* contains *N* real values, in either RCPack or RCPerm format, that describe the inverse FFT of *X(k)*.

**nsp?RealFftlNip()**. The nsp?RealFftlNip() function computes the FFT not-in-place. In the forward direction (*flags* = NSP_Forw), the input array *inSamps[n]* contains *N* real, time-domain samples that define a *N*-length sequence *x(n)*. On exit, the output array *outSamps[n]* contains *N* real values in either RCPack or RCPerm format that describe the forward FFT of *x(n)*.

In the inverse direction (*flags* = NSP_Inv), the input array *inSamps[k]* contains *N* real frequency-domain samples that define a *N*-length sequence *X(k)*. On exit, the output array *outSamps[k]* contains *N* real values, in either RCPack or RCPerm format, that describe the inverse FFT of *X(k)*.

## RCPack Format

This discussion, and the notation used in this section, assumes a forward FFT; when considering an inverse FFT, just replace `X()` by `x()`. In either case, since the input is real valued, the output will be complex conjugate-symmetric. Thus, the result of the fast Fourier transform can be described by ($N$/2) + 1 complex samples. But, since the first sample $X$(0), and the middle sample $X$($N$/2) are real, the result of the fast Fourier transform can be more compactly described by two real samples and $N$/2 - 1 complex samples.

The `RCPack` format is a convenient, compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms ("natural" in the sense that bit-reversed order is natural for radix-2 complex FFTs). In the `RCPack` format, the output samples of the FFT are arranged as follows:

**Table 7-4 Arrangement of Samples in RCPack Format**

| Index | Contents |
|---|---|
| 0 | $X(0)$ |
| 1 | $X(1)_R$ |
| 2 | $X(1)_I$ |
| 3 | $X(2)_R$ |
| 4 | $X(2)_I$ |
| . . . | . . . |
| N-3 | $X(N/2 - 1)_R$ |
| N-2 | $X(N/2 - 1)_I$ |
| N-1 | $X(N/2)$ |

The complete $N$-length FFT is then given by the following equation:

$$X(k) = \begin{cases} \mathtt{samps}[0],\ k = 0 \\ \mathtt{samps}[2k-1] + j \cdot \mathtt{samps}[2k],\ 1 \le k < \dfrac{N}{2} \\ \mathtt{samps}[N-1],\ k = \dfrac{N}{2} \\ X(N-k)^*,\ \dfrac{N}{2} < k < N \end{cases}$$

## RCPerm Format

The RCPerm format stores the values in the order in which the FFT algorithm uses them. This is the most natural way of storing values for the FFT algorithm. The RCPerm format is an arbitrary permutation of the RCPack format. An important characteristic of the RCPerm format is that the real and imaginary parts of a given sample need not be adjacent.

The NSP_OutRCPerm value for the *flags* argument results in the fastest possible FFT. However, the order of the samples in the output array are completely implementation-dependent. As a result, application programs have no way of interpreting this data.

Even though the data stored in RCPerm format cannot be interpreted, it can still be used. The Intel Signal Processing Library provides functions that allow applications to use RCPerm format for fast convolution. For example, you can use the nsp?MpyRCPerm3() function to multiply two vectors stored in RCPerm format to create a third vector also in RCPerm format. You can then use the nsp?CcsFftl() function to convert this vector back to a naturally-ordered, time-domain vector.

The following examples illustrate several different applications of the nsp?RealFftl() and nsp?RealFftlNip() functions.

Example 7-6 shows the code for performing the forward and inverse FFT.

### Example 7-6 Using nsp?RealFftl() to Perform the Forward and Inverse FFT

```
/*
 * The following code performs an elaborate
 * “do nothing” operation to illustrate the
 * appropriate calling sequences and flags.
 * It first takes a forward 64-point real
 * FFT of the input signal (x). This is done
 * in-place. It then calculates an inverse FFT,
 * depositing the results back in x.
 */
float        x[64];
/* fill in time-domain samples of x */
nspsRealFftl(x, 6, NSP_Forw|NSP_OutRCPack);
nspsCcsFftl(x, 6, NSP_Inv|NSP_InRCPack);
/* x is now the same as when you started */
```

Example 7-7 shows the code for using the FFT to perform low-pass filtering.

### Example 7-7 Using nsp?RealFftlNip() to Perform Low-Pass Filtering

```
/*
 * Low-pass filter an input signal using real
 * FFTs. This is accomplished by taking a
 * 128-point real FFT of the input signal (xTime)
 * and storing the result in xFreq. The higher
 * frequencies in xFreq are then set to zero, and
 * the inverse FFT (that is, the low-pass signal)
 * is stored in yTime.
 */
double xTime[128], xFreq[128], yTime[128];
/* insert code here to fill in 128 samples of xTime */
nspdRealFftlNip(xTime, xFreq, 7, NSP_Forw|NSP_OutRCPack);
nspdbZero(xFreq+63, 65); /* zero high freqs f=0.25 to 0.5 */
nspdCcsFftlNip(xFreq, yTime, 7, NSP_Inv|NSP_InRCPack);
/* low-pass version of xTime is now in yTime */
```

Example 7-8 is similar to the previous example, but the low-pass filtering is performed in-place.

### Example 7-8 Using nsp?RealFftl() to Perform Low-Pass Filtering In-Place

```
/*
 * Low-pass filter an input signal in-place
 * using real FFTs. This is accomplished by
 * taking a 128-point real FFT of the input
 * signal (x). The higher frequencies in x
 * are then set to zero, and the inverse FFT
 * (that is, the low-pass signal)is stored in x.
 */
double  x[128];
/* insert code to fill in 128 samples of x */
nspdRealFftl(x, 7, NSP_Forw|NSP_OutRCPack);
nspdbZero(x+63, 65); /* zero high freqs */
nspdCcsFftl(x, 7, NSP_Inv|NSP_InRCPack);
/* low-pass version now in x */
```

Example 7-9 shows the code for using the FFT for the fast convolution of two signals.

### Example 7-9 Using nsp?RealFftlNip() for the Fast Convolution of Real Signals

```
/*
 Perform the fast convolution of two real signals
 * by using real-valued 256-point FFTs. The FFTs of
 * the real-valued input signals (x and h) are computed
 * and stored in xFreq and hFreq in RCPerm format.
 * These are then multiplied using the MpyRCPerm3
 * function. The product is then inverse FFT'd and
 * stored in yTime.*/
double       hTime[256], hFreq[256];
double       xTime[256], xFreq[256], yTime[256], yFreq[256];
/* insert code here to fill in hTime and xTime vectors */
nspdRealFftlNip(hTime, hFreq, 8, NSP_Forw|NSP_OutRCPerm);
nspdRealFftlNip(xTime, xFreq, 8, NSP_Forw|NSP_OutRCPerm);
nspdMpyRCPerm3(hFreq, xFreq, yFreq, 8); /* y=h*x */
nspdCcsFftlNip(yFreq, yTime, 8, NSP_Inv|NSP_InRCPerm);
/* y now contains the (circular) convolution of h and x */
```

Example 7-10 is similar to the previous example, but the fast convolution of two signals is performed in-place.

### Example 7-10 Using nsp?RealFftl() for the Fast Convolution of Real Signals In-Place

```
/*
 * Perform the fast convolution of two real signals
 * in-place by using real-valued 256-point FFTs. The
 * FFTs of the real-valued input signals (x and h)
 * are computed and stored in RCPerm format. These
 * are then multiplied using the MpyRCPerm2 function.
 * The product is then inverse FFT'd and stored in x.
 */
double h[256], x[256];
/* insert code here to fill in h and x vectors */
nspdRealFftl(h, 8, NSP_Forw|NSP_OutRCPerm);
nspdRealFftl(x, 8, NSP_Forw|NSP_OutRCPerm);
nspdMpyRCPerm2(h, x, 8); /* multiply h into x */
nspdCcsFftl(x, 8, NSP_Inv|NSP_InRCPerm);
/* x now contains the (circular) convolution of h and x */
```

## Related Topics

| | |
|---|---|
| `CcsFftl` | Provides the inverse to the `nsp?RealFftl()` function (see page 7-35). |
| `CcsFftlNip` | Provides the inverse to the `nsp?RealFftlNip()` function (see page 7-35). |
| `MpyRCPack2` | Multiplies two vectors stored in `RCPack` format (see page 7-30). |
| `MpyRCPerm2` | Multiplies two vectors stored in `RCPerm` format (see page 7-32). |
| `RealFft` | Provides a higher level interface to the real FFT algorithms without the complications of `RCPack` and `RCPerm` formats (see page 7-38). |

| | |
|---|---|
| `RealFftNip` | Provides a higher level interface to the real FFT algorithms without the complications of `RCPack` and `RCPerm` formats (see page 7-38). |

See [Mit93], section 8-2-9, *Real-Valued FFTs*, for more information on the fast Fourier transforms of real signals.

## Vector Multiplication in RCPack or RCPerm Format

The functions described in this section perform the element-wise complex multiplication of vectors stored in `RCPack` or `RCPerm` formats. These functions are used with the `nsp?RealFftl()` and `nsp?CcsFftl()` functions to perform fast convolution on real signals.

The standard vector multiplication `nsp?bMpy2()` function cannot be used to multiply `RCPack` or `RCPerm` format vectors because:

- Two real samples are stored in the `RCPack` format.
- The `RCPerm` format might not pair the real parts of a signal with their corresponding imaginary parts.

The argument *order* indicates base-2 logarithm of the length of the FFT, $N$, where $N = 2^{order}$.

## MpyRCPack2, MpyRCPack3

*Multiplies two vectors stored in `RCPack` format.*

```
void nspsMpyRCPack2(const float *src, float *dst, int order);
void nspsMpyRCPack3(const float *srcA, const float *srcB,
      float *dst, int order);
      /* real values; single precesion */
void nspdMpyRcPack2(const double *src, double *dst, int order);
void nspdMpyRcPack3(const double *srcA, const double *srcB,
      double *dst, int order);
      /* real values; double precesion */
```

```
void nspwMpyRCPack2(const short *src, short *dst, int order,
      int ScaleMode, int *ScaleFactor);
void nspwMpyRCPack3(const short *srcA, const short *srcB,
      short *dst, int order, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| `dst` | Pointer to the vector which:<br>• holds the result of the multiplication (`src[n]` * `dst[n]`) for the `nsp?MpyRCPack2()` function.<br>• holds the result of the multiplication (`srcA[n]` * `srcB[n]`) for the `nsp?MpyRCPack3()` function.<br>The vector must be of length $N = 2^{order}$. |
| `order` | The base-2 logarithm of the number of samples in the FFT ($N$). |
| `src` | Pointer to the vector to be multiplied to `dst[n]`. The vector must be of length $N = 2^{order}$. |
| `srcA, srcB` | Pointers to the vectors to be multiplied together. The vectors must be of length $N = 2^{order}$. |
| `ScaleMode`, `ScaleFactor` | Refer to “Scaling Arguments” in Chapter 1. |

## Discussion

**`nsp?MpyRCPack2()`**. The `nsp?MpyRCPack2()` function multiplies the vector `src[n]` with `dst[n]` and stores the result into `dst[n]`.

**`nsp?MpyRCPack3()`**. The `nsp?MpyRCPack3()` function multiplies the vector `srcA[n]` with `srcB[n]` and stores the result into `dst[n]`.

## Related Topics

| | |
|---|---|
| `MpyRCPerm2` | Multiplies two vectors in `RCPerm` format (see page 7-32). |
| `MpyRCPerm3` | Multiplies two vectors in `RCPerm` format and stores the result in a third vector (see page 7-32). |

## MpyRCPerm2, MpyRCPerm3

*Multiplies two vectors stored in* `RCPerm` *format.*

```
void nspsMpyRCPerm2(const float *src, float *dst, int order);
void nspsMpyRCPerm3(const float *srcA, const float *srcB,
      float *dst, int order);
      /* real values; single precision */
void nspdMpyRCPerm2(const double *src, double *dst, int order);
void nspdMpyRCPerm3(const double *srcA, const double *srcB,
      double *dst, int order);
      /* real values; double precision */
void nspwMpyRCPerm2(const short *src, short *dst, int order,
      int ScaleMode, int *ScaleFactor);
void nspwMpyRCPerm3(const short *srcA, const short *srcB,
      short *dst, int order, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| *dst* | Pointer to the vector which:<br>• holds the result of the multiplication (*src[n]* * *dst[n]*) for the `nsp?MpyRCPerm2()` function.<br>• holds the result of the multiplication (*srcA[n]* * *srcB[n]*) for the `nsp?MpyRCPerm3()` function.<br>The vector must be of length $N = 2^{order}$. |
| *order* | The base-2 logarithm of the number of samples in the FFT ($N$). |
| *src* | Pointer to the vector to be multiplied to *dst[n]*. The vector must be of length $N = 2^{order}$. |

| | |
|---|---|
| *srcA*, *srcB* | Pointers to the vectors to be multiplied together. The vectors must be of length $N = 2^{order}$. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

**nsp?MpyRCPerm2()**. The function nsp?MpyRCPerm2() multiplies the vector *src[n]* with *dst[n]* and stores the result into *dst[n]*.

**nsp?MpyRCPerm3()**. The function nsp?MpyRCPerm3() multiplies the vector *srcA[n]* with *srcB[n]* and stores the result into *dst[n]*.

For an example of the use of the nsp?MpyRCPerm2() and nsp?MpyRCPerm3() functions, see Example 7-8 and Example 7-9.

### Related Topics

| | |
|---|---|
| MpyRCPack2 | Multiplies two vectors in RCPack format (see page 7-30). |
| MpyRCPack3 | Multiplies two vectors in RCPack format and stores the result in a third vector (see page 7-30). |

## Low-Level FFTs of Conjugate-Symmetric Signals

The functions described in this section provide a low-level interface to compute the FFT of conjugate-symmetric signals (in either time- or frequency-domain). These functions exploit symmetry properties of the Fourier transform and are significantly faster than the standard complex FFT.

The functions are referred to as "low-level" because the results are formatted in a somewhat complicated fashion. The results can be stored in either RCPack or RCPerm format. These formats are ways of arranging sequences of real and complex samples which are more convenient for the FFT algorithms. For more information on these formats, see RCPack Format" and RCPerm Format." For the description of a higher level interface to the FFT algorithm, see "CcsFft" in page 7-45 for information on nsp?CcsFft().

## Flags Argument

For low-level functions, the *flags* argument must also declare if the input is stored in RCPack or RCPerm format. This is in addition to the flag values described in Flags Argument, the “Basic FFT Functions” section.

The RCPack and RCPerm format flag values are described in Table 7-5. One of these flag values must be specified in the *flags* argument.

**Table 7-5 Flag Values for nsp?CcsFftl() and nsp?CcsFftlNip() Functions**

| Value | Description |
|---|---|
| NSP_InRCPack | Specifies that the input array (*samps[n]* or *inSamps[n]*) should be arranged in RCPack format. |
| NSP_InRCPerm | Specifies that the input array (*samps[n]* or *inSamps[n]*) should be arranged in RCPerm format. |

## Inverses of FFTs of Low-Level Conjugate-Symmetric Signals

The functions described in this section, nsp?CcsFftl() and nsp?CcsFftlNip(), do not provide their own inverses. Instead, the inverses are provided by the nsp?RealFftl() and nsp?RealFftlNip() functions.

For example, nspdCcsFftl() with the NSP_Forw flag transforms a conjugate-symmetric time-domain signal into a real frequency-domain signal, and nspdRealFftl() with the NSP_Inv flag transforms it back to the original, conjugate-symmetric time-domain signal. For further discussion of inverses of Fourier transform functions, see Appendix A.

## CcsFftl, CcsFftlNip

*Computes the forward or inverse FFT of a complex conjugate-symmetric (CCS) signal using `RCPack` or `RCPerm` format.*

```
void nspsCcsFftl(float *samps, int order, int flags);
void nspsCcsFftlNip(const float *inSamps, float *outSamps,
      int order, int flags);
      /* real values, single precision */
void nspdCcsFftl(double *samps, int order, int flags);
void nspdCcsFftlNip(const double *inSamps, double *outSamps,
      int order, int flags);
      /* real values, double precision */
void nspwCcsFftl(short *samps, int order, int flags, int scaleMode,
      int *ScaleFactor);
void nspwCcsFftlNip(const short *inSamps, short *outSamps,
      int order, int flags, int scaleMode, int *ScaleFactor);
      /* real values, short integer */
```

| | |
|---|---|
| *`flags`* | Indicates the direction of the fast Fourier transform, whether bit-reversal is to be performed, and the packing format. The argument consists of the bitwise-OR of one or more flags. One and only one of the flag values `NSP_Forw`, `NSP_Inv`, and `NSP_Init` must be specified. The `NSP_NoScale` flag value is optional. The values for the *`flags`* argument are described in Flags Argument, the "Basic FFT Functions" section, and Flags Argument, the "Low-Level FFTs of Conjugate-Symmetric Signals" section. |
| *`inSamps`* | Pointer to the real array which holds the input to the `nsp?CcsFftlNip()` function. The *`inSamps[n]`* array must be of length $N = 2^{order}$. |
| *`order`* | Base-2 logarithm of the number of samples in FFT($N$). |

| | |
|---|---|
| *outSamps* | Pointer to the real array which holds the output from the nsp?CcsFftlNip() function. The *outSamps[n]* array must be of length $N = 2^{order}$. |
| *samps* | Pointer to the real array which holds the input and output samples for the nsp?CcsFftl() function. The *samps[n]* array must be of length $N = 2^{order}$. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

**nsp?CcsFftl()**. The nsp?CcsFftl() function computes the FFT in-place. In the forward direction (*flags* = NSP_Forw), the array *samps[n]* contains *N* real values in either RCPack or RCPerm format. These values describe a complex conjugate-symmetric time-domain signal *x(n)*. On exit, *samps[n]* contains *N* real frequency-domain samples that are the forward FFT of *x(n)*.

In the inverse direction (*flags* = NSP_Inv), the array *samps[n]* contains *N* real values in either RCPack or RCPerm format. The values describe a complex conjugate-symmetric frequency-domain signal *X(k)*. On exit, *samps[n]* contains *N* real time-domain samples that are the inverse FFT of *X(k)*.

**nsp?CcsFftlNip()**. The function nsp?CcsFftlNip() computes the FFT not-in-place. In the forward direction (*flags* = NSP_Forw), the input array *inSamps[n]* contains *N* real values in either RCPack or RCPerm format. These values describe a complex conjugate-symmetric time-domain signal *x(n)*. On exit, the output array *outSamps[n]* contains *N* real frequency-domain samples that are the forward FFT of *x(n)*.

In the inverse direction (*flags* = NSP_Inv), the input array *inSamps[n]* contains *N* real values in either RCPack or RCPerm format that describe a complex conjugate-symmetric frequency-domain signal *X(k)*. On exit, the output array *outSamps[n]* contains *N* real time-domain samples that are the inverse FFT of *X(k)*.

### Related Topics

| | |
|---|---|
| `CcsFft` | Provides a higher level interface to the FFT without the complications of `RCPerm` and `RCPack` formats (see page 7-45). |
| `CcsFftNip` | Provides a higher level interface to the FFT without the complications of `RCPerm` and `RCPack` formats (see page 7-45). |
| `RealFftl` | Provides the functional inverse to the `nsp?CcsFftl()` function (see page 7-23). |
| `RealFftlNip` | Provides the functional inverse to the `nsp?CcsFftlNip()` function (see page 7-23). |

# FFTs of Real Signals

The functions described in this section compute the FFT of real signals (either in the time- or frequency-domain), yielding a complex conjugate-symmetric signal. These functions exploit symmetry properties of the Fourier transform and are significantly faster than the standard FFT.

The `nsp?RealFft()` and `nsp?RealFftNip()` functions store the real samples in `RCCcs` format. This is a simpler and easier to use format than the `RCPack` and `RCPerm` formats used by `nsp?RealFftl()` and `nsp?RealFftlNip()`. However, `RCCcs` format requires slightly more memory. The arrangement of samples in `RCCcs` format is described in Table 7-6.

## Inverses of FFTs of Real Signals

The `nsp?RealFft()` and `nsp?RealFftNip()` functions do not provide their own inverses. Rather, the inverses are provided by the `nsp?CcsFft()` and `nsp?CcsFftNip()` functions.

For example, `nspdRealFft()` called with the `NSP_Forw` flag transforms a real time-domain signal into a conjugate-symmetric frequency-domain signal, and `nspdCcsFft()` called with the `NSP_Inv` flag transforms it back to the original, real time-domain signal. In typical signal processing, these two operations (real time-domain to conjugate-symmetric frequency and back) are more frequently used than the other two operations (conjugate-symmetric time-domain to real frequency-domain forward and back). For further discussion of the inverses of Fourier transform functions, see Appendix A.

## RealFft, RealFftNip

*Computes the forward or inverse FFT of a real signal.*

```
void nspsRealFft(float *samps, int order, int flags);
void nspsRealFftNip(const float *inSamps, SCplx *outSamps,
      int order, int flags);
      /* real values; single precision */
void nspdRealFft(double *samps, int order, int flags);
void nspdRealFftNip(const double *inSamps, DCplx *outSamps,
      int order, int flags);
      /* real values; double precision */
void nspwRealFft(short *samps, int order, int flags, int ScaleMode,
      int *ScaleFactor);
void nspwRealFftNip(const short *inSamps, WCplx *outSamps,
      int order, int flags, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
```

`flags` Indicates the direction of the fast Fourier transform and whether bit-reversal is performed. The argument consists of the bitwise-OR of one or more flags. One and only one of the flag values `NSP_Forw`, `NSP_Inv`, and `NSP_Init` must be specified. The `NSP_NoScale` flag is optional. The section Flags Argument in “Basic FFT Functions” describes the values for the `flags` argument.

`inSamps` Pointer to the real array which holds the input to the `nsp?RealFftNip()` function. The `inSamps[n]` array must be of length $N = 2^{order}$.

| | |
|---|---|
| *order* | The Base-2 logarithm of the number of samples in the FFT (*N*). |
| *outSamps* | Pointer to the complex array which holds the output from the `nsp?RealFftNip()` function. The *outSamps[n]* array must be in `RCCcs` format and be of length *N/2+1* complex samples. |
| *samps* | Pointer to the real array which holds the input and output samples for the `nsp?RealFft()` function. The *samps[n]* array must be of length *N* + 2 elements (`floats` or `doubles`). On input, *samps[n]* should be considered a real array, the first *N* elements of which are data and the last two elements are ignored. On output, *samps[n]* should be considered a complex array of length *N*/2 + 1 complex samples in `RCCcs` format. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

**`nsp?RealFft()`.** The `nsp?RealFft()` function performs the FFT in-place. In the forward direction (*flags* = `NSP_Forw`), *samps[n]* contains *N* real, time-domain samples that define an *N*-length sequence *x(n)*. On exit, *samps[n]* contains *N*/2 + 1 complex samples in `RCCcs` format that describe the forward FFT of *x(n)*.

In the inverse direction (*flags* = `NSP_Inv`), *samps[k]* contains *N* real frequency-domain samples that define a *N*-length sequence *X(k)*. On exit, *samps[n]* contains *N*/2 + 1 complex samples in `RCCcs` format that describe the inverse FFT of *X(k)*.

There are two requirements for the length of *samps[n]*:

- The array *samps[n]* must be of length *N* + *2* real elements so that it can contain the *N*/2 + 1 complex numbers that are returned. The two extra elements (at the end of the array) are ignored on input.

- Upon return, the array *samps[n]* should be treated as an array of $N/2 + 1$ complex numbers rather than an array of real numbers. This can be done by appropriate casting. The complex elements $X(0)$ to $X(N/2)$ span normalized frequency or, in the case of an inverse FFT, normalized time from 0.0 to 0.5.

**nsp?RealFftNip().** The nsp?RealFftNip() function computes the FFT not-in-place. In the forward direction (*flags* = NSP_Forw), the input array *inSamps[n]* contains $N$ real, time-domain samples that define an $N$-length sequence *x(n)*. On exit, the output array *outSamps[n]* contains $N/2 + 1$ complex samples in RCCcs format that describe the forward FFT of *x(n)*. The forward FFT of *x(n)* is defined as follows:

$$X(k) = \begin{cases} outsamps[k], & 0 \le k \le \frac{N}{2} \\ X(N-k)^*, & \frac{N}{2} < k < N \end{cases}$$

In the inverse direction (*flags* = NSP_Inv), the input array *inSamps(n)* contains $N$ real frequency-domain samples that define a $N$-length sequence *X(k)*. On exit, *outSamps(n)* contains $N/2 + 1$ complex samples in RCCcs format that describe the inverse FFT of *X(k)*. The inverse FFT of *X(k)* is defined as follows:

$$x(n) = \begin{cases} outsamps[n], & 0 \le n \le \frac{N}{2} \\ X(N-n)^*, & \frac{N}{2} < n < N \end{cases}$$

Table 7-6 describes the arrangement of samples in RCCcs format.

**Table 7-6 Arrangement of Samples in RCCcs Format**

| Real Index | Complex Index | Contents |
|---|---|---|
| 0 | 0 | $X(0)_R$ |
| 1 | 0 | $X(0)_I$ |
| 2 | 1 | $X(1)_R$ |
| 3 | 1 | $X(1)_I$ |
| . . . | . . . | . . . |
| $N$ - 2 | $N$/2 - 1 | $X(N/2 - 1)_R$ |
| $N$ - 1 | $N$/2 - 1 | $X(N/2 - 1)_I$ |
| $N$ | $N$/2 | $X(N/2)_R$ |
| $N$ + 1 | $N$/2 | $X(N/2)_I$ |

The following examples illustrate the use of the `nsp?RealFft()` and `nsp?RealFftNip()` functions.

Example 7-11 shows the code to perform the FFT of a real signal.

**Example 7-11 Using nsp?RealFftNip() to Take the FFT of a Real Signal**

```
/* take the FFT of a
 * real signal
 */
double       xTime[128];
DCplx        xFreq[65], xFreqFull[128];
/* insert code here to put time-domain samples in xTime */
nspdRealFftNip(xTime, xFreq, 7, NSP_Forw);
/* xFreq now has frequency-domain samples from f=0.0 to 0.5 */

nspzbConjExtend2(xFreq, xFreqFull, 65);
/* xFreqFull contains freqency samples from f=0.0 to 1.0 */
```

Example 7-12 shows the code to perform low-pass filtering.

### Example 7-12 Using nsp?RealFftNip() to Perform Low-Pass Filtering

```
/* use FFT functions to perform
 * low-pass filtering
 */
double  xTime[128], yTime[128];
DCplx        xFreq[65];
/* insert code here to fill in 128 samples of xTime */
nspdRealFftNip(xTime, xFreq, 7, NSP_Forw);
nspzbZero(xFreq+33, 32); /* zero high freqs f=0.25 to 0.5 */
nspdCcsFftNip(xFreq, yTime, 7, NSP_Inv);
/* low-pass version of xTime is now in yTime */
```

Example 7-13 is similar to the previous example, except the low-pass filtering is performed in-place.

### Example 7-13 Using nsp?RealFft() to Perform Low-Pass Filtering In-Place

```
/* use the FFT functions to perform low-pass
 * filtering in-place
 */
double x[130];
/* insert code to fill in 128 samples of x */
nspdRealFft(x, 7, NSP_Forw);
nspzZero(((DCplx*)xTime)+33, 32); /* zero high freqs */
nspdCcsFft(x, 7, NSP_Inv);
/* low-pass version now in x */
```

Example 7-14 shows the code to perform the fast convolution of two real signals.

**Example 7-14 Using nsp?RealFftNip() to Perform Fast Convolution**

```
/* use the FFT functions to perform fast
 * convolution of real signals
 */
double hTime[256], xTime[256], yTime[256];
DCplx        hFreq[129], xFreq[129], yFreq[129];
/* insert code here to fill in hTime and xTime vectors */
nspdRealFftNip(hTime, hFreq, 8, NSP_Forw);
nspdRealFftNip(xTime, xFreq, 8, NSP_Forw);
nspzbMpy3(hFreq, xFreq, yFreq, 129);  /* y=h*x */
nspdCcsFftNip(yFreq, yTime, 8, NSP_Inv);
/* y now contains the (circular) convolution of h and x */
```

Example 7-15 is similar to the previous example, except the fast convolution is performed in-place.

**Example 7-15 Using nsp?RealFft() to Perform Fast Convolution In-Place**

```
/* use the FFT functions to perform fast
 * convolution of real signals in-place
 */
double h[258], x[258];
/* insert code here to fill 256 samples of h and x vectors */
nspdRealFft(h, 8, NSP_Forw);
nspdRealFft(x, 8, NSP_Forw);
nspzbMpy2((DCplx*)h, (DCplx*)x, 129); /* x = h*x */
nspdCcsFft(x, 8, NSP_Inv);
/* x now contains the (circular) convolution of h and x */
```

### Related Topics

| | |
|---|---|
| `CcsFft` | Provides the inverse to the `nsp?RealFft()` function (see page 7-45). |
| `CcsFftNip` | Provides the inverse to the `nsp?RealFftNip()` function (see page 7-45). |

| | |
|---|---|
| `bConjExtend1` | Extends the output arrays produced by the `nsp?RealFft()` and `nsp?RealFftNip()` functions into full *N*-length signals (see page 3-52). |
| `RealFftl` | Provides a lower-level interface to the FFT algorithm (see page 7-23). |

See [Mit93], section 8-2-9, *Real-Valued FFTs*, for more information about real-valued fast Fourier transform.

# FFTs of Conjugate-Symmetric Signals

The functions described in this section compute the FFT of complex conjugate-symmetric signals (time- or frequency-domain), yielding a real signal. These functions exploit symmetry properties of the Fourier transform and are significantly faster than the standard complex FFT.

The `nsp?CcsFft()` and `nsp?CcsFftNip()` functions store the complex conjugate-symmetric samples in `RCCcs` format. This is a simpler and easier to use format than the `RCPack` and `RCPerm` formats used by `nsp?CcsFftl()` and `nsp?CcsFftlNip()`. However, `RCCcs` format requires slightly more memory. The arrangement of samples in `RCCcs` format is described in Table 7-6.

## Inverses of FFTs of Conjugate-Symmetric Signals

These `nsp?CcsFft()` and `nsp?CcsFftNip()` functions do not provide their own inverses. Instead, the inverses are provided by the `nsp?RealFft()` and `nsp?RealFftNip()` functions.

For example, `nspdCssFft()` called with the `NSP_Forw` flag transforms a conjugate-symmetric time-domain signal into a real frequency-domain signal, and `nspdRealFft()` called with the `NSP_Inv` flag transforms it back to the original, conjugate-symmetric time-domain signal. For more information about inverses of Fourier transform functions, see Appendix A.

## CcsFft, CcsFftNip

*Computes the forward or inverse FFT of a complex conjugate-symmetric (CCS) signal.*

```
void nspsCcsFft(float *samps, int order, int flags);
void nspsCcsFftNip(const SCplx *inSamps, float *outSamps,
      int order, int flags);
      /* real values; single precision */
void nspdCcsFft(double *samps, int order, int flags);
void nspdCcsFftNip(const DCplx *inSamps, double *outSamps,
      int order, int flags);
      /* real values; double precision */
void nspwCcsFft(short *samps, int order, int flags, int ScaleMode,
      int *ScaleFactor);
void nspwCcsFftNip(const WCplx *inSamps, short *outSamps,
      int order, int flags, int ScaleMode, int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| *flags* | Indicates the direction of the fast Fourier transform and whether bit-reversal is performed. The argument consists of the bitwise-OR of one or more flags. One and only one of the flag values `NSP_Forw`, `NSP_Inv`, and `NSP_Init` must be specified. The `NSP_NoScale` flag is optional. The values for the *flags* argument are described in Flags Argument of the “Basic FFT Functions” section earlier in this chapter. |
| *inSamps* | Pointer to the complex array which holds the input to the `nsp?CcsFftNip()` function. The *inSamps[n]* array must be in `RCCcs` format and be of length *N*/2+1 complex samples. |
| *order* | The base-2 logarithm of the number of samples in the FFT (*N*). |

| | |
|---|---|
| `outSamps` | Pointer to the real array which holds the output from the `nsp?CCsFftNip()` function. The `outSamps[n]` array must be of length $N = 2^{order}$. |
| `samps` | Pointer to the array which holds the input and output samples for the `nsp?CCsFft()` function. The `samps[n]` array must be of length $N + 2$ elements (`floats` or `doubles`). On input, `samps[n]` should be considered a complex array of length $N/2 + 1$ complex samples in `RCCcs` format. On output, `samps[n]` should be considered as a real array, the first $N$ elements of which are data and the last two elements are ignored. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

**`nsp?CcsFft()`.** The function `nsp?CcsFft()` computes the FFT in-place. In the forward direction (`flags` = `NSP_Forw`), `samps[n]` contains $N/2 + 1$ complex samples in `RCCcs` format that describe a conjugate-symmetric time-domain signal $x(n)$. On exit, `samps[n]` contains $N$ real frequency-domain samples that are the forward FFT of $x(n)$.

In the inverse direction (`flags` = `NSP_Inv`), `samps[n]` contains $N/2 + 1$ complex samples in `RCCcs` format that describe a conjugate-symmetric frequency-domain signal $X(k)$. On exit, `samps[n]` contains $N$ real time-domain samples that are the inverse FFT of $X(k)$.

- The array `samps[n]` must be of length $N/2 + 1$ complex elements so that it can contain the $N + 2$ real numbers that are returned. The two extra elements (at the end of the array) are ignored on output.
- Upon return, the array `samps[n]` should be treated as an array of $N + 2$ real numbers rather than an array of complex numbers. This can be done by appropriate casting. The real elements $x(0)$ to $x(N)$ span normalized time or, in the case of an inverse FFT, normalized frequency from 0.0 to 0.5.

**`nsp?CcsFftNip()`.** The function `nsp?CcsFftNip()` computes the FFT not-in-place. In the forward direction (`flags` = `NSP_Forw`), the input array `inSamps[n]` contains $N/2 + 1$ complex samples in `RCCcs` format. The samples describe a conjugate-symmetric time-domain signal `x(n)`. On exit, the output array `outSamps[n]` contains `N` real frequency-domain samples that are the forward FFT of `x(n)`.

In the inverse direction (`flags` = `NSP_Inv`), the input array `inSamps[n]` contains `N`/2 + 1 complex samples in `RCCcs` format. The samples describe a conjugate-symmetric frequency-domain signal `X(k)`. On exit, the output array `outSamps[n]` contains `N` real time-domain samples that are the inverse FFT of `X(k)`.

## Related Topics

| | |
|---|---|
| `RealFft` | Provides the inverse to the `nsp?CcsFft()` function (see page 7-38). |
| `RealFftNip` | Provides the inverse to the `nsp?CcsFftNip()` function (see page 7-38). |

# FFTs of Two Real Signals

The `nsp?Real2Fft()` and `nsp?Real2FftNip()` functions described in this section compute the forward or inverse FFT of two real signals (either time- or frequency-domain). See “Fft” in page 7-16 for a description of `nsp?Fft()` and Appendix A for general information on the FFT.

The forward or inverse FFT of a real signal is conjugate-symmetric. For example, the forward FFT of a real time-domain signal is conjugate-symmetric in frequency. This property allows two real FFTs to be simultaneously computed using a single complex FFT. The algorithms used to implement these functions are very different from the ones used for `nsp?RealFft()` even though the functions are quite similar.

## Inverses of FFTs of Two Real Signals

The `nsp?Real2Fft()` and `nsp?Real2FftNip()` functions do not provide their own inverses. Instead, the inverses are provided by the `nsp?Ccs2Fft()` and `nsp?Ccs2FftNip()` functions.

# Real2Fft, Real2FftNip

*Computes the forward or inverse FFT of two real signals.*

```
void nspsReal2Fft(float *xSamps, float *ySamps, int order,
      int flags);
void nspsReal2FftNip(const float *xInSamps, SCplx *xOutSamps,
      const float *yInSamps, SCplx *yOutSamps, int order,
      int flags);
      /* real values, single precision */
void nspdReal2Fft(double *xSamps, double *ySamps, int order,
      int flags);
```

```
void nspdReal2FftNip(const double *xInSamps, DCplx *xOutSamps,
      const double *yInSamps, DCplx *yOutSamps, int order,
      int flags);
      /* real values, double precision */
void nspwReal2Fft(short *xSamps, short *ySamps, int order,
      int flags, int ScaleMode, int *ScaleFactor);
void nspwReal2FftNip(const short *xInSamps, WCplx *xOutSamps,
      const short *yInSamps, WCplx *yOutSamps, int order,
      int flags, int ScaleMode, int *ScaleFactor);
      /* real values, short integer */
```

| | |
|---|---|
| *flags* | Indicates the direction of the fast Fourier transform and whether bit-reversal is performed. The argument consists of the bitwise-OR of one or more flags. One and only one of the flag values `NSP_Forw`, `NSP_Inv`, and `NSP_Init` must be specified. The `NSP_NoScale` flag is optional. The values for the *flags* argument are described in Flags Argument of the “Basic FFT Functions” sections. |
| *order* | The base-2 logarithm of the number of samples in the FFT ($N$). |
| *xInSamps* | Pointer to the array which holds the real samples to be input to the `nsp?Real2FftNip()` function. The array must be of length $N = 2^{order}$. |
| *xOutSamps* | Pointer to the array which holds the complex samples output from the `nsp?Real2FftNip()` function. The array is in `RCCcs` format and must be of length $N/2 + 1$ complex samples. |
| *xSamps* | Pointer to the array which holds the input and output of the `nsp?Real2Fft()` function. The *xSamps[n]* array must be of length $N + 2$ elements (`floats` or `doubles`). On input, the array should be considered as a real array, the first $N$ elements of which are data and the last two elements of which are ignored. On output, the array should be considered a complex array of length $N/2 + 1$ complex samples. |

| | |
|---|---|
| `yInSamps` | Pointer to the array which holds the real samples to be input to the `nsp?Real2FftNip()` function. The array must be of length $N = 2^{order}$. |
| `yOutSamps` | Pointer to the array which holds the complex samples output from the `nsp?Real2FftNip()` function. The array is in `RCCcs` format and must be of length $N/2 + 1$ complex samples. |
| `ySamps` | Pointer to the array which holds the input and output of the `nsp?Real2Fft()` function. The `ySamps[n]` array must be of length $N + 2$ elements (`floats` or `doubles`). On input, the array should be considered as a real array, the first $N$ elements of which are data and the last two elements of which are ignored. On output, the array should be considered a complex array of length $N/2+ 1$ complex samples. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

**`nsp?Real2Fft()`.** The function `nsp?Real2Fft()` computes the FFT in-place. It computes the FFT of the $N$ real samples stored in `xSamps[n]`, and returns $N/2+1$ complex samples to `xSamps[n]`. Similarly, the FFT of the samples in `ySamps[n]` are returned to `ySamps[n]`.

**`nsp?Real2FftNip()`.** The function `nsp?Real2FftNip()` computes the FFT not-in-place. It computes the FFT of the $N$ real samples in `xInSamps[n]`, storing $N/2 + 1$ complex samples in `xOutSamps[n]`. Similarly, the FFT of the $N$ samples in `yInSamps[n]` are stored into `yOutSamps[n]`.

Example 7-16 shows how to use the `nsp?Real2FftNip()` function to convolve two real signals.

#### Example 7-16 Using nsp?Real2FftNip() to Convolve Two Real Signals

```
/* perform fast convolution
 * of real signals
 */
double    xTime[256], hTime[256], yTime[256];
DCplx     xFreq[129], hFreq[129], yFreq[129];
/* insert code here to fill xTime and hTime vectors */
nspdReal2FftNip(xTime, xFreq, hTime, hFreq, 8, NSP_Forw);
nspzbMpy3(xFreq, hFreq, yFreq, 129);
nspdCcsFftNip(yFreq, yTime, 8, NSP_Inv);
/* y now contains the (circular) convolution of h and x */
```

### Related Topics

`Ccs2Fft` Provides the inverse to the `nsp?Real2Fft()` function (see page 7-52).

`RealFft` Computes the FFT of a single, real signal (see page 7-38).

`RealFftl` Provides a lower-level interface to the FFT algorithm (see page 7-23).

See [Mit93], section 8-2-9, *Real-Valued FFTs*, for more information on real-valued fast Fourier transforms.

## FFTs of Two Conjugate-Symmetric Signals

The `nsp?Ccs2Fft()` and `nsp?Ccs2FftNip()` functions described in this section compute the forward or inverse FFT of two independent conjugate-symmetric signals (either time- or frequency-domain), yielding two real signals. See “Fft” in page 7-16 for a description of `nsp?Fft()` and Appendix A for general information on the FFT. The algorithms used to implement these functions are very different from the ones used for `nsp?CcsFft()` even though the functions are quite similar.

## Inverses of FFTs of Two Conjugate-Symmetric Signals

The `nspsCcs2Fft()` and `nspsCcs2Fft()` functions do not provide their own inverses. Instead, the inverses are provided by the `nsp?Real2Fft()` and `nsp?Real2FftNip()` functions.

# Ccs2Fft, Ccs2FftNip

*Computes the forward or inverse FFT of two complex conjugate-symmetric (CCS) signals.*

```
void nspsCcs2Fft(float *xSamps, float *ySamps, int order,
      int flags);
void nspsCcs2FftNip(const SCplx *xInSamps, float *xOutSamps,
      const SCplx *yInSamps, float *yOutSamps, int order,
      int flags);
      /* real values; single precision */
void nspdCcs2Fft(double *xSamps, double *ySamps, int order,
      int flags);
void nspdCcs2FftNip(const DCplx *xInSamps, double *xOutSamps,
      const DCplx *yInSamps, double *yOutSamps, int order,
      int flags);
      /* real values; double precision */
void nspwCcs2Fft(short *xSamps, short *ySamps, int order,
      int flags, int scaleMode, int *scaleFactor);
void nspwCcs2FftNip(const WCplx *xInSamps, short *xOutSamps,
      const WCplx *yInSamps, short *yOutSamps, int order,
      int flags, int scaleMode, int *scaleFactor);
      /* real values; short integer */
```

*flags* — Indicates the direction of the fast Fourier transform and whether bit-reversal is performed. The argument consists of the bitwise-OR of one or more flags. One and only one of the flag values `NSP_Forw`, `NSP_Inv`, and `NSP_Init` must be specified. The `NSP_NoScale` flag is optional. The values for the `flags` argument are described in Flags Argument of the "Basic FFT Functions" section.

*`order`* The base-2 logarithm of the number of samples in the FFT (*`N`*).

*`xInSamps`* Pointer to the array which holds the complex conjugate-symmetric samples in `RCCcs` format for input to the `nsp?Ccs2FftNip()` function. The *`xInSamps[n]`* array must be of length *`N`*/2 + 1 complex samples.

*`xOutSamps`* Pointer to the array which holds the real samples output from the `nsp?Ccs2FftNip()` function. The *`xOutSamps[n]`* array must be of length $N = 2^{order}$.

*`xSamps`* Pointer to the array which holds the input and output of the `nsp?Ccs2Fft()` function. On input, *`xSamps[n]`* should be considered as a complex array of length *`N`*/2 + 1 complex samples in `RCCcs` format. On output, *`xSamps[n]`* should be considered as a real array, the first *`N`* elements of which are data, and the last two elements are ignored.

*`yInSamps`* Pointer to the array which holds the complex conjugate-symmetric samples in `RCCcs` format for input to the `nsp?Ccs2FftNip()` function. The *`yInSamps[n]`* array must be of length *`N`*/2 + 1 complex samples.

*`yOutSamps`* Pointer to the array which holds the real samples output from the `nsp?Ccs2FftNip()` function. The *`yOutSamps[n]`* array must be of length $N = 2^{order}$.

*`ySamps`* Pointer to the array which holds the input and output of the `nsp?Ccs2Fft()` function. On input, *`ySamps[n]`* should be considered as a complex array of length *`N`*/2 + 1 complex samples in `RCCcs` format. On output, *`ySamps[n]`* should be considered as a real array, the first *`N`* elements of which are data, and the last two elements are ignored.

| | |
|---|---|
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

### Discussion

**nsp?Ccs2Fft().** The function `nsp?Ccs2Fft()` computes the FFT in-place. It computes the FFT of the *N*/2 + 1 complex samples stored in *xSamps[n]*, and returns *N* real samples back into *xSamps[n]*. Similarly, the FFT of the samples in *ySamps[n]* are returned to *ySamps[n]*.

**nsp?Ccs2FftNip().** The function `nsp?Ccs2FftNip()` computes the FFT not-in-place. It computes the FFT of the *N*/2 + 1 complex conjugate-symmetric samples in *xInSamps[n]*, and returns *N* real samples to *xOutSamps[n]*. Similarly, the FFT of the samples in *yInSamps[n]* are returned to *yOutSamps[n]*.

### Related Topics

| | |
|---|---|
| `CcsFft` | Calculates the FFT of a single, conjugate-symmetric signal (see page 7-45). |
| `Real2Fft` | Provides the inverse to the function `nsp?Ccs2Fft()` (see page 7-48). |

See [Mit93], section 8-2-9, *Real-Valued FFTs*, for more information on the FFTs of real-valued signals.

## Memory Reclaim Functions

This section describes the `nspFreeBitrevTbls()` and `nsp?FreeTwdTbls()`functions that free the memory allocated for bit-reversed indices tables and for twiddle tables, respectively. These tables are used by the Signal Processing library internally.

You need to use the functions decribed in this section only if you are particularly concerned about clearing the memory. Otherwise, the memory is always reclaimed at the program exit.

## FreeBitRevTbls

*Frees dynamic memory for tables of bit-reversed indices.*

```
void nspFreeBitRevTbls();
```

### Discussion

The `nspFreeBitRevTbls()` function frees all dynamic memory for all bit-reversal tables of any size previously allocated.

## FreeTwdTbls

*Frees memory associated with all twiddle tables of a particular type.*

```
void nspcFreeTwdTbls();
      /* complex values; single precision */
void nspzFreeTwdTbls();
      /* complex values; double precision */
void nspvFreeTwdTbls();
      /* complex values; short integer */
```

### Discussion

The `nsp?FreeTwdTbls()` function frees all memory associated with all twiddle tables of a particular data type. Thus the function `nspcFreeTwdTbls()` frees all memory associated with all single-precision FFT and DFT twiddle tables. Similarly, the function `nspzFreeTwdTbls()` frees all memory associated with all double-precision FFT and DFT twiddle tables; and the function `nspvFreeTwdTbls()` frees all memory with all short integer FFT and

DFT twiddle tables. The function leaves the internal pointer tables properly initialized so that subsequent memory allocation by internal functions will succeed.

Use the `nsp?FreeTwdTbls()` function at the end of a program to release all of the dynamic memory allocated previously.

## DCT Function

This section describes the `nsp?Dct()` function that computes the discrete cosine transform of a signal.

## Dct

*Computes the forward or inverse discrete cosine transform (DCT) of a signal.*

```
void nspsDct(const float *src, float *dst, int len, int flags);
      /* real values; single precision */
void nspdDct(const double *src, double *dst, int len, int flags);
      /* real values; double precision */
void nspwDct(const short *src, short *dst, int len, int flags);
      /*complex values; short integer */
```

| | |
|---|---|
| *src* | Pointer to the input data array. |
| *dst* | Pointer to the output data array. |
| *len* | The number of elements in the *src* (and *dct*) array. |
| *flags* | Specifies how the DCT should be performed; can be one of the following:<br>`NSP_DCT_Forward` for the forward DCT,<br>`NSP_DCT_Inverse` for the inverse DCT, or<br>`NSP_DCT_Free` for deallocating the memory used for an internal table of transform coefficients. |

## Discussion

The `nsp?Dct()` function computes the forward and inverse discrete cosine transform (DCT). If *len* is a power of 2, the function uses an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *len*, this function uses the direct formulas given below; however, the symmetry of cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT, $N$ = *len*,

$C(k) = \frac{1}{\sqrt{N}}$ for $k = 0$, $C(k) = \frac{\sqrt{2}}{\sqrt{N}}$ for $k > 0$;

$x(n)$ is *src[n]* and $y(k)$ is *dst[k]* for the forward DCT;

$x(n)$ is *dst[n]* and $y(k)$ is *src[k]* for the inverse DCT.

The forward DCT is defined by the formula

$$y(k) = C(k)\sum_{n=0}^{N-1} x(n) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

The definition of the inverse discrete cosine transform is:

$$x(n) = \sum_{k=0}^{N-1} C(k)y(k) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

The argument *flags* has no default value; you have to specify one of the values `NSP_DCT_Forward`, `NSP_DCT_Inverse`, or `NSP_DCT_Free`. If you specify `NSP_DCT_Free`, the function ignores all other parameters and frees the memory used for the internal table of transform coefficients

Example 7-17 illustrates the use of the `nsp?Dct()` function.

#### Example 7-17 Using nsp?Dct() to Compress and Reconstruct a Signal

```
/* Example of using DCT in the data compression 4:1 */
float x[len], y[len];
int n;
/* data: Gaussian function, magn = 1 and sigma = N/3 */
for(n=0; n<len; n++) x[n] =
(float)(exp(-0.5*(len/2-n)*(len/2-n)/(len/3.0)));
      /* get cosine transform coefficients */
nspsDct(x, y, len, NSP_DCT_Forward);
      /* Set 3/4 of these coefficients to zero */
for(n=len/4; n<len; n++) y[n] = 0.0f;
      /* restore signal using len/4 values */
nspsDct(y, x, len, NSP_DCT_Inverse);
      /* Free the factors table memory*/
nspsDct(NULL, NULL, 0, NSP_DCT_Free);
```

**Application Notes:** On Intel® processors with MMX™ technology, the `nspwDct()` function computes the DCT of short integer data about twice as fast as `nspsDct()` computes the DCT of real data.

However, the accuracy of the short integer computation might be insufficient if the data array is long. For example, the mean square error for the integer computation can be on the order of 0.0001 for $n = 8$, and 0.01 for $n = 32$.

### Related Topics

You can find more details about the DCT and its implementation in [Fei92], [NIC91], and [Rao90] (see the Bibliography section).

# Filtering Functions

# 8

Library function lists

The functions described in this chapter implement the following types of filters:

- finite impulse response (FIR)
- adaptive finite impulse response using least mean squares (LMS)
- infinite impulse response (IIR)
- median

To understand the background of the filters used by the Signal Processing Library, see Appendix B.

Depending on the application, there are two different filtering modes:

batch — The signal to be filtered is finite and stored entirely in memory. Such a signal can be filtered in “batch” mode, that is, all at once in a single (large) operation. The signal’s samples are convolved with a set of filter coefficients to produce an output signal. In this case, non-causal filtering is possible since the entire signal is available.

cyclic — The signal to be filtered is not stored entirely in memory, either because it is too large, infinite in length, or the output is required before input is entirely known. Such a signal can be filtered in “cyclic” mode, that is, in small pieces. In this case, a portion of the signal is read into memory, filtered, the output is written out, and then the process is repeated with the next portion.

Cyclic filtering, in contrast to batch filtering, requires information (that is, “state”) to be preserved between each cycle. Managing this state can be complicated. Thus the library functions for cyclic processing are divided into two groups:

| | |
|---|---|
| low-level | These functions give the application direct access to all state information and allow the state to be shared among different filters. |
| normal | These functions group all state information into a single pointer using dynamic memory allocation, providing a simpler interface. |

The normal functions perform all of their own memory allocation while the low-level functions do not perform any memory allocation. Typically, you will only use the low-level functions if you need to closely manage or make special arrangements for the way your application allocates memory.

## Low-Level FIR Filter Functions

The functions described in this section initialize a low-level finite impulse response (FIR) filter, get and set the filter coefficients and delay line, and perform the filtering function. The low-level FIR functions are intended for cyclic processing: first, initialize the filter, then filter the samples one at a time or in blocks. This allows a relatively expensive initialization function to pre-compute values so that later filtering is efficient (this is particularly useful for multi-rate filtering). For non-cyclic (batch) FIR filtering, see “Conv” in Chapter 9 for a description of the convolution function `nsp?Conv()`.

The low-level FIR functions maintain the filter coefficients separately from the delay line, allowing multiple delay lines to be used with the same set of taps. Also, the low-level FIR functions do not use any dynamic memory allocation.

To use a low-level FIR filter, follow this general scheme:

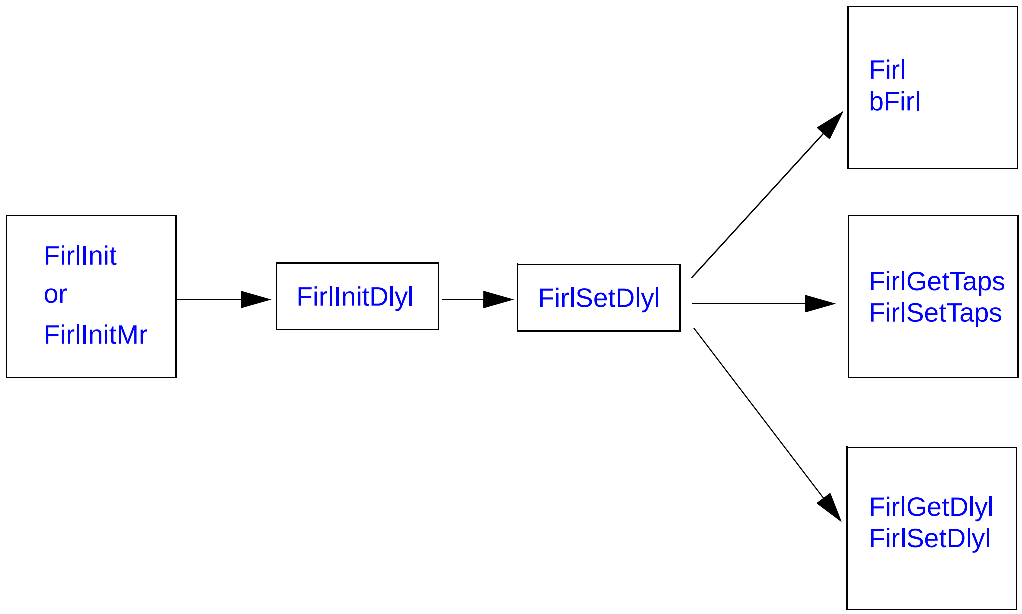
1. Call either `nsp?FirlInit()` to initialize the coefficients and structure of a single-rate filter or call `nsp?FirlInitMr()` to initialize the coefficients and structure of a multi-rate filter.
2. Call `nsp?FirlInitDlyl()` to initialize the structure of a delay line.

   The delay line is associated with a particular set of taps. Multiple delay lines for a given set of taps can be initialized by calling this function multiple times, but there should be only one call for each delay line.
3. Call `nsp?FirlSetDlyl()` to initialize the delay line itself.
4. After this initialization, you have a choice of functions to call, depending on what you want to accomplish.
   a. Call the `nsp?Firl()` function to filter a single sample through a single-rate filter and/or call `nsp?bFirl()` to filter a block of consecutive samples through a single-rate or multi-rate filter.
   b. Call the `nsp?FirlGetTaps()` function and then the `nsp?FirlSetTaps()` function to get and set the filter coefficients (taps).
   c. Call the `nsp?FirlGetDlyl()` function and then the `nsp?FirlSetDlyl()` function to get and set the values in the delay line.

   Real and complex taps can be mixed with real and complex delay lines (that is, all four combinations are allowable). However, taps and delay lines of different precision must not be mixed. It is the application's responsibility to call the correct function for the given type combination. This is not checked at compile time nor is it required to be checked at run-time.

Figure 8-1 illustrates the order of use of the low-level FIR filter functions.

**Figure 8-1 Order of Use of the Low-Level FIR Functions**




## FirlInit, FirlInitMr, FirlInitDlyl

*Initializes a low-level FIR filter.*

```
void nspsFirlInit(float *taps, int tapsLen,
      NSPFirTapState *tapStPtr);
void nspsFirlInitMr(float *taps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase,
      NSPFirTapState *tapStPtr);
```

```
void nspsFirlInitDlyl(const NSPFirTapState *tapStPtr, float *dlyl,
      NSPFirDlyState *dlyStPtr);
      /* real values; single precision */

void nspcFirlInit(SCplx *taps, int tapsLen,
      NSPFirTapState *tapStPtr);
void nspcFirlInitMr(SCplx *taps, int tapsLen,int upFactor,
      int upPhase, int downFactor, int downPhase,
      NSPFirTapState *tapStPtr);
void nspcFirlInitDlyl(const NSPFirTapState *tapStPtr, SCplx *dlyl,
      NSPFirDlyState *dlyStPtr);
      /* complex values; single precision */

void nspdFirlInit(double *taps, int tapsLen,
      NSPFirTapState *tapStPtr);
void nspdFirlInitMr(double *taps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase,
      NSPFirTapState *tapStPtr);
void nspdFirlInitDlyl(const NSPFirTapState *tapStPtr, double *dlyl,
      NSPFirDlyState *dlyStPtr);
      /* real values; double precision */

void nspzFirlInit(DCplx *taps, int tapsLen,
      NSPFirTapState *tapStPtr);
void nspzFirlInitMr(DCplx *taps, int tapsLen,
      int upFactor, int upPhase, int downFactor, int downPhase,
      NSPFirTapState *tapStPtr);
void nspzFirlInitDlyl(const NSPFirTapState *tapStPtr, DCplx *dlyl,
      NSPFirDlyState *dlyStPtr);
      /* complex values; double precision */

void nspwFirlInit(float *taps, int tapsLen,
      NSPFirTapState *tapStPtr);
void nspwFirlInitMr(float *taps, int tapsLen, int upFactor,
      int upPhase, int downFactor, int downPhase,
      NSPFirTapState *tapStPtr);
void nspwFirlInitDlyl(const NSPFirTapState *tapStPtr, short *dlyl,
      NSPFirDlyState *dlyStPtr);
      /* real values; short integer */
```

| | |
|---|---|
| *dlyl* | Pointer to the array which specifies the initial values for the delay line for the `nsp?FirlInitDlyl()` function. |
| *dlylStPtr* | Pointer to the `NSPFirDlylState` structure. |
| *downFactor* | The factor value used by the `FirlInitMr()` function for down-sampling multi-rate signals. |
| *downPhase* | The phase value used by the `FirlInitMr()` function for down-sampling multi-rate signals. |
| *taps* | Pointer to the array which specifies the filter coefficients for the `nsp?FirlInit()` and `nsp?FirlInitMr()` functions. |
| *tapsLen* | The number of taps in the *taps[n]* array. |
| *tapStPtr* | Pointer to the `NSPFirTapState` structure. |
| *upFactor* | The factor value used by the `FirlInitMr()` function for up-sampling multi-rate signals. |
| *upPhase* | The phase value used by the `FirlInitMr()` function for up-sampling multi-rate signals. |

## Discussion

**`nsp?FirlInit()`.** The `nsp?FirlInit()` function configures a single-rate filter. The array *taps[n]* specifies the filter coefficients (taps) *h(n)*. The `nsp?FirlInit()` function initializes the structure pointed to by *tapStPtr*. The structure `NSPFirTapState` defines the length of the FIR filter, *tapsLen*, and a pointer to the *taps[n]* array. In addition, the contents of the *taps[n]* array can be permuted in an implementation-dependent way to allow faster filtering. The pointer *tapStPtr* is used in subsequent calls to reference the taps and filter structure.

**`nsp?FirlInitMr()`.** The `nsp?FirlInitMr()` function configures a multi-rate filter; that is, a filter that internally up-samples and/or down-samples using a polyphase filter structure. It initializes *tapStPtr* in the same way as described for single-rate filters, but includes additional information about the required up-sampling and down-sampling parameters. The arguments *upFactor* and *upPhase* are the same as described for the `nsp?UpSample()` function, and the arguments `downFactor` and `downPhase` are the same as described for the `nsp?DownSample()` function. For more information on multi-rate filters, see Appendix C.

**Application Notes:** If your application is running on a Pentium® processor with MMX™ technology or Pentium Pro processor, the call of `nspwFirlInit()` or `nspwFirlInitMr()` causes both the permutation and conversion (from floating-point to short format) of the user-defined filter taps. This is due to a call of `nspwFirlSetTaps()` inside initialization functions. For more information see FirlGetTaps, FirlSetTaps on page 8-14.

**`nsp?FirlInitDlyl()`.** The `nsp?FirlInitDlyl()` function associates a delay line with a particular set of taps. During initialization, you must specify the delay line array `dlyl[n]`. This array provides the initial values of the delay line, and is updated during each filtering operation. The delay line can be permuted in an implementation-dependent way to allow faster filtering. The pointer `dlyStPtr` is used in subsequent calls to reference the delay line. For single-rate filters, `dlyl[n]` must be `tapsLen` long, though only the first `tapsLen` - 1 samples provide initial values. For multi-rate filters, define the length of the delay line array `dlyl` as `PL`, where

$$PL = \lceil tapsLen / upFactor \rceil.$$

As discussed in Appendix C, the length of the delay line for a multi-rate filter does not reduce to the length of a single-rate filter.

The array `taps[n]` and the array `dlyl[n]` are used every time the filter functions are called. Thus they must exist while the filter exists (that is, they must not be stored in a stack variable that goes out of scope prior to the last `nsp?Firl()` invocation). Further, since the arrays might be permuted, they must not be referenced by the application except as described for the functions `nsp?FirlSetTaps()` and `nsp?FirlSetDlyl()`. It is helpful to view the array `taps[n]` as part of `tapStPtr` and to view the array `dlyl[n]` as part of `dlyStPtr`.

**Application Notes:** The contents of the `NSPFirTapState` and `NSPFirDlyState` structures are implementation-dependent.

The structures `NSPFirTapState` and `NSPFirDlyState` are data type-independent.

The `nsp?FirlInitDlyl()` function can accept tap arrays and delay lines of different types but not of different precisions. For example, `float` with `SCplx`, or `double` with `DCplx` are permissible but `double` with `float` is not.

The coefficient and delay line values in *`taps[n]`* and *`dlyl[n]`* can be stored in an implementation-dependent order to permit efficient computation of single- and multi-rate filtering.

## Related Topics

| | |
|---|---|
| `bFirl` | Filters a block of samples through a low-level FIR filter (see page 8-9). |
| `DownSample` | Down-samples a signal, conceptually decreasing its sampling rate by an integer factor (see page 3-58). |
| `Firl` | Filters a single sample through a low-level FIR filter (see page 8-9). |
| `FirlGetDlyl` | Gets the delay line contents for a low-level FIR filter (see page 8-18). |
| `FirlGetTaps` | Gets the tap coefficients for a low-level FIR filter (see page 8-14). |
| `FirlSetDlyl` | Sets the delay line contents for a low-level FIR filter (see page 8-18). |
| `FirlSetTaps` | Sets the tap coefficients for a low-level FIR filter (see page 8-14). |
| `UpSample` | Up-samples a signal, conceptually increasing its sampling rate by an integer factor (see page 3-55). |

## FirI, bFirI

*Low-level functions which filter either a single sample or block of samples through an FIR filter.*

```
float nspsFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, float samp);
void nspsbFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr,  const float *inSamps,
      float *outSamps, int numIters);
      /* real input, real taps; single precision */

SCplx nspcFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, SCplx samp);
void nspcbFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr,  const SCplx *inSamps,
      SCplx *outSamps, int numIters);
      /* complex input, complex taps; single precision */

SCplx nspscFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, float samp);
void nspscbFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr,  const float *inSamps,
      SCplx *outSamps, int numIters);
      /* real input, complex taps; single precision */

SCplx nspcsFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, SCplx samp);
void nspcsbFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr,  const SCplx *inSamps,
      SCplx *outSamps, int numIters);
      /* complex input, real taps; single precision */

double nspdFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, double samp);
```

```
void nspdbFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr,  const double *inSamps,
      double *outSamps, int numIters);
      /* real input, real taps; double precision */

DCplx nspzFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, DCplx samp);
void nspzbFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr,  const DCplx *inSamps,
      DCplx *outSamps, int numIters);
      /* complex input, complex taps; double precision */

DCplx nspdzFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, double samp);
void nspdzbFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr,  const double *inSamps,
      DCplx *outSamps, int numIters);
      /* real input, complex taps; double precision */

DCplx nspzdFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, DCplx samp);
void nspzdbFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr,  const DCplx *inSamps,
      DCplx *outSamps, int numIters);
      /* complex input, real taps; double precision */

float nspwFirl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlyStPtr, float samp, int ScaleMode,
      int *ScaleFactor);
void nspwbFirl(const NSPFirTapState *tapStPtr,
NSPFirDlyState *dlyStPtr, const short *inSamps, short *outSamps,
int numIters, int ScaleMode, int *ScaleFactor);
      /* real input, real taps; short integer */
```

| | |
|---|---|
| *dlylStPtr* | Pointer to the `NSPFirDlylState` structure. |
| *inSamps* | Pointer to the array which stores the input samples to be filtered by the `nsp?bFirl()` function. |
| *numIters* | The number of samples (single-rate) or blocks (multi-rate) to be filtered by the `nsp?bFirl()` function. |

| | |
|---|---|
| *outSamps* | Pointer to the array which stores the output samples filtered by the `nsp?bFirl()` function. |
| *samp* | Pointer to the current sample for the `nsp?Firl()` function. |
| *tapStPtr* | Pointer to the `NSPFirTapState` structure. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

The `nsp?Firl()` and `nsp?bFirl()` functions filter either a single sample or block of samples through a low-level finite impulse response (FIR) filter. Many combinations of input (*x(n)*) types and filter coefficients (taps) types are possible. Real or complex input can be mixed with real or complex filter coefficients. This is indicated by the `s`, `c`, `sc`, `cs`, `d`, `z`, `dz`, `zd`, and `w` type codes following the `nsp` prefix in the function names above. For both of the functions, `nsp?Firl()` and `nsp?bFirl()`, the allowed combinations of real and complex input and filter coefficients are described in Table 8-1.

**Table 8-1 Input and Taps Combinations for nsp?Firl() and nsp?bFirl() Functions**

| Type Codes | x(n) (or input) Type | Filter Coefficient (or taps) Type | y(n) (or output) Type |
|---|---|---|---|
| s | float | float | float |
| c | SCplx | SCplx | SCplx |
| sc | float | SCplx | SCplx |
| cs | SCplx | float | SCplx |
| d | double | double | double |
| z | DCplx | DCplx | DCplx |
| dz | double | DCplx | DCplx |
| zd | DCplx | double | DCplx |
| w | short | float/short | short |

Even though real or complex input can be mixed with real or complex filter coefficients, input and filter coefficients of different precision cannot be mixed.

**Previous Tasks:** Before using either `nsp?FirlI()` or `nsp?bFirl()`, you must initialize the filter taps state *`tapStPtr`*, a taps array *`taps[n]`*, the number of taps *`tapsLen`*, and any multi-rate parameters by calling either `nsp?FirlInit()` or `nsp?FirlInitMr()`. The taps values are denoted $h(0)...h(tapsLen - 1)$.

You must also initialize the delay line state *`dlyStPtr`* and a delay line array *`dlyl[n]`* by calling `nsp?FirlInitDlyl()`, and then update the delay line state by calling `nsp?FirlSetDlyl()`. For single-rate filters, the contents of the delay line array are denoted as

$x(n - tapsLen + 1)...x(n - 1)$.

For multi-rate filters, the contents of the delay line array are denoted as

$x(n - PL)...x(n - 1)$.

**`nsp?Firl()`.** The `nsp?Firl()` function filters a single sample through a single-rate filter. The argument *`samp[n]`* is the sample to be filtered and is denoted as $x(n)$. The return value is $y(n)$, and is calculated as follows:

$$y(n) = \sum_{k=0}^{tapsLen-1} h(k) \cdot x(n-k)$$

The delay line *`dlyl`* (and the delay line state pointer *`dlyStPtr`*, if appropriate) is updated to contain $x(n)$, and $x(n - tapsLen + 1)$ is discarded from the delay line.

**`nsp?bFirl()`.** The `nsp?bFirl()` function filters a block of consecutive samples through a single-rate or multi-rate filter. For single-rate filters, the *`numIters`* samples in the array *`inSamps[n]`* are filtered, and the resulting *`numIters`* samples are stored in the array *`outSamps[n]`*. The results are identical to *`numIters`* consecutive calls to `nsp?Firl()`. The values in the *`outSamps[n]`* array are calculated as follows:

$inSamps[m] = x(n+m),\ 0 \le m < tapsLen$

$$y(n+m) = outSamps[m] = \sum_{k=0}^{tapsLen-1} h(k) \cdot x(n+m-k)$$

For multi-rate filters, the (*numIters* * *downFactor*) samples in the array *inSamps[n]* are filtered, and the resulting (*numIters* * *upFactor*) samples are stored in the array *outSamps[n]*. For both single-rate and multi-rate filters, the appropriate number of samples from *inSamps[n]* are copied into the delay line, and the oldest samples are discarded. See Appendix C for more information on multi-rate filtering.

Example 8-1 illustrates single-rate filtering with the `nsp?Firl()` function.

### Example 8-1 Single-Rate Filtering with the nsp?Firl() Function

```
/* standard
 * single-rate filtering
 */
NSPFirTapState tapSt;
NSPFirDlyState dlySt;
double      taps[32];
double      dlyl[32];
int            i;
double      xval, yval;
/* insert code here to initialize taps */

nspdFirlInit(taps, 32, &tapSt);
nspdFirlInitDlyl(&tapSt, dlyl, &dlySt);
/* zero out the delay line */
nspdFirlSetDlyl(&tapSt, (double *)NULL, &dlySt);
for (i=0; i < 2000; i++) {
     xval = /* insert code here get next val of x(n) */;
     yval = nspdFirl(&tapSt, &dlySt, xval);
     /* yval has the output sample */
}
```

## Related Topics

| | |
|---|---|
| `DownSample` | Down-samples a signal, conceptually decreasing its sampling rate by an integer factor (see page 3-58). |
| `FirlGetDlyl` | Gets the delay line contents for a low-level FIR filter (see page 8-18). |

| | |
|---|---|
| `FirlGetTaps` | Gets the tap coefficients for a low-level FIR filter (see page 8-14). |
| `FirlInit` | Initializes a single-rate, low-level FIR filter (see page 8-4). |
| `FirlInitDlyl` | Initializes the delay line for a low-level FIR filter (see page 8-4). |
| `FirlInitMr` | Initializes a multi-rate, low-level FIR filter (see page 8-4). |
| `FirlSetDlyl` | Sets the delay line contents for a low-level FIR filter (see page 8-18). |
| `FirlSetTaps` | Sets the tap coefficients for a low-level FIR filter (see page 8-14). |
| `UpSample` | Up-samples a signal, conceptually increasing its sampling rate by an integer factor (see page 3-55). |

## FirlGetTaps, FirlSetTaps

*Gets and sets the tap coefficients of low-level FIR filters.*

```
void nspsFirlGetTaps(NSPFirTapState *tapStPtr, float *outTaps);
void nspsFirlSetTaps(float *inTaps, NSPFirTapState *tapStPtr);
      /* real values; single precision */

void nspcFirlGetTaps(NSPFirTapState *tapStPtr, SCplx *outTaps);
void nspcFirlSetTaps(SCplx *inTaps, NSPFirTapState *tapStPtr);
      /* complex values; single precision */

void nspdFirlGetTaps(NSPFirTapState *tapStPtr, double *outTaps);
void nspdFirlSetTaps(double *inTaps, NSPFirTapState *tapStPtr);
      /* real values; double precision */

void nspzFirlGetTaps(NSPFirTapState *tapStPtr, DCplx *outTaps);
```

```
void nspzFirlSetTaps(DCplx *inTaps, NSPFirTapState *tapStPtr);
      /* complex values; double precision */

void nspwFirlGetTaps(NSPFirTapState *tapStPtr, float *outTaps);
void nspwFirlSetTaps(float *inTaps, NSPFirTapState *tapStPtr);
      /* real values; short integer */
```

| | |
|---|---|
| *inTaps* | Pointer to the array holding copies of the tap coefficients. |
| *outTaps* | Pointer to the array holding copies of the tap coefficients. |
| *tapStPtr* | Pointer to the `NSPFirTapState` structure. |

## Discussion

The `nsp?FirlGetTaps()` and `nsp?FirlSetTaps()` functions provide a safe mechanism to get and set the taps of a low-level FIR filter. Because the taps might be stored in permuted order, it is not safe for the application to directly access the tap array. Instead, `nsp?FirlGetTaps()` and `nsp?FirlSetTaps()` should be used.

**Previous Tasks:** Before calling either `nsp?FirlGetTaps()` or `nsp?FirlSetTaps()`, you must initialize the filter tap state *tapStPtr* by calling either `nsp?FirlInit()` or `nsp?FirlInitMr()`. The data type used during initialization must match the data type used here.

**nsp?FirlGetTaps().** The `nsp?FirlGetTaps()` function copies the tap coefficients from the array *taps[n]* to the *tapsLen* length array *outTaps[n]*, unpermuting them if required so that *outTaps*[*n*] = *h*(*n*).

**nsp?FirlSetTaps().** The `nsp?FirlSetTaps()` function copies the *tapsLen* tap coefficients from the array *inTaps[n]* into the array *taps[n]*, permuting them if required.

**Application Notes:** The `nsp?FirlGetTaps()` and `nsp?FirlSetTaps()` functions can be used to permute or unpermute an FIR filter's taps in-place or not-in-place. That is, if the pointer *inTaps* points to an array other than *taps[n]* (for `nsp?FirlSetTaps()`), or if *outTaps* points to an array other than *taps[n]* (for `nsp?FirlGetTaps()`), then the permutation is performed not-in-place.

If, on the other hand, `inTaps` or `outTaps` points to the same array, `taps[n]`, then the permutation is performed in-place. You might want your application to do this to avoid allocating a separate array to hold the permuted values. However, if your application unpermutes the `taps[n]` array in-place (via `nsp?FirlGetTaps()`), the `taps[n]` array must be re-permuted (via `nsp?FirlSetTaps()`) before the filter can be used again. Thus, you must use caution when permuting in-place.

When running on Pentium® processor, the function `nspwFirlSetTaps()` only permutes user-defined filter taps. On Pentium processor with MMX™ technology or Pentium Pro processor, the function `nspwFirlSetTaps()` converts user-defined filter taps from floating-point to short format and permutes them. Short filter taps ensure high performance on both Pentium processor with MMX technology and Pentium Pro processor (but may cause some loss of precision). The conversion of the floating-point filter taps `h[k]` to the short taps `sh[k]` is as follows:

$$sh[k] = Round(2^f \times h[k])$$

where:

$$\frac{1}{2} \times \mathrm{NSP\_MAX\_SHORT\_INT} \le 2^f \times \sum_{k=0}^{tapsLen-1} |h[k]| < \mathrm{NSP\_MAX\_SHORT\_INT}$$

The above factor $f$ is stored in the `tapStPtr` stucture.

The `nspwFirlSetTaps()` function running in-place on Pentium processor with MMX technology or Pentium Pro processor will destroy the original user-defined filter taps. Using the not-in-place operation is safe in all cases, as shown in Example 8-2.

### Example 8-2 Single-rate Filtering of Two Signals with the nspwbFirl() Function

```
   /* filtering two signals with a single FIR filter */
NSPFirTapState tapSt;
NSPFirDlyState lChannelDlySt;
NSPFirDlyState rChannelDlySt;
float usr_taps[32];   /* usr_taps controlled by user */
float fir_taps[32];   /* fir_taps controlled by FIR filter */
short lChannelDelay[32]; /* delay line of signal 1 */
short rChannelDelay[32]; /* delay line of signal 2 */
short lChannelX[1000];   /* input buffer of signal 1 */
short lChannelY[1000];   /* output buffer of signal 1 */
short rChannelX[1000];   /* input buffer of signal 2 */
short rChannelY[1000];   /* output buffer of signal 2 */
int i;
   /* insert code here to initialize usr_taps */
   /* zeros fir_taps array (you can choose another way
      to set proper values in the fir_taps array)*/
memset(fir_taps, 0, 32*sizeof(float));
   /* initialize the filter taps and delay lines */
nspwFirlInit(fir_taps, 32, &tapSt);
nspwFirlInitDlyl(&tapSt, lChannelDelay, &lChannelDlySt);
nspwFirlInitDlyl(&tapSt, rChannelDelay, &rChannelDlySt);
   /* not-in-place setting filter taps */
nspwFirlSetTaps(usr_taps, &tapSt);
   /* zeros: both delay lines */
nspwFirlSetDlyl(&tapSt, NULL, &lChannelDlySt);
nspwFirlSetDlyl(&tapSt, NULL, &rChannelDlySt);
for(i=0; i < 20; i++) {
   /* get next block of input signal 1 into the lChannelX */;
   /* get next block of input signal 2 into the rChannelX */;
   /* filtering two signals */
   nspwbFirl(&tapSt, &lChannelDlySt, lChannelX, lChannelY,
   1000, 0,0);
   nspwbFirl(&tapSt, &rChannelDlySt, rChannelX, rChannelY,
   1000, 0,0)
   /* lChannelY, rChannelY contains two output blocks */
}
```

The `nspwFirlGetTaps()` function uses the factor *f* to return filter taps.

This function unpermutes the tap coefficients from *`tapStPtr`* only if it runs on a Pentium® processor. Otherwise, the `nspwFirlGetTaps()` function converts tap coefficients from internal representation (short) to the external (floating-point) format.

### Related Topics

| | |
|---|---|
| `bFirl` | Filters a block of samples through a low-level FIR filter (see page 8-9). |
| `Firl` | Filters a single sample through a low-level FIR filter (see page 8-9). |
| `FirlInit` | Initializes a single-rate, low-level FIR filter (see page 8-23). |
| `FirlInitDlyl` | Initializes the delay line for a low-level FIR filter (see page 8-4). |
| `FirlInitMr` | Initializes a multi-rate, low-level FIR filter (see page 8-4). |

## FirlGetDlyl, FirlSetDlyl

*Gets and sets the delay line contents of low-level FIR filters.*

```
void nspsFirlGetDlyl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlylStPtr, float *outDlyl);
void nspsFirlSetDlyl(const NSPFirTapState *tapStPtr, float *inDlyl,
      NSPFirDlyState *dlylStPtr);
      /* real values; single precision */
void nspcFirlGetDlyl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlylStPtr, SCplx *outDlyl);
void nspcFirlSetDlyl(const NSPFirTapState *tapStPtr, SCplx *inDlyl,
      NSPFirDlyState *dlylStPtr);
      /* complex values; single precision */
```

```
void nspdFirlGetDlyl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlylStPtr, double *outDlyl);
void nspdFirlSetDlyl(const NSPFirTapState *tapStPtr,
      double *inDlyl, NSPFirDlyState *dlylStPtr);
      /* real values; double precision */
void nspzFirlGetDlyl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlylStPtr, DCplx *outDlyl);
void nspzFirlSetDlyl(const NSPFirTapState *tapStPtr, DCplx *inDlyl,
      NSPFirDlyState *dlylStPtr);
      /* complex values; double precision */
void nspwFirlGetDlyl(const NSPFirTapState *tapStPtr,
      NSPFirDlyState *dlylStPtr, short *outDlyl);
void nspwFirlSetDlyl(const NSPFirTapState *tapStPtr, short *inDlyl,
      NSPFirDlyState *dlylStPtr);
      /* real values; short integer */
```

| | |
|---|---|
| *dlylStPtr* | Pointer to the `NSPFirDlylState` structure. |
| *inDlyl* | Pointer to the array holding copies of the delay line values for the `nsp?FirlSetDlyl()` function. |
| *outDlyl* | Pointer to the array holding copies of the delay line values for the `nsp?FirlGetDlyl()` function. |
| *tapStPtr* | Pointer to the `NSPFirTapState` structure. |

## Discussion

These `nsp?FirlGetDlyl()` and `nsp?FirlSetDlyl()` functions provide a safe mechanism to get and set the delay line values of a low-level FIR filter. Because the delay line might be stored in permuted order, it is not safe for the application to directly access the delay line array. Instead, `nsp?FirlGetDlyl()` and `nsp?FirlSetDlyl()` should be used.

**Previous Tasks:** Before calling either `nsp?FirlGetDlyl()` or `nsp?FirlSetDlyl()`, you must initialize the filter tap state pointed to by *tapStPtr*, the (permuted) taps array *taps[n]*, and the filter length *tapsLen* by calling either `nsp?FirlInit()` or `nsp?FirlInitMr()`. In addition, you must initialize the delay line state pointed to by *dlylStPtr* and the (permuted) delay line array *dlyl[n]* by calling `nsp?FirlInitDlyl()`. You must also update the delay line pointer *dlylStPtr* by calling `nsp?FirlSetDlyl()`. Both `nsp?FirlGetDlyl()` and `nsp?FirlSetDlyl()` require *tapStPtr* as an argument to describe the delay line permutation. The data type used for these functions must match the data type of the delay line initialization (and not the data type of the taps initialization).

**nsp?FirlGetDlyl().** The nsp?FirlGetDlyl() function copies the delay line values from the array *dlyl[n]* and stores them into the array *outDlyl[n]*. The function also unpermutes the delay line values if necessary so that *outDlyl[k]* = *x*(*n* - *tapsLen* + *2* + *k*), where *x(n)* is the last filtered sample. For single-rate filters, *outDlyl[n]* must be *tapsLen* - 1 long. For multi-rate filters *outDlyl[n]* must be *PL* long, where *PL* is defined as

$$PL = \lceil tapsLen/upFactor \rceil.$$

**nsp?FirlSetDlyl().** The nsp?FirlSetDlyl() function permutes the values in the array *inDlyl[n]*, stores them into *dlyl[n]*, and updates *dlylStPtr*. For single-rate filters, *inDlyl[n]* must be *tapsLen* - 1 long, and for multi-rate filters it must be *PL* long. If *inDlyl* is NULL, the delay line is initialized to all zeros.

**Application Notes:** The nsp?FirlGetDlyl() and nsp?FirlSetDlyl() functions can be used to permute or unpermute an FIR filter's taps in-place or not-in-place. That is, if the pointer *inDlyl* points to an array other than *dlyl[n]* (for nsp?FirlSetDlyl()), or if *outDlyl* points to an array other than *dlyl[n]* (for nsp?FirlGetDlyl()), then the permutation is performed not-in-place.

If, on the other hand, *inDlyl* or *outDlyl* points to the same array, *dlyl[n]*, then the permutation is performed in-place. You might want your application to do this to avoid allocating a separate array to hold the permuted values. However, if your application unpermutes the *dlyl[n]* array in-place (via nsp?FirlGetDlyl()), the *dlyl[n]* array must be re-permuted (via nsp?FirlSetDlyl()) before the filter can be used again. Thus, you must use caution when permuting in-place.

## Related Topics

| | |
|---|---|
| bFirl | Filters a block of samples through a low-level FIR filter (see page 8-9). |
| Firl | Filters a single sample through a low-level FIR filter (see page 8-9). |

| | |
|---|---|
| `FirlInit` | Initializes a single-rate, low-level FIR filter (see page 8-4). |
| `FirlInitDlyl` | Initializes the delay line for a low-level FIR filter (see page 8-4). |
| `FirlInitMr` | Initializes a multi-rate, low-level FIR filter (see page 8-4). |

# FIR Filter Functions

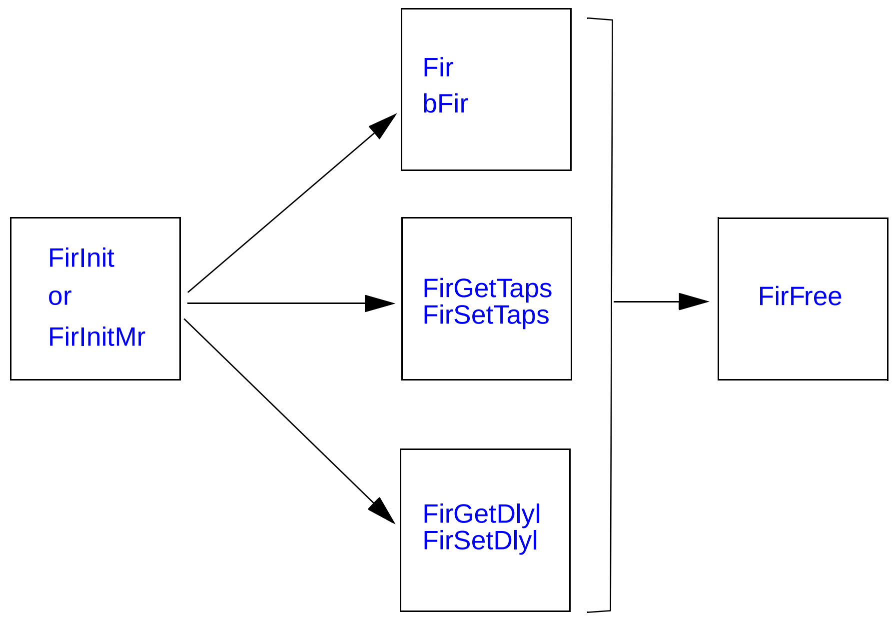
The functions described in this section initialize a finite impulse response filter, get and set the delay line and filter coefficients (taps) and perform the filtering function. They are intended for cyclic processing. For batch mode filtering, see “Conv” in Chapter 9 for a description of the `nsp?Conv()` function.

These functions provide a higher-level interface than the corresponding low-level FIR functions (see “Firl” on page 8-9 for a description of `nsp?Firl()`). In particular, they bundle the taps and delay line into a single state.

Also, the FIR filter functions dynamically allocate memory for the taps and delay line; thus the arrays storing the taps and delay line values are not accessed after initialization, and need not exist while the filter exists.

Figure 8-2 illustrates the order of use of the FIR filter functions.

**Figure 8-2 Order of Use of the FIR Functions**




To use the FIR filter functions, follow this general scheme:

1. Call either `nsp?FirInit()` to initialize the coefficients, delay line, and structure of a single-rate filter, or call `nsp?FirInitMr()` to initialize the coefficients, delay line, and structure of a multi-rate filter.
2. After this initialization, you have a choice of functions to call, depending on what you want to accomplish.
   a. Call the `nsp?Fir()` function to filter a single sample through a single-rate filter and/or call `nsp?bFir()` to filter a block of consecutive samples through a single-rate or multi-rate filter.
   b. Call the `nsp?FirGetTaps()` function and then the `nsp?FirSetTaps()` function to get and set the filter coefficients (taps).

c. Call the `nsp?FirGetDlyl()` function and then the `nsp?FirSetDlyl()` function to get and set the values in the delay line.

3. Call the `nspFirFree()` function to free dynamic memory associated with the FIR filter.

Real and complex taps can be mixed with real and complex delay lines (that is, all four combinations are allowable). However, taps and delay lines of different precision must not be mixed. It is the application's responsibility to call the correct function for the given type combination. This is not checked at compile time nor is it required to be checked at run-time.

## FirInit, FirInitMr, FirFree

*Initializes a finite impulse response filter.*

```
void nspsFirInit(const float *tapVals, int tapsLen,
      const float *dlyVals, NSPFirState *statePtr);
void nspsFirInitMr(const float *tapVals, int tapsLen,
      const float *dlyVals,int upFactor, int upPhase,
      int downFactor, int downPhase, NSPFirState *statePtr);
      /* real delay line, real taps; single precision */

void nspcFirInit(const SCplx *tapVals, int tapsLen,
      const SCplx *dlyVals, NSPFirState *statePtr);
void nspcFirInitMr(const SCplx *tapVals, int tapsLen,
      const SCplx *dlyVals, int upFactor, int upPhase,
      int downFactor, int downPhase,  NSPFirState *statePtr);
      /* complex delay line, complex taps; single precision */

void nspscFirInit(const SCplx *tapVals, int tapsLen,
      const float *dlyVals, NSPFirState *statePtr);
void nspscFirInitMr(const SCplx *tapVals, int tapsLen,
      const float *dlyVals,int upFactor, int upPhase,
      int downFactor, int downPhase,  NSPFirState *statePtr);
      /* real delay line, complex taps; single precision */

void nspcsFirInit(const float *tapVals, int tapsLen,
      const SCplx *dlyVals, NSPFirState *statePtr);
```

```
void nspcsFirInitMr(const float *tapVals, int tapsLen,
      const SCplx *dlyVals, int upFactor, int upPhase,
      int downFactor, int downPhase,  NSPFirState *statePtr);
      /* complex delay line, real taps; single precision */

void nspdFirInit(const double *tapVals, int tapsLen,
      const double *dlyVals, NSPFirState *statePtr);
void nspdFirInitMr(const double *tapVals, int tapsLen,
      const double *dlyVals, int upFactor, int upPhase,
      int downFactor, int downPhase,  NSPFirState *statePtr);
      /* real delay line, real taps; double precision */
void nspzFirInit(const DCplx *tapVals, int tapsLen,
      const DCplx *dlyVals, NSPFirState *statePtr);
void nspzFirInitMr(const DCplx *tapVals, int tapsLen,
      const DCplx *dlyVals, int upFactor, int upPhase,
      int downFactor, int downPhase,  NSPFirState *statePtr);
      /* complex delay line, complex taps; double precision */

void nspdzFirInit(const DCplx *tapVals, int tapsLen,
      const double *dlyVals, NSPFirState *statePtr);
void nspdzFirInitMr(const DCplx *tapVals, int tapsLen,
      const double *dlyVals, int upFactor, int upPhase,
      int downFactor, int downPhase,  NSPFirState *statePtr);
      /* real delay line, complex taps; double precision */

void nspzdFirInit(const double *tapVals, int tapsLen,
      const DCplx *dlyVals, NSPFirState *statePtr);
void nspzdFirInitMr(const double *tapVals, int tapsLen,
      const DCplx *dlyVals, int upFactor, int upPhase,
      int downFactor, int downPhase,  NSPFirState *statePtr);
      /* complex delay line, real taps; double precision */

void nspwFirInit(const float *tapVals, int tapsLen,
      const short *dlyVals, NSPFirState *statePtr);
void nspwFirInitMr(const float *tapVals, int tapsLen,
      const short *dlyVals,int upFactor, int upPhase,
      int downFactor, int downPhase, NSPFirState *statePtr);
      /* real delay line, real taps; short integer */

void nspFirFree(NSPFirState *statePtr);
      /* releases all dynamic memory associated with
         FIR filter */
```

| | |
|---|---|
| *dlyVals* | Pointer to the array containing the delay line values. |
| *downFactor* | The factor used by the `nsp?FirInitMr()` function for down-sampling multi-rate signals. |
| *downPhase* | The phase value used by the `nsp?FirInitMr()` function for down-sampling multi-rate signals. |
| *statePtr* | Pointer to the `NSPFirState` structure. |
| *tapVals* | Pointer to the array containing the filter coefficient (taps) values. |
| *tapsLen* | The number of values in the array containing the filter coefficients (taps). |
| *upFactor* | The factor used by the `nsp?FirInitMr()` function for up-sampling multi-rate signals. |
| *upPhase* | The phase value used by the `nsp?FirInitMr()` function for up-sampling multi-rate signals. |

## Discussion

The `nsp?FirInit()` and `nsp?FirInitMr()` functions initialize a finite impulse response filter. They are intended for cyclic processing. For batch mode filtering, see "Conv" in Chapter 9 for the description of the `nsp?Conv()` function.

The `nsp?FirInit()` and `nsp?FirInitMr()` functions provide a higher-level interface than the corresponding low-level FIR functions (see "FirlInit" and "FirlInitMr" on page 8-4 for a description of `nsp?FirlInit()` and `nsp?FirlInitMr()`). In particular, they bundle the taps and delay line into the state structure `NSPFirState`. Also, `nsp?FirInit()` and `nsp?FirInitMr()` dynamically allocate memory for the taps and delay line arrays. Thus the data in the arrays *tapVals[n]* and *dlyVals[n]* need not exist while the filter exists. That is, your application can overwrite or deallocate the values in *tapVals[n]* and *dlyVals[n]* after calling the `nsp?FirInit()` or `nsp?FirInitMr()` function.

Many combinations of real and complex delay lines and filter coefficients are possible. This is indicated by the `s`, `c`, `sc`, `cs`, `d`, `z`, `dz`, `zd`, and `w` type codes following the `nsp` prefix in the function names above. For both of the functions, `nsp?FirInit()` and `nsp?FirInitMr()`, the allowed combinations of real and complex taps and delay lines are described in Table 8-2.

**Table 8-2 Delay Line and Taps Combinations for nsp?FirInit() and nsp?FirInitMr() Functions**

| Type Codes | Delay Line Type | Filter Coefficient (or taps) Type | y(n) (or output) Type |
|---|---|---|---|
| s | float | float | float |
| c | SCplx | SCplx | SCplx |
| sc | float | SCplx | SCplx |
| cs | SCplx | float | SCplx |
| d | double | double | double |
| z | DCplx | DCplx | DCplx |
| dz | double | DCplx | DCplx |
| zd | DCplx | double | DCplx |
| w | short | float | short |

**nsp?FirInit().** The `nsp?FirInit()` function initializes a single-rate filter. The *tapsLen* length array *tapVals[n]* specifies the filter coefficients as follows:

$tapVals[k] = h(k),\ 0 \le k < tapsLen$

If the *tapsLen* - 1 length array *dlyVals[n]* is non-`NULL`, the following equation provides initial samples for the delay line:

$dlylVals[k] = x(-tapsLen + 1 + k),\ 0 \le k < tapsLen - 1$

where $x(0)$ will be the first sample filtered. If *dlyVals[n]* is `NULL`, the delay line is initialized to zero.

**nsp?FirInitMr().** The `nsp?FirInitMr()` function initializes a multi-rate filter; that is, a filter that internally up-samples and/or down-samples using a polyphase filter structure. It initializes the

`NSPFirState` structure pointed to be *`statePtr`* in the same way as described for single-rate filters, but includes additional information about the required up-sampling and down-sampling parameters.

The argument *`upFactor`* is the factor by which the filtered signal is internally up-sampled (see “UpSample” in Chapter 3). That is, *`upFactor`* - 1 zeros are inserted between each sample of input signal.

The argument *`upPhase`* is the parameter which determines where a non-zero sample lies within the *`upFactor`*-length block of up-sampled input signal.

The argument *`downFactor`* is the factor by which the FIR response obtained by filtering an up-sample input signal is internally down-sampled (see “DownSample” in Chapter 3). That is, *`downFactor`*-1 output samples are discarded from each *`downFactor`*-length output block of up-sampled filter response.

The argument *`downPhase`* is the parameter which determines where non-discarded sample lies within a block of up-sampled filter response.

The delay line array *`dlyVals[n]`* is defined in the same way as in the single-rate case, but if the array is non-`NULL` its length is defined as $PL = \lceil tapsLen/upFactor \rceil$.

**Application Notes:** Taps of the integer filter are given in floating-point format. The internal usage of these taps is different on different processors. On Pentium® processor, the functions `nspwFirInit()` and `nspwFirInitMr()` copy user-defined filter taps into a dynamically allocated array. On the other hand, on Pentium processor with MMX™ technology or Pentium Pro processor these functions convert the user-defined filter taps from floating-point to short format and then store them in a dynamically allocated array. The float-to-short conversion uses the following formula:

$$sh[k] = Round(2^f \times h[k])$$

where

$$\frac{1}{2} \times \mathrm{NSP\_MAX\_SHORT\_INT} \le 2^f \times \sum_{k=0}^{tapsLen-1} |h[k]| < \mathrm{NSP\_MAX\_SHORT\_INT}$$

**nspFirFree().** The nspFirFree() function frees all memory associated with a filter created by either the nsp?FirInit() or nsp?FirInitMr() function. You should call nspFirFree() after the application has finished filtering with *statePtr*. After calling nspFirFree(), you should not reference *statePtr* again.

**Application Notes:** The contents of the NSPFirState structure are implementation-dependent. The contents of NSPFirState includes a dynamically allocated array for the taps and delay line. For more information, see the "Application Notes" on page 8-7 for the "FirlInit" and "FirlInitMr" sections (that is, for the low-level functions nsp?FirlInit() and nsp?FirlInitMr()).

## Related Topics

| | |
|---|---|
| upSample | Up-samples a signal, conceptually increasing its sampling rate by an integer factor (see page 3-55). |
| downSample | Down-samples a signal, conceptually decreasing its sampling rate by an integer factor (see page 3-58). |
| bFir | Filters a block of samples through an FIR filter (see page 8-29). |
| Fir | Filters a single sample through an FIR filter (see page 8-29). |
| FirGetDlyl | Gets the delay line contents for an FIR filter (see page 8-35). |
| FirGetTaps | Gets the tap coefficients for an FIR filter (see page 8-34). |
| FirSetDlyl | Sets the delay line contents for an FIR filter (see page 8-35). |
| FirSetTaps | Sets the tap coefficients for an FIR filter (see page 8-34). |

## Fir, bFir

*Performs finite impulse response filtering.*

```
float nspsFir(NSPFirState *statePtr, float samp);
void nspsbFir(NSPFirState *statePtr, const float *inSamps,
      float *outSamps, int numIters);
      /* real delay line, real taps; single precision */

SCplx nspcFir(NSPFirState *statePtr, SCplx samp);
void nspcbFir(NSPFirState *statePtr, const SCplx *inSamps,
      SCplx *outSamps, int numIters);
      /* complex delay line, complex taps; single precision */

SCplx nspscFir(NSPFirState *statePtr, float samp);
void nspscbFir(NSPFirState *statePtr, const float *inSamps,
      SCplx *outSamps, int numIters);
      /* real delay line, complex taps; single precision */

SCplx nspcsFir(NSPFirState *statePtr, SCplx samp);
void nspcsbFir(NSPFirState *statePtr, const SCplx *inSamps,
      SCplx *outSamps, int numIters);
      /* complex delay line, real taps; single precision */

double nspdFir(NSPFirState *statePtr, double samp);
void nspdbFir(NSPFirState *statePtr, const double *inSamps,
      double *outSamps, int numIters);
      /* real delay line, real taps; double precision */

DCplx nspzFir(NSPFirState *statePtr, DCplx samp);
void nspzbFir(NSPFirState *statePtr, const DCplx *inSamps,
      DCplx *outSamps, int numIters);
      /* complex delay line, complex taps; double precision */

DCplx nspdzFir(NSPFirState *statePtr, double samp);
void nspdzbFir(NSPFirState *statePtr, const double *inSamps,
      DCplx *outSamps, int numIters);
      /* real delay line, complex taps; double precision */
```

```
DCplx nspzdFir(NSPFirState *statePtr, DCplx samp);
void nspzdbFir(NSPFirState *statePtr, const DCplx *inSamps,
      DCplx *outSamps, int numIters);
      /* complex delay line, real taps; double precision */

float nspwFir(NSPFirState *statePtr, short samp, int ScaleMode,
      int *ScaleFactor);
void nspwbFir(NSPFirState *statePtr, const short *inSamps,
      short *outSamps, int numIters, int ScaleMode,
      int *ScaleFactor);
      /* real delay line, real taps; short integer */
```

| | |
|---|---|
| *inSamps* | Pointer to the array which stores the input samples to be filtered by the `nsp?bFir()` function. |
| *numIters* | Parameter associated with the number of samples to be filtered by the `nsp?bFir()` function. For single-rate filters, the *numIters* samples in the array *inSamps[n]* are filtered and the resulting *numIters* samples are stored in the array *outSamps[n]*. For multi-rate filters, the (*numIters* * *downFactor*) samples in the array *inSamps[n]* are filtered and the resulting (*numIters* * *upFactor*) samples are stored in the array *outSamps[n]*. |
| *outSamps* | Pointer to the array which stores the output samples filtered by the `nsp?bFir()` function. |
| *samp* | The input sample to be filtered by the `nsp?bFir()` function. |
| *statePtr* | Pointer to the `NSPFirState` structure. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

The `nsp?Fir()` and `nsp?bFir()` functions perform finite impulse response filtering. The `nsp?Fir()` function filters a single sample through a single-rate filter and the `nsp?bFir()` function filters a block of consecutive samples through a single-rate or multi-rate filter.

**Previous Tasks:** Before calling either the `nsp?Fir()` or `nsp?bFir()` function, you must initialize the filter state pointed to by *`statePtr`* by calling either `nsp?FirInit()` or `nsp?FirInitMr()`. You must specify the number of taps *`tapsLen`*, and the taps values, denoted as *h*(0)...*h*(*`tapsLen`* - 1). You must also specify the delay line values. For single-rate filters the values are denoted as *x*(*n*-*`tapsLen`* + 1)...*x*(*n* - 1); for multi-rate filters, they are denoted as *x*(*n* - *PL*)...*x*(*n* - 1).

The data type of the function used here must match the data type of the function used for initialization. For example, if the filter was initialized with `nspcsFirInit()`, use the `nspcsFir()` function to filter the sample. For a description of the initialization functions, see `nsp?FirInit()` and `nsp?FirInitMr()` on page 8-23.

Table 8-3 describes the `s`, `c`, `sc`, `cs`, `d`, `z`, `dz`, `zd`, and `w` type codes following the `nsp` prefix.

**Table 8-3 Delay Line and Taps Combinations for nsp?Fir() and nsp?bFir() Functions**

| Type Codes | Delay Line Type | Filter Coefficient (or taps) Type | y(n) (or output) Type |
|---|---|---|---|
| s | float | float | float |
| c | SCplx | SCplx | SCplx |
| sc | float | SCplx | SCplx |
| cs | SCplx | float | SCplx |
| d | double | double | double |
| z | DCplx | DCplx | DCplx |
| dz | double | DCplx | DCplx |
| zd | DCplx | double | DCplx |
| w | short | float\|short | short |

**`nsp?Fir()`.** The `nsp?Fir()` function filters a single sample through a single-rate filter. In the following definition of the FIR filter, the sample to be filtered is denoted *`x(n)`* and the filter coefficients are denoted *`h(k)`*.

The return value `y(n)` is calculated as follows:

$$y(n) = \sum_{k=0}^{tapsLen-1} h(k) \cdot x(n-k)$$

The `nsp?Fir()` function then updates the delay line `dlyl[n]` to contain `x(n)`, and discards the value $x(n$ - `tapsLen` $+ 1)$ from the delay line.

**`nsp?bFir()`**. The `nsp?bFir()` function filters a block of consecutive samples through a single-rate or multi-rate filter. For single-rate filters, the `numIters` samples in the array `inSamps[n]` are filtered, and the resulting `numIters` samples are stored in the array `outSamps[n]`. The results are identical to `numIters` consecutive calls to `nsp?Fir()`. The values in the `outSamps[n]` array are calculated as follows:

`inSamps[m]` $= y(n + m), 0 \le m <$ `numIters`

$$y(n+m) = outSamps[m] = \sum_{m=0}^{numIters-1} h(k) \cdot x(n+m-k)$$

For multi-rate filters, `nsp?bFir()` filters the (`numIters` * `downFactor`) samples in the array `inSamps[n]`, and stores the resulting (`numIters` * `upFactor`) samples in the array `outSamps[n]`.
See Appendix C for more information about multi-rate filters. For both single-rate and multi-rate filters, the appropriate number of samples from `inSamps[n]` are copied into the delay line, and the oldest samples are discarded.

**Application Notes:** The `nspwFir()` and `nspwbFir()` functions use the floating-point taps directly if your computer has a Pentium® processor. Otherwise, these functions convert the taps from floating-point to short format. The calculations involve fixed-point operations. This improves the performance on Pentium processors with MMX™ technology and Pentium Pro processors. On the other hand, taps in the short format can cause accuracy problems.

Example 8-3 illustrates single-rate filtering with the `nsp?Fir()` function.

**Example 8-3 Single-Rate Filtering with the nsp?Fir() Function**

```
/* standard
 * single-rate filtering
 */
NSPFirState firSt;
double h[32], xval, yval;
int    i;

/* insert code here to initialize h */

nspdFirInit(h, 32, NULL, &firSt);
for (i=0; i < 2000; i++) {
   xval = /* insert code here to get
           * next value of x(n)
           */;
   yval = nspdFir(&firSt, xval);
   /* yval has the output sample */
   }
```

## Related Topics

| | |
|---|---|
| `bFirl` | Filters a block of samples through a single-rate or multi-rate low-level FIR filter (see page 8-9). |
| `Firl` | Filters a single sample through a single-rate. low-level FIR filter (see page 8-9). |
| `FirInit` | Initializes a single-rate FIR filter (see page 8-23). |
| `FirInitMr` | Initializes a multi-rate FIR filter (see page 8-23). |

## FirGetTaps, FirSetTaps

*Gets and sets the tap coefficients of FIR filters.*

```
void nspsFirGetTaps(const NSPFirState *statePtr, float *outTaps);
void nspsFirSetTaps(const float *inTaps, NSPFirTapState *statePtr);
      /* real values; single precision */

void nspcFirGetTaps(const NSPFirTapState *statePtr, SCplx *outTaps);
void nspcFirSetTaps(const SCplx *inTaps, NSPFirTapState *statePtr);
      /* complex values; single precision */

void nspdFirGetTaps(const NSPFirState *statePtr, double *outTaps);
void nspdFirSetTaps(const double *inTaps, NSPFirTapState *statePtr);
      /* real values; double precision */

void nspzFirGetTaps(const NSPFirTapState *statePtr, DCplx *outTaps);
void nspzFirSetTaps(const DCplx *inTaps, NSPFirTapState *statePtr);
      /* complex values; double precision */

void nspwFirGetTaps(const NSPFirState *statePtr, float *outTaps);
void nspwFirSetTaps(const float *inTaps, NSPFirTapState *statePtr);
      /* real values; short integer */
```

| | |
|---|---|
| *inTaps* | Pointer to the array holding copies of the tap coefficients. |
| *outTaps* | Pointer to the array holding copies of the tap coefficients. |
| *statePtr* | Pointer to the `NSPFirState` structure. |

### Discussion

The `nsp?FirGetTaps()` and `nsp?FirSetTaps()` functions get and set the taps of a FIR filter. The data type of the function used here must match the data type for the taps used during initialization.

**Previous Tasks:** Before calling either `nsp?FirGetTaps()` or `nsp?FirSetTaps()`, you must initialize the state structure `NSPFirState` pointed to by *`statePtr`* by calling either `nsp?FirInit()` or `nsp?FirInitMr()`. You must also specify the tap length *`tapsLen`* and the taps *h*(0)...*h*(*`tapsLen`* - 1).

**`nsp?FirGetTaps()`.** The `nsp?FirGetTaps()` function copies the tap coefficients from *`statePtr`* to the *`tapsLen`* length array *`outTaps[n]`*, unpermuting them if required so that *`outTaps[n]`* = *`h(n)`*.

**Application Notes:** The `nspwFirGetTaps()` function copies the taps from *`statePtr`* if and only if it runs on a Pentium® processor. Otherwise, the `nspwFirGetTaps()` function converts the taps from the internal (short) format to the float format.

**`nsp?FirSetTaps()`.** The `nsp?FirSetTaps()` function copies the *`tapsLen`* tap coefficients from the *`inTaps[n]`* array into *`statePtr`*, permuting them if required so that *`h(n)`* = *`inTaps[n]`*.

### Related Topics

| | |
|---|---|
| `bFir` | Filters a block of samples through a single-rate or multi-rate FIR filter (see page 8-29). |
| `Fir` | Filters a single sample through a single-rate FIR filter (see page 8-29). |
| `FirInit` | Initializes a single-rate FIR filter (see page 8-23). |
| `FirInitMr` | Initializes a multi-rate FIR filter (see page 8-23). |

## FirGetDlyl, FirSetDlyl

*Gets and sets the delay line contents of FIR filters.*

```
void nspsFirGetDlyl(const NSPFirState *statePtr, float *outDlyl);
void nspsFirSetDlyl(const float *inDlyl, NSPFirState *statePtr);
      /* real values; single precision */
```

```
void nspcFirGetDlyl(const NSPFirState *statePtr, SCplx *outDlyl);
void nspcFirSetDlyl(const SCplx *inDlyl, NSPFirState *statePtr);
      /* complex values; single precision */

void nspdFirGetDlyl(const NSPFirState *statePtr, double *outDlyl);
void nspdFirSetDlyl(const double *inDlyl, NSPFirState *statePtr);
      /* real values; double precision */

void nspzFirGetDlyl(const NSPFirState *statePtr, DCplx *outDlyl);
void nspzFirSetDlyl(const DCplx *inDlyl, NSPFirState *statePtr);
      /* complex values; double precision */

void nspwFirGetDlyl(const NSPFirState *statePtr, short *outDlyl);
void nspwFirSetDlyl(const short *inDlyl, NSPFirState *statePtr);
      /* real values; short integer */
```

| | |
|---|---|
| *inDlyl* | Pointer to the array holding copies of the delay line values for the `nsp?FirSetDlyl()` function. |
| *outDlyl* | Pointer to the array holding copies of the delay line values for the `nsp?FirGetDlyl()` function. |
| *statePtr* | Pointer to the `NSPFirState` structure. |

## Discussion

The `nsp?FirGetDlyl()` and `nsp?FirSetDlyl()` functions get and set the delay line of an FIR filter. The data type of the function used here must match the data type for the delay line used during initialization.

**Previous Tasks:** Before calling either `nsp?FirGetDlyl()` or `nsp?FirSetDlyl()`, you must initialize the state structure `NSPFirState` pointed to by *statePtr* by calling either `nsp?FirInit()` or `nsp?FirInitMr()`. You must also specify the tap length *tapsLen* and the delay line values. For single-rate filters, the delay line values are denoted as $x(n - tapsLen + 1)...x(n - 1)$; for multi-rate filters they are denoted as $x(n - PL)...x(n - 1)$.

**`nsp?FirGetDlyl()`.** The `nsp?FirGetDlyl()` function unpermutes the delay line values in *statePtr* and stores them into the array *outDlyl[n]* so that *outDlyl[k]* $= x(n - tapsLen + 2 - k)$, where *x(n)* is the last filtered sample. For single-rate filters, the array *`outDlyl[n]`* must be *`tapsLen`* - 1 long, and for multi-rate it must be *`PL`* long, where *`PL`* is defined as follows:

*`PL`* = ⌈ *`tapsLen`*/*`upFactor`*⌉.

**`nsp?FirSetDlyl()`.** The `nsp?FirSetDlyl()` function permutes the values in the array *`inDlyl[n]`* and stores them into *`statePtr`*. For single-rate filters, *`inDlyl[n]`* must be *`tapsLen`* - 1 long, and for multi-rate filters, it must be *`PL`* long. If *`inDlyl`* is `NULL`, the delay line is initialized to all zeros.

### Related Topics

| | |
|---|---|
| `bFir` | Filters a block of samples through a single-rate or multi-rate FIR filter (see page 8-29). |
| `Fir` | Filters a single sample through a single-rate FIR filter (see page 8-29). |
| `FirInit` | Initializes a single-rate FIR filter (see page 8-23). |
| `FirInitMr` | Initializes a multi-rate FIR filter (see page 8-23). |

## FIR Filter Design Functions

This section describes the library's filter design functions. Use these functions to compute the taps for lowpass, highpass, bandpass, and bandstop FIR filters. Then you can pass the computed taps to the FIR filters and low-level FIR filters described earlier in this chapter.

The main input information for computing the taps is cut-off frequency for lowpass and highpass filters, or a pair of frequencies for bandpass and bandstop filters. For more information on FIR filter design, see [Mit93] and [Cap78].

## FirLowpass

*Computes taps for a lowpass FIR filter.*

```
int nspdFirLowpass (double rfreq, double *taps, int tapsLen,
    NSP_WindowType winType, int doNormal);
    /* real values; double precision */
```

| | |
|---|---|
| *rfreq* | Cut-off frequency value (0 < *rfreq* < 0.5). |
| *taps* | Pointer to the vector of taps to be computed. |
| *tapsLen* | The number of taps in *taps* (*tapsLen* ≥ 5). |
| *winType* | Specifies the smoothing window type. Can be one of the following: |

| | |
|---|---|
| NSP_WinRect | no smoothing (rectangular window); |
| NSP_WinBartlett | smoothing by Bartlett window; |
| NSP_WinBlackmanOpt | smoothing by optimal Blackman window; |
| NSP_WinHamming | smoothing by Hamming window; |
| NSP_WinHann | smoothing by Hann window. |

| | |
|---|---|
| *doNormal* | Sets the normalization mode. If *doNormal* = 0, the function computes taps without normalization; otherwise, the taps will be normalized. |

### Discussion.

This function computes the taps for a lowpass FIR filter. It forms an ideal (infinite) impulse response of a lowpass filter and smoothes the response data by the specified window. If *doNormal* is non-zero, the function normalizes the taps. (For normalized taps, the response is 1 near the zero frequency.)

The function returns one of the following values:

| | |
|---|---|
| NSP_StsOk | if the taps have been computed successfully |
| NSP_fStsBadPointer | if the pointer to the *taps* vector is null |
| NSP_fStsBadLen | if the number of taps (*tapsLen*) is less than five |
| NSP_fStsBadFreq | if the frequencies are not in the interval (0, 0.5) |

Example 8-4 illustrates the usage of the `nspdFirLowpass()` function.

**Example 8-4 Using nspdFirLowpass to design a lowpass FIR filter**

```
/*  using the lowpass FIR filter design function */

   NSPFirState firSt;
   double rfreq = 0.3; /* cut-off frequency  */
   double h[32];
   double input[2000];
   double output[2000];

   /* insert code here to initialize input   */

   /* design lowpass filter                  */
   nspdFirLowpass(rfreq, h, 32,
                     NSP_WinBlackmanOpt, 1);

   /* filtering                              */
   nspdFirInit(h, 32, NULL, &firSt);
   nspdbFir(&firSt, input, output, 2000);

   /* now output contains the result of filtering */

   nspFirFree(&firSt);
```

## FirHighpass

*Computes taps for a highpass FIR filter.*

```
int nspdFirHighpass (double rfreq, double *taps, int tapsLen,
    NSP_WindowType winType, int doNormal);
    /* real values; double precision */
```

| | |
|---|---|
| *rfreq* | Cut-off frequency value (0 < *rfreq* < 0.5). |
| *taps* | Pointer to the vector of taps to be computed. |
| *tapsLen* | The number of taps in the *taps* array (*tapsLen* ≥ 5). |
| *winType* | Specifies the smoothing window type. Can be one of the following: |

| | |
|---|---|
| NSP_WinRect | no smoothing (rectangular window); |
| NSP_WinBartlett | smoothing by Bartlett window; |
| NSP_WinBlackmanOpt | smoothing by optimal Blackman window; |
| NSP_WinHamming | smoothing by Hamming window; |
| NSP_WinHann | smoothing by Hann window. |

| | |
|---|---|
| *doNormal* | Sets the normalization mode. If *doNormal* = 0, the function computes taps without normalization; otherwise, the taps will be normalized. |

### Discussion.

This function computes the taps for a highpass FIR filter. It forms an ideal (infinite) impulse response of a highpass filter and smoothes the response data by the specified window. If *doNormal* is non-zero, the function normalizes the taps. (For normalized taps, the response is 1 at the frequency equal to 0.5.)

The function returns one of the following values:

| | |
|---|---|
| NSP_fStsOk | if the taps have been computed successfully |
| NSP_fStsBadPointer | if the pointer to the *taps* vector is null |
| NSP_fStsBadLen | if the number of taps (*tapsLen*) is less than five |
| NSP_fStsBadFreq | if the frequencies are not in the interval (0, 0.5) |

Example 8-5 illustrates the usage of the `nspdFirHighpass()` function.

**Example 8-5 Using nspdFirHighpass to design a highpass FIR filter**

```
/* using the highpass FIR filter design function */

   NSPFirState firSt;
   double rfreq = 0.3; /* cut-off frequency  */
   double h[32];
   double input[2000];
   double output[2000];

   /* insert code here to initialize input   */

   /* design highpass filter                 */
   nspdFirHighpass(rfreq, h, 32,
                      NSP_WinBlackmanOpt, 1);

   /* filtering                              */
   nspdFirInit(h, 32, NULL, &firSt);
   nspdbFir(&firSt, input, output, 2000);

   /* now output contains the result of filtering */

   nspFirFree(&firSt);
```

## FirBandpass

*Computes taps for a bandpass FIR filter.*

```
int nspdFirBandpass (double rLowFreq, double rHighFreq,
   double *taps, int tapsLen, NSP_WindowType winType, int doNormal);
   /* real values; double precision */
```

| | |
|---|---|
| *rLowFreq* | Low cut-off frequency (0 < *rLowFreq* < *rHighFreq*). |
| *rHighFreq* | High cut-off frequency (*rLowFreq* < *rHighFreq* < 0.5) |
| *taps* | Pointer to the vector of taps to be computed. |
| *tapsLen* | The number of taps in *taps* (*tapsLen* ≥ 5). |
| *winType* | Specifies the smoothing window type. Can be one of the following: |

| | |
|---|---|
| NSP_WinRect | no smoothing (rectangular window); |
| NSP_WinBartlett | smoothing by Bartlett window; |
| NSP_WinBlackmanOpt | smoothing by optimal Blackman window; |
| NSP_WinHamming | smoothing by Hamming window; |
| NSP_WinHann | smoothing by Hann window. |

| | |
|---|---|
| *doNormal* | Sets the normalization mode. If *doNormal* = 0, the function computes taps without normalization; otherwise, the taps will be normalized. |

### Discussion.

This function computes the taps for a bandpass FIR filter. It forms an ideal (infinite) impulse response of bandpass filter and smoothes the response data by the specified window. If *doNormal* is non-zero, the function normalizes the taps. (For normalized taps, the response is 1 in the middle of the frequency interval, at (1/2)*(*rLowFreq* + *rHighFreq*).)

The function returns one of the following values:

| | |
|---|---|
| NSP_fStsOk | if the taps have been computed successfully |
| NSP_fStsBadPointer | if the pointer to the *taps* vector is null |
| NSP_fStsBadLen | if the number of taps (*tapsLen*) is less than five |
| NSP_fStsBadFreq | if the frequencies are not in the interval (0, 0.5) |
| NSP_fStsBadRel | if *rLowFreq* ≥ *rHighFreq* . |

Example 8-6 illustrates the usage of the `nspdFirHighpass()` function.

**Example 8-6 Using nspdFirBandpass to design a bandpass FIR filter**

```
/*  using the bandpass FIR filter design function */

   NSPFirState firSt;
   double rfreq_low=0.1; /* low cut-off frequency  */
   double rfreq_high=0.3;/* high cut-off frequency */
   double h[32];
   double input[2000];
   double output[2000];

   /* insert code here to initialize input   */

   /* design bandpass filter                 */
   nspdFirBandpass(rfreq_low, rfreq_high, h, 32,
                      NSP_WinBlackmanOpt, 1);

   /* filtering                              */
   nspdFirInit(h, 32, NULL, &firSt);
   nspdbFir(&firSt, input, output, 2000);

   /* now output contains the result of filtering */

   nspFirFree(&firSt);
```

## FirBandstop

*Computes taps for a bandstop FIR filter.*

```
int nspdFirBandstop (double rLowFreq, double rHighFreq,
   double *taps, int tapsLen, NSP_WindowType winType, int doNormal);
   /* real values; double precision */
```

| | |
|---|---|
| *7rLowFreq* | Low cut-off frequency (0 < *rLowFreq* < *rHighFreq*). |
| *rHighFreq* | High cut-off frequency (*rLowFreq* < *rHighFreq* < 0.5) |
| *taps* | Pointer to the vector of taps to be computed. |
| *tapsLen* | The number of taps in *taps* (*tapsLen* ≥ 5). |
| *winType* | Specifies the smoothing window type. Can be one of the following: |

| | |
|---|---|
| NSP_WinRect | no smoothing (rectangular window); |
| NSP_WinBartlett | smoothing by Bartlett window; |
| NSP_WinBlackmanOpt | smoothing by optimal Blackman window; |
| NSP_WinHamming | smoothing by Hamming window; |
| NSP_WinHann | smoothing by Hann window. |

| | |
|---|---|
| *doNormal* | Sets the normalization mode. If *doNormal* = 0, the function computes taps without normalization; otherwise, the taps will be normalized. |

### Discussion.

This function computes the taps for a bandstop FIR filter. It forms an ideal (infinite) impulse response of bandstop filter and smoothes the response data by the specified window. If *doNormal* is non-zero, the function normalizes the taps. (For normalized taps, the response is 1 near the zero frequency.)

The function returns one of the following values:

| | |
|---|---|
| NSP_fStsOk | if the taps have been computed successfully |
| NSP_fStsBadPointer | if the pointer to the *taps* vector is null |
| NSP_fStsBadLen | if the number of taps (*tapsLen*) is less than five |
| NSP_fStsBadFreq | if the frequencies are not in the interval (0, 0.5) |
| NSP_fStsBadRel | if *rLowFreq* ≥ *rHighFreq* . |

## Low-Level LMS Filter Functions

This section describes the low-level adaptive finite impulse response (FIR) filter functions. These filter functions employ the least mean squares (LMS) adaptation. The functions initialize the filter, get and set its taps and delay line, and perform the filter function. Unlike the FIR filters (whose filter coefficients do not vary over time) an adaptive filter varies its coefficients to try to make its output match some prototype “desired” signal as closely as possible. The low-level LMS functions maintain the filter coefficients separately from the delay line, allowing multiple delay lines to be used with the same set of taps. The low-level LMS functions do not allocate memory dynamically.

To use a low-level LMS filter, follow this general scheme:

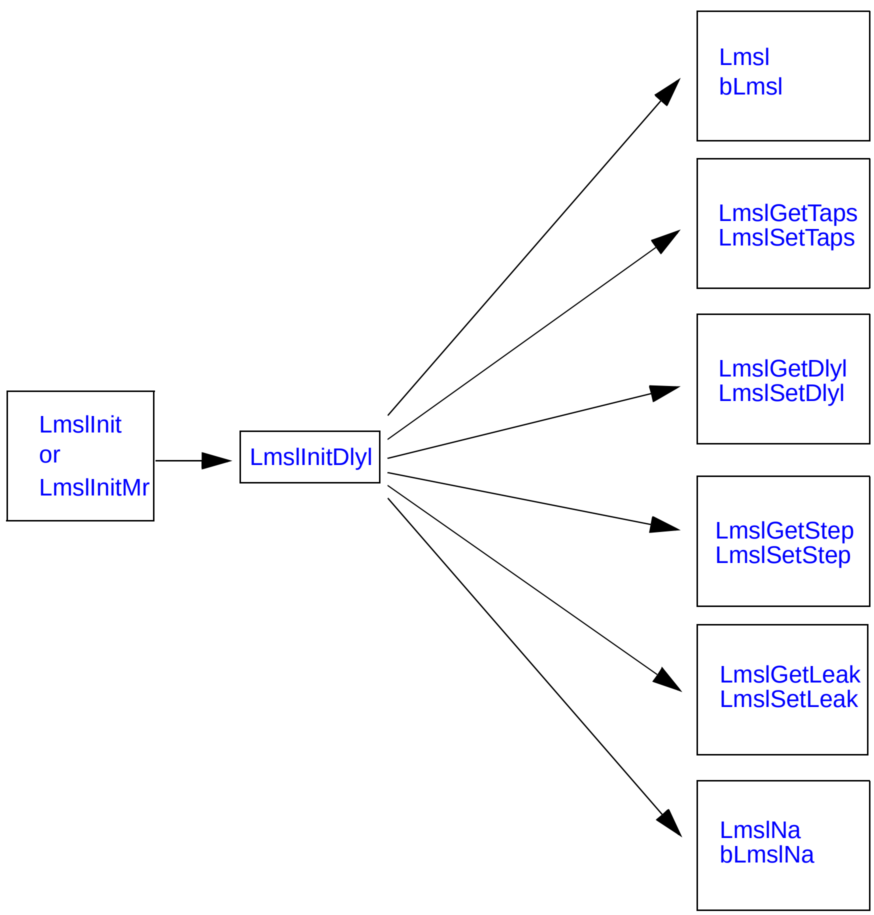
1. Call either `nsp?LmslInit()` to initialize the coefficients and structure of a single-rate filter or call `nsp?LmslInitMr()` to initialize the coefficients and structure of a multi-rate filter.
2. Call `nsp?LmslInitDlyl()` to initialize a delay line.

   The delay line is associated with a particular set of taps. Multiple delay lines for a given set of taps can be initialized by calling this function multiple times, but there should be only one call for each delay line.
3. After this initialization, you now have a choice of functions to call, depending on what you want to accomplish.
   a. Call the `nsp?Lmsl()` function to filter a single sample through a single-rate filter and/or call `nsp?bLmsl()` to filter a block of consecutive samples through a single-rate or multi-rate filter.
   b. Call the `nsp?LmslGetTaps()` function and the `nsp?LmslSetTaps()` function to get and set the filter coefficients (taps).
   c. Call the `nsp?LmslGetDlyl()` function and the `nsp?LmslSetDlyl()` function to get and set the values in the delay line.
   d. Call the `nsp?LmslGetStep()` function and the `nsp?LmslSetStep()` function to get and set the convergence step size values.
   e. Call the `nsp?LmslGetLeak()` function and the `nsp?LmslSetLeak()` function to get and set the leak values.

f. Call the functions `nsp?LmslNa()` or `nsp?bLmslNa()` to allow a second signal to be filtered independent of the first signal driving the adaptation. (That is, a second signal is filtered without the adaptation which is being applied to the first signal.)

Figure 8-3 illustrates the order of use of the low-level LMS filter functions.

**Figure 8-3 Order of Use of the Low-Level LMS Functions**



## LmslInit, LmslInitMr, LmslInitDlyl

*Initializes a low-level FIR filter that uses the least mean square (LMS) adaptation.*

```
void nspsLmslInit(NSPLmsType lmsType, float *taps,
      int tapsLen, float step, float leak, int errDly,
      NSPLmsTapState *tapStPtr);
void nspsLmslInitMr(NSPLmsType lmsType, float *taps, int tapsLen,
      float step, float leak, int errDly, int downFactor,
      int downPhase, NSPLmsTapState *tapStPtr);
void nspsLmslInitDlyl(NSPLmsTapState *tapStPtr, float *dlyl,
      int adaptB, NSPLmsDlyState *dlyStPtr);
      /* real values; single precision */

void nspcLmslInit(NSPLmsType lmsType, SCplx *taps, int tapsLen,
      float step, float leak, int errDly, NSPLmsTapState *tapStPtr);
void nspcLmslInitMr(NSPLmsType lmsType, SCplx *taps, int tapsLen,
      float step, float leak, int errDly, int downFactor,
      int downPhase, NSPLmsTapState *tapStPtr);
void nspcLmslInitDlyl(NSPLmsTapState *tapStPtr, SCplx *dlyl,
      int adaptB, NSPLmsDlyState *dlyStPtr);
      /* complex values; single precision */

void nspdLmslInit(NSPLmsType lmsType, double *taps, int tapsLen,
      float step, float leak, int errDly, NSPLmsTapState *tapStPtr);
void nspdLmslInitMr(NSPLmsType lmsType, double *taps, int tapsLen,
      float step, float leak, int errDly, int downFactor,
      int downPhase, NSPLmsTapState *tapStPtr);
void nspdLmslInitDlyl(NSPLmsTapState *tapStPtr, double *dlyl,
      int adaptB, NSPLmsDlyState *dlyStPtr);
      /* real values; double precision */

void nspzLmslInit(NSPLmsType lmsType, DCplx *taps, int tapsLen,
      float step, float leak, int errDly, NSPLmsTapState *tapStPtr);
void nspzLmslInitMr(NSPLmsType lmsType, DCplx *taps, int tapsLen,
      float step, float leak, int errDly, int downFactor,
      int downPhase, NSPLmsTapState *tapStPtr);
```

```
void nspzLmslInitDlyl(NSPLmsTapState *tapStPtr, DCplx *dlyl,
      int adaptB, NSPLmsDlyState *dlyStPtr);
      /* complex values; double precision */

void nspwLmslInit(NSPLmsType lmsType, float *taps, int tapsLen,
      float step, int errDly, NSPWLmsTapState *tapStPtr);
void nspwLmslInitDlyl(NSPWLmsTapState *tapStPtr, short *dlyl,
      NSPWLmsDlyState *dlyStPtr); /* short values */
```

| | |
|---|---|
| *adaptB* | Indicates whether the delay line will be used to adapt the LMS filter (that is, whether `nsp?Lmsl()` or `nsp?LmslNa()` will be called). The values for `adaptB` are `TRUE` or `FALSE`. This argument is used by the `nsp?LmslInitDlyl()` function. For integer LMS filters, `adaptB` is not used. In these filters, the taps are always re-calculated. |
| *dlyl* | Pointer to the array storing the delay line values for the `nsp?LmslInit()` and `nsp?LmslInitMr()` functions. |
| *dlylStPtr* | Pointer to the `NSPLmsDlylState` structure. This pointer is used by the `nsp?LmslInitDlyl()` function. |
| *lmsType* | The type of least mean square algorithm used. Currently, this must always be `NSP_LmsDefault`. |
| *downFactor* | For multi-rate filters. The factor by which the signal is down-sampled. That is, `downFactor` - 1 samples are discarded from the signal. This argument is used by the `nsp?LmslInitMr()` function. |
| *downPhase* | For multi-rate filters. A parameter that determines which of the samples within each block are not discarded. The value of `downPhase` is required to be $0 \le$ *downPhase* $<$ *downFactor*. This argument is used by the `nsp?LmslInitMr()` function. |
| *errDly* | The delay (in samples) from the output signal of the LMS filter to the error signal input. The *errDly* argument is used by the `nsp?LmslInit()` and `nsp?LmslInitMr()` functions. |

| | |
|---|---|
| `leak` | How much the tap values "leak" towards 0.0 on each iteration. The value must be between 0.0 and 1.0. The `leak` argument is used by the `nsp?LmslInit()` and `nsp?LmslInitMr()` functions. For integer LMS filters, the `leak` argument is not used/specified. |
| `step` | The convergence step size. This value must be between 0.0 and 1.0. A non-zero value enables adaptation while a value of 0.0 disables the filter adaptation. The `step` argument is used by the `nsp?LmslInit()` and `nsp?LmslInitMr()` functions. |
| `taps` | Pointer to the array storing the filter coefficient values for the `nsp?LmslInit()` and `nsp?LmslInitMr()` functions. Note that real values of `taps` are specified for integer LMS filters; the internal representation of `taps` and computation in the corresponding functions use fixed-point format. |
| `tapsLen` | The number of taps values in the `taps[n]` array. |
| `tapStPtr` | Pointer to the `NSPLmsTapState` structure. The `tapStPtr` pointer is used by the `nsp?LmslInit()` and `nsp?LmslInitMr()` functions. |

## Discussion

**`nsp?LmslInit()`.** The `nsp?LmslInit()` function initializes `tapStPtr` to describe a single-rate LMS filter. The argument `taps` is a pointer to an array of filter coefficient values and `tapsLen` is the length of the array. The array defines the signal $h_0(k)$. The filter taps are considered to be a signal $h_n(k)$, where $n$ indexes the evolution over time (that is, updating the taps corresponds to increasing $n$), and where $k$ indexes the taps from 0 to `tapsLen` - 1 for a fixed point in time.

For example, at time = 0, the taps are denoted as

$h_0(0)$, $h_0(1)$, . . . $h_0(k)$.

At time = 1, the taps are denoted as

$h_1(0)$, $h_1(1)$, . . . $h_1(k)$.

The argument `leak` controls how much the tap values "leak" towards 0 on each iteration. Its value should be between 0.0 and 1.0. A value of 0.0 yields a traditional non-leaky LMS filter; a typical leaky filter uses a small value for `leak`.

The argument `errDly` specifies the delay (in samples) from the output signal of the LMS filter to the error signal input. In principle, the value of `errDly` must be at least 1. A delay of 1 corresponds to subtracting the filter output sample `y(n)` from the desired signal `d(n)` to obtain the error signal `e(n)`. The value of the error signal is then passed to the next invocation of the filter. In the general case, `errDly` is chosen as follows:

`e(n)` = `d(n - errDly)` - `y(n - errDly)`

where `e(n)` is the error signal. See "Lmsl" (for `nsp?Lmsl()`) on page 8-58 for a description of how `e(n)` is used.

**`nsp?LmslInitMr()`.** The `nsp?LmslInitMr()` function initializes `tapStPtr` to describe a multi-rate filter; that is, a filter which internally down-samples using a polyphase filter structure. As discussed in Appendix C, only down-sampling is supported for the LMS filters as opposed to both up-sampling and down-sampling. The argument `taps` is a pointer to an array of filter coefficient values and `tapsLen` is the length of the array. The array defines the signal $h_0(k)$. The filter taps are considered to be a signal $h_n(k)$, where $n$ indexes the evolution over time (that is, updating the taps corresponds to increasing $n$), and where $k$ indexes the taps from 0 to `tapsLen` - 1 for a fixed point in time.

The argument `downFactor` is the factor by which the signal is down-sampled. That is, `downFactor` - 1 samples are discarded from the signal. The argument `downPhase` determines which of the samples within each block are not discarded. The value of `downPhase` is required to be 0 ≤ `downPhase` < `downFactor`. For more information on down-sampling, see "DownSample" in Chapter 3 for the description of the `nsp?DownSample` function.

The arguments `step`, `leak`, and `errDly` are the same as described for the `nsp?LmslInit()` function above. Note that `errDly` is relative to the output rate, not the input rate.

Short integer LMS filters do not have a multi-rate filter mode; accordingly, there is no initialization function for this mode.

**nsp?LmslInitDlyl().** The `nsp?LmslInitDlyl()` function initializes *`dlyStPtr`* to describe a delay line. The tap state *`tapStPtr`* must have been previously initialized by either `nsp?LmslInit()` or `nsp?LmslInitMr()`.

The argument *`adaptB`* specifies whether the delay line will be used to adapt the LMS filter (that is, whether `nsp?Lmsl()` or `nsp?LmslNa()` will be called). This is important because a delay line used for adaptation requires more previous samples than otherwise. If *`adaptB`* is `TRUE`, the delay line will be used for adaptation, and the array *`dlyl[n]`* must be *`tapsLen`* + *`errDly`* long. If *`adaptB`* is `FALSE`, the array *`dlyl[n]`* must be *`tapsLen`* long. Only the first *`tapsLen`* + *`errDly`* - 1 (or *`tapsLen`* - 1 for non-adapting) samples provide initial values. For multi-rate filters, the delay line length is greater by the *`downFactor`* value. The *`adaptB`* argument is used by the `nsp?Lmsl()` function only to determine the correct length of the delay line; the *`adaptB`* argument does not determine whether adaptation is done. To disable adaptation, your application should set *`step`* to 0.

The *`step`* and *`leak`* parameters are single precision (`float`) for all types of the `nsp?LmslInit()` and `nsp?LmslInitMr()` functions defined above. These parameters do not require the extra precision available with double precision (`double`).

Do not deallocate or overwrite the arrays *`taps[n]`* and *`dlyl[n]`* during the life of the filter. Your application must not directly access these arrays because the `nsp?LmslInit()`, `nsp?LmslInitMr()` and `nsp?LmslInitDlyl()`functions can permute their contents in an implementation-dependent way.

**Application Notes:** The taps array *`taps[n]`* and delay line array *`dlyl[n]`* can be permuted as described for the FIR filters. See the "Low-Level FIR Filter Functions" for details. The permuted order used by the LMS functions is implementation-dependent and might or might not be the same as that used by the FIR functions.

### Related Topics

| | |
|---|---|
| `bLmsl` | Filters samples using a low-level, multi-rate, adaptive LMS filter to produce a single sample (see page 8-58). |

| | |
|---|---|
| `bLmslNa` | Filters a block of signals using a low-level, adaptive LMS filter but does not adapt the filter for a secondary signal (see page 8-64). |
| `Lmsl` | Filters a single sample using a low-level, single-rate, adaptive LMS filter (see page 8-58). |
| `LmslGetLeak` | Gets the leak values for a low-level LMS filter (see page 8-57). |
| `LmslGetStep` | Gets the step values for a low-level LMS filter (see page 8-57). |
| `LmslNa` | Filters a signal using a low-level, adaptive LMS filter but does not adapt the filter for a secondary signal (see page 8-64). |
| `LmslSetLeak` | Sets the leak values for a low-level LMS filter (see page 8-57). |
| `LmslSetStep` | Sets the step values for a low-level LMS filter (see page 8-57). |

## LmslGetTaps, LmslSetTaps

*Gets and sets the tap coefficients of low-level LMS filters.*

```
void nspsLmslGetTaps(const NSPLmsTapState *tapStPtr,
      float *outTaps);
void nspsLmslSetTaps(const float *inTaps, NSPLmsTapState *tapStPtr);
      /* real values; single precision */

void nspcLmslGetTaps(const NSPLmsTapState *tapStPtr,
      SCplx *outTaps);
void nspcLmslSetTaps(const SCplx *inTaps, NSPLmsTapState *tapStPtr);
      /* complex values; single precision */

void nspdLmslGetTaps(const NSPLmsTapState *tapStPtr,
      double *outTaps);
```

```
void nspdLmslSetTaps(const double *inTaps,
      NSPLmsTapState *tapStPtr);
      /* real values; double precision */

void nspzLmslGetTaps(const NSPLmsTapState *tapStPtr,
      DCplx *outTaps);
void nspzLmslSetTaps(const DCplx *inTaps, NSPLmsTapState *tapStPtr);
      /* complex values; double precision */

void nspwLmslGetTaps(const NSPWLmsTapState *tapStPtr,
      float *outTaps);
void nspwLmslSetTaps(const float *inTaps, NSPWLmsTapState
*tapStPtr); /* short values */
```

| | |
|---|---|
| *inTaps* | Pointer to the array holding copies of the tap coefficients for the nsp?LmslSetTaps() function. |
| *outTaps* | Pointer to the array holding copies of the tap coefficients for the nsp?LmslGetTaps() function. |
| *tapStPtr* | Pointer to the NSPLmsTapState structure. |

## Discussion

The nsp?LmslGetTaps() and nsp?LmslSetTaps() functions provide a safe mechanism to get and set the taps of a low-level LMS filter. Because the taps may be stored in permuted order, it is not safe for the application to directly access the tap array. Instead, nsp?LmslGetTaps() and nsp?LmslSetTaps() should be used. Note that *real* values of taps are specified for integer LMS filters; the internal representation of taps and computation in the corresponding functions use fixed-point format.

**Previous Tasks:** Before calling either the nsp?LmslGetTaps() or the nsp?LmslSetTaps()function, you must initialize the filter tap state *tapStPtr* by calling either nsp?LmslInit() or nsp?LmslInitMr(). This references the (permuted) tap array *taps[n]* and the filter length *tapsLen*. The data type used during initialization must match the data type used here.

**nsp?LmslGetTaps().** The function nsp?LmslGetTaps() copies the tap coefficients from *taps[n]* to the *tapsLen* length array *outTaps[n]*, unpermuting them if required so that *outTaps*[*n*]=*h(n)*.

**nsp?LmslSetTaps().** The function `nsp?LmslSetTaps()` copies the *tapsLen* tap coefficients from the *inTaps[n]* array into *taps[n]*, permuting them if required.

**Application Notes:** The `nsp?LmslGetTaps()` and `nsp?LmslSetTaps()` functions can be used to permute or unpermute an LMS filter's taps in-place or not-in-place. That is, if the pointer *inTaps* points to an array other than *taps[n]* (for `nsp?LmslSetTaps()`), or if *outTaps* points to an array other than *taps[n]* (for `nsp?LmslGetTaps()`), then the permutation is performed not-in-place.

If, on the other hand, *inTaps* or *outTaps* points to the same array, *taps[n]*, then the permutation is performed in-place. You might want your application to do this to avoid allocating a separate array to hold the permuted values. However, if your application unpermutes the *taps[n]* array in-place (via `nsp?LmslGetTaps()`), the *taps[n]* array must be re-permuted (via `nsp?LmslSetTaps()`) before the filter can be used again. Thus, you must use caution when permuting in-place.

### Related Topics

| | |
|---|---|
| `LmslInit` | Initializes a low-level, single-rate LMS filter (see page 8-47). |
| `LmslInitMr` | Initializes a low-level, multi-rate LMS filter (see page 8-47). |

## LmslGetDlyl, LmslSetDlyl

*Gets and sets the delay line contents of low-level LMS filters.*

```
void nspsLmslGetDlyl(const NSPLmsTapState *tapStPtr,
      const NSPLmsDlyState *dlyStPtr, float *outDlyl);
void nspsLmslSetDlyl(const NSPLmsTapState *tapStPtr,
      const float *inDlyl, NSPLmsDlyState *dlyStPtr);
      /* real values; single precision */
```

```
void nspcLmslGetDlyl(const NSPLmsTapState *tapStPtr,
      const NSPLmsDlyState *dlyStPtr, SCplx *outDlyl);
void nspcLmslSetDlyl(const NSPLmsTapState *tapStPtr,
      const SCplx *inDlyl, NSPLmsDlyState *dlyStPtr);
      /* complex values; single precision */

void nspdLmslGetDlyl(const NSPLmsTapState *tapStPtr,
      const NSPLmsDlyState *dlyStPtr, double *outDlyl);
void nspdLmslSetDlyl(const NSPLmsTapState *tapStPtr,
      const double *inDlyl, NSPLmsDlyState *dlyStPtr);
      /* real values; double precision */

void nspzLmslGetDlyl(const NSPLmsTapState *tapStPtr,
      const NSPLmsDlyState *dlyStPtr, DCplx *outDlyl);
void nspzLmslSetDlyl(const NSPLmsTapState *tapStPtr,
      const DCplx *inDlyl, NSPLmsDlyState *dlyStPtr);
      /* complex values; double precision */

void nspwLmslGetDlyl(const NSPWLmsTapState *tapStPtr,
      const NSPWLmsDlyState *dlyStPtr, short *outDlyl);
void nspwLmslSetDlyl(const NSPWLmsTapState *tapStPtr,
      const short *inDlyl, NSPWLmsDlyState *dlyStPtr);
      /* short values */
```

| | |
|---|---|
| *dlyStPtr* | Pointer to the `NSPLmsDlyState` structure. |
| *inDlyl* | Pointer to the array holding copies of the delay line values for the `nsp?LmslSetDlyl()` function. |
| *outDlyl* | Pointer to the array holding copies of the delay line values for the `nsp?LmslGetDlyl()` function. |
| *tapStPtr* | Pointer to the `NSPLmsTapState` structure. |

## Discussion

The `nsp?LmslGetDlyl()` and `nsp?LmslSetDlyl()` functions provide a safe mechanism to get and set the delay line of a low-level LMS filter. Because the delay line may be stored in permuted order, it is not safe for the application to directly access the delay line array. Instead, `nsp?LmslGetDlyl()` and `nsp?LmslSetDlyl()` should be used. The data type used for `nsp?LmslGetDlyl()` and `nsp?LmslSetDlyl()` must match the data type of the delay line initialization (and not the data type of the tap initialization). For more information on initializing delay lines, see "LmslInitDlyl" on page 8-47 for `nsp?LmslInitDlyl()`.

**Previous Tasks:** Before calling either `nsp?LmslGetDlyl()` or `nsp?LmslSetDlyl()`, you must initialize the filter tap state pointed to by *`tapStPtr`*, the (permuted) tap array *`taps`* and the filter length *`tapsLen`* by calling either `nsp?LmslInit()` or `nsp?LmslInitMr()`. In addition, you must initialize the delay line state pointed to by *`dlyStPtr`* and the (permuted) delay line array *`dlyl[n]`* by calling `nsp?LmslInitDlyl()`. Both `nsp?LmslGetDlyl()` and `nsp?LmslSetDlyl()` require *`tapStPtr`* as an argument to describe the delay line permutation.

**`nsp?LmslGetDlyl()`.** The `nsp?LmslGetDlyl()` function unpermutes the delay line values in *`dlyl[n]`* and stores them into the array *`outDlyl[n]`* so that *`outDlyl[k]=x(n-k)`*, where *`x(n)`* was the last sample that was filtered. For single-rate filters, *`outDlyl[n]`* must be *`tapsLen`* + *`errDly`* -1 long if the delay line is used for adaptation, and *`tapsLen`* - 1 long otherwise. For multi-rate filters an additional *`downFactor`* samples are required in *`outDlyl[n]`*.

**`nsp?LmslSetDlyl()`.** The `nsp?LmslSetDlyl()` function permutes the values in the array *`inDlyl[n]`*, stores them into *`dlyl[n]`*, and updates *`dlyStPtr`*. For single-rate filters, *`inDlyl[n]`* must be *`tapsLen`* + *`errDly`* - 1 long if the delay line is used for adaptation and *`tapsLen`* - 1 long otherwise. For multi-rate filters an additional *`downFactor`* samples are required in *`inDlyl[n]`*. If *`inDlyl`* is `NULL`, the delay line is initialized to all zeros.

**Application Notes:** The `nsp?LmslGetDlyl()` and `nsp?LmslSetDlyl()` functions can be used to permute or unpermute an LMS filter's taps in-place or not-in-place. That is, if the pointer *`inDlyl`* points to an array other than *`dlyl[n]`* (for `nsp?LmslSetDlyl()`), or if *`outDlyl`* points to an array other than *`dlyl[n]`* (for `nsp?LmslGetDlyl()`), then the permutation is performed not-in-place.

If, on the other hand, *`inDlyl`* or *`outDlyl`* points to the same array, *`dlyl[n]`*, then the permutation is performed in-place. You might want your application to do this to avoid allocating a separate array to hold the permuted values. However, if your application unpermutes the *`dlyl[n]`* array in-place (via `nsp?LmslGetDlyl()`), the *`dlyl[n]`* array must be re-permuted (via `nsp?LmslSetDlyl()`) before the filter can be used again. Thus, you must use caution when permuting in-place.

### Related Topics

| | |
|---|---|
| `LmslInit` | Initializes a low-level, single-rate LMS filter (see page 8-47). |
| `LmslInitDlyl` | Initializes the delay line values for an LMS filter (see page 8-47). |
| `LmslInitMr` | Initializes a low-level, multi-rate LMS filter (see page 8-47). |

## LmslGetStep, LmslSetStep, LmslGetLeak, LmslSetLeak

*Gets and sets the leak and step values of a low-level LMS filter.*

```
float nspsLmslGetStep(const NSPLmsTapState *statePtr);
void nspsLmslSetStep(float step, NSPLmsTapState *statePtr);
      /* real values; single precision */

float nspsLmslGetLeak(const NSPLmsTapState *statePtr);
void nspsLmslSetLeak(float leak, NSPLmsTapState *statePtr);
      /* real values; single precision */

float nspwLmslGetStep(const NSPWLmsTapState *statePtr);
void nspwLmslSetStep(float step, NSPWLmsTapState *statePtr);
       /* short values */
```

| | |
|---|---|
| *`leak`* | How much the tap values “leak” towards 0.0 on each iteration. The value must be between 0.0 and 1.0. |
| *`statePtr`* | Pointer to the `NSPLmsTapState` structure. |
| *`step`* | The convergence step size. This value must be between 0.0 and 1.0. |

### Discussion

The `nsp?LmslGetLeak()` and `nsp?LmslSetLeak()` functions allow your application to get and set the *`leak`* parameter of a low-level LMS filter described by *`statePtr`*.

The `nsp?LmslGetStep()` and `nsp?LmslSetStep()` functions allow your application to get and set the *`step`* parameter of a low-level LMS filter described by *`statePtr`*.

Only the single-precision *`step`* and *`leak`* parameters are supported. For integer low-level LMS filters, the real floating-point value of *`step`* is converted to the fixed-point format in the `SetStep()` function; fixed-to-float conversion is performed in `GetStep()`. Therefore, you should not directly manipulate the step field in a structure specifying the filter. The error in the fixed-point data representation in LMS filters is about $10^{-4}$ (for a given number of bits on the right of the fixed point in signed 32-bit words). The data range is approximately $-10^{-6}$ to $10^{-6}$.

Integer low-level LMS filters do not use the *`leak`* parameter; there are no `SetLeak` and `GetLeak` functions for these filters.

### Related Topics

| | |
|---|---|
| `LmslInit` | Initializes a low-level, single-rate LMS filter (see page 8-47). |
| `LmslInitMr` | Initializes a low-level, multi-rate LMS filter (see page 8-47). |

## Lmsl, bLmsl

*Filters samples through a low-level LMS adaptation filter.*

```
float nspsLmsl(NSPLmsTapState *tapStPtr, NSPLmsDlyState *dlyStPtr,
      float samp, float err);
```

```
float nspsbLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  const float *inSamps, float err);
      /* real input, real taps; single precision */

SCplx nspcLmsl(NSPLmsTapState *tapStPtr, NSPLmsDlyState *dlyStPtr,
      SCplx samp, SCplx err);
SCplx nspcbLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  const SCplx *inSamps, SCplx err);
      /* complex input, complex taps; single precision */

SCplx nspscLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  float samp, SCplx err);
SCplx nspscbLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const float *inSamps, SCplx err);
      /* real input, complex taps; single precision */

double nspdLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  double samp, double err);
double nspdbLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const double *inSamps, double err);
      /* real input, real taps; double precision */

DCplx nspzLmsl(NSPLmsTapState *tapStPtr, NSPLmsDlyState *dlyStPtr,
      DCplx samp, DCplx err);
DCplx nspzbLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  const DCplx *inSamps, DCplx err);
      /* complex input, complex taps; double precision */

DCplx nspdzLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  double samp, DCplx err);
DCplx nspdzbLmsl(NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const double *inSamps, DCplx err);
      /* real input, complex taps; double precision */

short nspwLmsl(NSPWLmsTapState *tapStPtr, NSPWLmsDlyState *dlyStPtr,
      short samp, short err);  /* short values */
```

| | |
|---|---|
| *dlyStPtr* | Pointer to the `NSPLmsDlyState` structure. |
| *err* | The error signal sample. |
| *inSamps* | Pointer to the array containing the input samples for the `nsp?bLmsl()` function. |

*samp* The input sample for the `nsp?Lmsl()` function.

*tapStPtr* Pointer to the `NSPLmsTapState` structure.

## Discussion

The `nsp?Lmsl()` and `nsp?bLmsl()` functions perform a single iteration of LMS adaptation and filtering. The `nsp?Lmsl()` function filters a sample through a single-rate filter and the `nsp?bLmsl()` function filters samples through a multi-rate filter to produce a single sample.

**Previous Tasks:** Before using either `nsp?Lmsl()` or `nsp?bLmsl()`, you must initialize the *tapStPtr* argument by calling either `nsp?LmslInit()` or `nsp?LmslInitMr()`, and the argument *dlyStPtr* by calling `nsp?LmslInitDlyl()` with *adaptB* = `TRUE`.

Many combinations of input (*x(n)*) types and filter coefficient types are possible. Real or complex input can be mixed with real or complex filter coefficients. This is indicated by the `s`, `c`, `sc`, `d`, `z`, and `dz` type codes following the `nsp` prefix in the function names above. For both of the functions, `nsp?Lmsl()` and `nsp?bLmsl()`, the allowed combinations of real and complex input and filter coefficients are described in Table 8-4.

The data type of the output is also described in this table. The data type of the error signal must match the data type of the output signal.

**Table 8-4 Input and Taps Combinations for nsp?Lmsl() and nsp?bLmsl() Functions**

| Type Codes | x(n) (or input) Type | Filter Coefficient (or taps) Type | y(n) (or output) Type |
|---|---|---|---|
| w | short | float/fixed | short |
| s | float | float | float |
| c | SCplx | SCplx | SCplx |
| sc | float | SCplx | SCplx |
| d | double | double | double |
| z | DCplx | DCplx | DCplx |
| dz | double | DCplx | DCplx |

**NOTE.** *The complex input, real tap types (`cs`, `zd`), are not provided. While it is possible to implement such constrained LMS filters by projecting the error term onto the real taps, the method of projection is application-dependent. In contrast, the real input, complex tap types (`sc`, `dz`), are provided, but might have convergence problems depending on the input signal.*

**`nsp?Lmsl()`.** The `nsp?Lmsl()` function filters a sample through a single-rate filter. In the function definition below, the input sample *`samp[n]`* is *`x(n)`*, *`err`* is the error sample *`e(n)`*, and *`y(n)`* is the returned sample: $e(n) = d(n - errDly) - y(n - errDly)$

$$h_n(k) = (1 - leak) \cdot h_{n-1}(k) + step \cdot e(n) \cdot x(n - k - errDly)^*,$$
$$0 \le k < tapsLen$$

$$y(n) = \sum_{k=0}^{tapsLen-1} h_n(k) \cdot x(n-k)$$

where $x(n\text{-}k\text{-}errDly)^*$ denotes the complex conjugate of $x(n\text{-}k\text{-}errDly)$ and the "·" operator denotes complex multiplication.

Integer LMS filters use the following formula for computing $h_n(k)$:

$$h_n(k) = h_{n-1}(k) + step \cdot e(n) \cdot x(n - k - errDly)^*,$$
$$0 \le k < tapsLen$$

The above formulations define the filter for all combinations of real and complex input and coefficients. If the input is real, the complex conjugation of *`x()`* in the coefficient update equation is not necessary.

An alternative formulation, found in many textbooks, gives identical results, but differs in how it defines *`e(n)`*. This alternative formulation is difficult to interpret when *`errDly`* > 1 and cannot be directly implemented because it requires that *`d(n)`* be available to the LMS filtering function. You should formulate the error sample calculation for your application in terms of the above set of equations.

**nsp?bLmsl().** The nsp?bLmsl() function filters samples through a multi-rate filter to produce a single sample. The argument *tapStPtr* uses the *downFactor* argument specified by nsp?LmslInitMr(). The argument *err* is the error signal *e(n)*, and the *downFactor* length array *inSamps[n]* provides samples of *x(n)*. The filtered result *y(n)* is returned. The sample rate of the input signal is greater than the sample rate of the output and error by a factor of *downFactor*.

Even though nsp?bLmsl() has the b prefix flag to indicate a blocked function, this function does not perform more than one iteration. This is because doing so would introduce excess delay into the error signal. Instead, this function is provided for multi-rate filtering, which requires a vector (blocked) input array.

Note that integer low-level LMS filters do have no multi-level filter mode.

Example 8-7 illustrates the use of the low-level LMS functions to initialize and filter a signal sample.

### Example 8-7 Filtering with the Low-Level LMS Filter

```
#define TAPSLEN  2
#define LEN   100
const float STEP = 0.6;
NSPLmsTapState tapStPtr;
NSPLmsDlyState dlyStPtr;
double x[LEN], d[LEN], y[LEN], z[LEN];
double h[TAPSLEN];
int n;
double err = 0;
```

continued ☛

**Example 8-7 Filtering with the Low-Level LMS Filter** (continued)

```
/*
 * Generate the input and desired signals.  The input
 * signal is a sine wave with amplitude 1.0 at 0.2 Fs.
 * The desired signal is a cosine wave with amplitude
 * 2.0 at the same frequency.
 */
for (n = 0; n < LEN; n++) {
      x[n] = sin(NSP_2PI*n/10);
      d[n] = 2*cos(NSP_2PI*n/10);
      z[n] = 0.0;
}
/*  Initialize taps values to zero */
for (n = 0; n < TAPSLEN; n++) {
      h[n] = 0.0;
}
/*  Initialize filter */
nspdLmslInit(NSP_LmsDefault, h, TAPSLEN, STEP, 0.0, 0, &tapStPtr);
 /* Initialize delay line */
nspdLmslInitDlyl(&tapStPtr, z, TRUE, &dlyStPtr);

/*  Filter LEN samples using single-rate adaptive filtering */
for (n = 0; n < LEN; n++) {
      y[n] = nspdLmsl(&tapStPtr, &dlyStPtr, x[n], err);
      err = d[n] - y[n];
} /* The final taps values = {2.75,-3.40} and
 * err = 0 are obtained
 */
```

## Related Topics

| | |
|---|---|
| `bLmslNa` | Filters a block of signals using a low-level, adaptive LMS filter but does not adapt the filter for a secondary signal (see page 8-64). |
| `LmslGetLeak` | Gets the leak values for a low-level LMS filter (see page 8-57). |
| `LmslGetStep` | Gets the step values for a low-level LMS filter (see page 8-57). |

| | |
|---|---|
| `LmslInitDlyl` | Initializes the delay line values for a low-level LMS filter (see page 8-47). |
| `LmslNa` | Filters a signal using a low-level, adaptive LMS filter but does not adapt the filter for a secondary signal (see page 8-64). |
| `LmslSetLeak` | Sets the leak values for a low-level LMS filter (see page 8-57). |
| `LmslSetStep` | Sets the step values for a low-level LMS filter (see page 8-57). |

## LmslNa, bLmslNa

*Filters a signal using a low-level adaptive LMS filter, but does not adapt the filter for a secondary signal.*

```
float nspsLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, float samp);
void nspsbLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const float *inSamps,
      float *outSamps, int numIters);
      /* real input, real taps; single precision */

SCplx nspcLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  SCplx samp);
void nspcbLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const SCplx *inSamps,
      SCplx *outSamps, int numIters);
      /* complex input, complex taps; single precision */

SCplx nspscLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  float samp);
void nspscbLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const float *inSamps,
      SCplx *outSamps, int numIters);
      /* real input, complex taps; single precision */
```

```
SCplx nspcsLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  SCplx samp);
void nspcsbLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const SCplx *inSamps,
      SCplx *outSamps, int numIters);
      /* complex input, real taps; single precision */

double nspdLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, double samp);
void nspdbLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  const double *inSamps,
      double *outSamps, int numIters);
      /* real input, real taps; double precision */

DCplx nspzLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  DCplx samp);
void nspzbLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const DCplx *inSamps,
      DCplx *outSamps, int numIters);
      /* complex input, complex taps; double precision */

DCplx nspdzLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  double samp);
void nspdzbLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const double *inSamps,
      DCplx *outSamps, int numIters);
      /* real input, complex taps; double precision */

DCplx nspzdLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr,  DCplx samp);
void nspzdbLmslNa(const NSPLmsTapState *tapStPtr,
      NSPLmsDlyState *dlyStPtr, const DCplx *inSamps,
      DCplx *outSamps, int numIters);
      /* complex input, real taps; double precision */
```

| | |
|---|---|
| *dlyStPtr* | Pointer to the `NSPLmsDlyState` structure. |
| *inSamps* | Pointer to the array containing the input samples for the `nsp?bLmslNa()` function. |
| *numIters* | The number of samples to be filtered by the `nsp?bLmsNa()` function. |

| | |
|---|---|
| *outSamps* | Pointer to the array containing the output samples for the `nsp?bLmslNa()` function. |
| *samp* | The input sample for the `nsp?LmslNa()` function. |
| *tapStPtr* | Pointer to the `NSPLmsTapState` structure. |

## Discussion

The `nsp?LmslNa()` and `nsp?bLmslNa()` functions allow a secondary signal, *w(n)*, to be filtered independently of the primary signal, *x(n)*, which drives the adaptation. The secondary signal must have its own delay line independent of the delay line used for the primary signal. The functions update the delay line for the secondary signal but not for the primary signal. The functions also filter the secondary signal by the same taps as used for the primary signal. The taps themselves are not modified.

The argument *tapStPtr* must have been previously initialized by `nsp?LmslInit()` or `nsp?LmslInitMr()`, and the pointer *dlyStPtr* must have been previously initialized by `nsp?LmslInitDlyl()` with *adaptB* = `FALSE`.

In terms of supported data types, the `nsp?LmslNa()` and `nsp?bLmslNa()` functions are less restrictive than `nsp?Lmsl()` and `nsp?bLmsl()`. In particular, all combinations of real and complex input data types and filter coefficients are supported. The complete blocked form is also supported. The data type codes following the `nsp` prefix in the function names are described in Table 8-5.

**Table 8-5 Input and Taps Combinations for nsp?LmslNa() and nsp?bLmslNa() Functions**

| Type Codes | x(n) (or input) Type | Filter Coefficient (or taps) Type | y(n) (or output) Type |
|---|---|---|---|
| `s` | `float` | `float` | `float` |
| `c` | `SCplx` | `SCplx` | `SCplx` |
| `sc` | `float` | `SCplx` | `SCplx` |
| `cs` | `SCplx` | `float` | `SCplx` |

continued ☞

**Table 8-5 Input and Taps Combinations for nsp?LmslNa() and nsp?bLmslNa() Functions** (continued)

| Type Codes | x(n) (or input) Type | Filter Coefficient (or taps) Type | y(n) (or output) Type |
|---|---|---|---|
| d | double | double | double |
| z | DCplx | DCplx | DCplx |
| dz | double | DCplx | DCplx |
| zd | DCplx | double | DCplx |

**nsp?LmslNa().** The `nsp?LmslNa()` function filters a single sample through a single-rate filter without adapting the filter. The argument *samp[n]* provides the sample signal, *w(n)*, and the result *v(n)* is returned. The result, *v(n)*, is defined as follows:

$$v(n) = \sum_{k=0}^{tapsLen-1} h_n(k) \cdot w(n-k)$$

**nsp?bLmslNa().** The `nsp?bLmslNa()` function filters a block of samples through a single-rate or multi-rate filter without adapting the filter. For single-rate filters, the *numIters* samples in the array *inSamps[n]* are filtered, and the resulting *numIters* samples are stored in the array *outSamps[n]*. The results are identical to *numIters* consecutive calls to `nsp?LmslNa()`. The values in the *outSamps[n]* array are calculated as follows:

$$inSamps[m] = x(n+m)$$

$$outSamps[m] = y(n+m) = \sum_{k=0}^{tapsLen-1} h(k) \cdot x(n+m-k)$$

For multi-rate filters, the *numIters *downFactor* samples in the array *inSamps[n]* are filtered, and the resulting *numIters* samples are stored in the array *outSamps[n]*. See Appendix C for more information on multi-rate filtering. For both single-rate and multi-rate, the appropriate number of samples from *inSamps[n]* are copied into the delay line, and the oldest samples are discarded.

Example 8-8 illustrates using the `nsp?LmslNa()` functions to filter a signal with adapted filter coefficients.

**Example 8-8 Creating a Filter Predictor with the LMS Filter Functions**

```
/* A filter-predictor using adaptive and
 nonadaptive filter functions */
#include <math.h>
#define nsp_UsesLms
#include "nsp.h"
#define HLEN (5)    /* taps number */
#define DLY  (1)   /* Prediction depth */

float
      y[100],       /* Output predicted signal */
      d[100],       /* Output signal of adaptive filter */
      z_a[HLEN+DLY], /* Delay line of adaptive filter */
      z_n[HLEN],    /* Delay line of filter-predictor */
      x[100],       /* Output signal */
      h[HLEN],      /* taps */
      err;          /* Adaptation error signal */

int main (void)
{
      int i;
      NSPLmsTapState tapst;
      NSPLmsDlyState dlyst_a,dlyst_n;

      /* Adaptive filter initialization */
      nspsLmslInit (NSP_LmsDefault, h,HLEN, 0.05,\
         0.0, DLY,&tapst);

      /* The adaptive filter delay line initialization */
      nspsLmslInitDlyl (&tapst, z_a, TRUE, &dlyst_a);

      /* The predictor delay line initialization */
      nspsLmslInitDlyl (&tapst, z_n, FALSE, &dlyst_n);
```

continued

**Example 8-8 Creating a Filter Predictor with the LMS Filter Functions** (continued)

```
/* Initial values of taps and
 * delay line are updated here
 */
      /* Generate model input signal */
      for(i=0;i<100;i++) x[i]=cos(NSP_2PI*i/16);
      err = 0;
      /* Taps adaptation */
      for(i=DLY; i<100; i++) {
         err = x[i] - (d[i]=nspsLmsl (&tapst, &dlyst_a,\
            x[i-DLY], err));
         /* the coefficients have now
          * been adapted using err
          */

         /* Signal prediction */
         y[i] = nspsLmslNa (&tapst, &dlyst_n, x[i]);
      }
      return 0;
}
```

**Application Notes:** The `nsp?LmslNa()` and `nsp?bLmslNa()` functions are intended only for filtering the secondary signal without adapting the coefficients. They should not be used to filter the primary signal without adapting the coefficients, because they manage the delay line in a manner incompatible with `nsp?Lmsl()`. To filter a primary signal without adaptation, use the `nsp?LmslSetStep()` function to set the *`step`* argument to zero, and then use the `nsp?Lmsl()` or `nsp?bLmsl()` function.

The `nsp?LmslNa()` and `nsp?bLmslNa()` functions do not support filtering of the primary signal *`x(n)`* because the primary signal requires a longer delay line.

## Related Topics

| | |
|---|---|
| `bLmsl` | Filters samples using a low-level, multi-rate, adaptive LMS filter to produce a single sample (see page 8-58). |

| | |
|---|---|
| `Lmsl` | Filters a single sample using a low-level, single-rate, adaptive LMS filter (see page 8-58). |
| `LmslGetStep` | Gets the step values for a low-level LMS filter (see page 8-57). |
| `LmslInit` | Initializes a low-level, single-rate LMS filter (see page 8-47). |
| `LmslInitDlyl` | Initializes the delay line contents for a low-level LMS filter (see page 8-47). |
| `LmslInitMr` | Initializes a low-level, multi-rate LMS filter (see page 8-47). |
| `LmslSetStep` | Sets the step values for a low-level LMS filter (see page 8-57). |

## LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize an LMS filter
- get and set the delay line values
- get and set the filter coefficients (taps) values
- get and set the step values
- get and set the leak values
- get and set the error signals
- compute error signals
- perform the filtering function
- free dynamic memory allocated for the functions

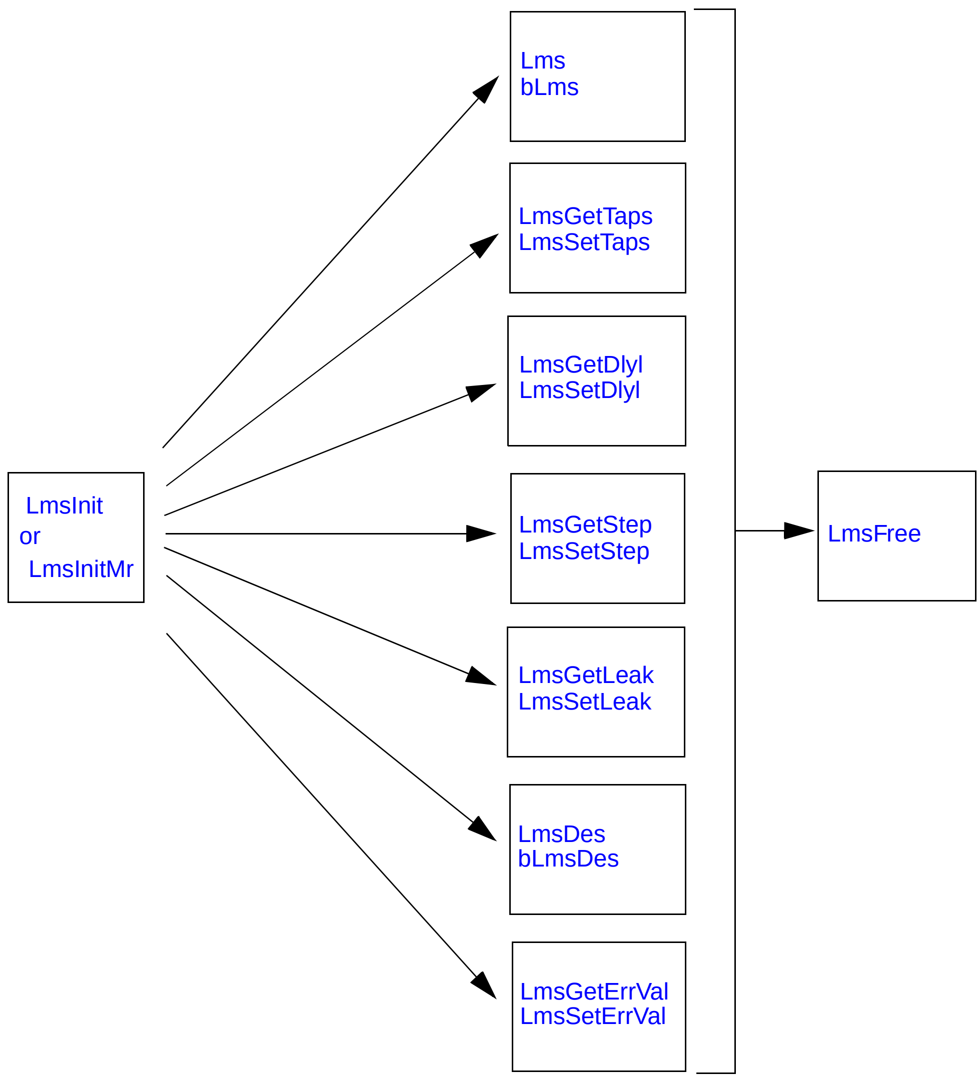
These functions provide a higher-level interface than the corresponding low-level LMS functions (see “Lmsl” on page 8-58 for a description of `nsp?Lmsl()`). In particular, they bundle the taps and delay line into a single state. Also, the LMS functions dynamically allocate memory for the taps and delay line; thus the arrays storing the taps and delay line values are not accessed after initialization, and need not exist while the filter exists.

To use the LMS adaptive filter functions, follow this general scheme:

1. Call `nsp?LmsInit()` to initialize a single-rate LMS filter or call `nsp?LmsInitMr()` to initialize a multi-rate LMS filter.
2. After this initialization, you have a choice of functions to call, depending on what you want to accomplish.
   a. Call the `nsp?Lms()` function to filter a single sample through a single-rate filter and/or call `nsp?bLms()` to filter a block of consecutive samples through a single-rate or multi-rate filter.
   b. Call the `nsp?LmsGetTaps()` function and then the `nsp?LmsSetTaps()` function to get and set the filter coefficients (taps).
   c. Call the `nsp?LmsGetDlyl()` function and then the `nsp?LmsSetDlyl()` function to get and set the values in the delay line.
   d. Call the `nspsLmsGetStep()` function and the `nspsLmsSetStep()` function to get and set the convergence step size values.
   e. Call the `nspsLmsGetLeak()` function and the `nspsLmsSetLeak()` function to get and set the leak values.
   f. Call the `nsp?LmsDes()` function to compute an error signal and filter a sample through a single-rate filter and/or call `nsp?bLmsDes()` to compute an error signal and filter a sample through a multi-rate signal.
   g. Call the `nsp?LmsGetErrVal()` function and the `nsp?LmsSetErrVal()` function to get and set the error signal of an LMS filter.
3. Call the `nspLmsFree()` function to release dynamic memory associated with the LMS filter.

Figure 8-4 illustrates the order of use of the LMS filter functions.

**Figure 8-4 Order of Use of the LMS Functions**



## LmsInit, LmsInitMr, LmsFree

*Initializes an adaptive FIR filter that uses the least mean-square (LMS) algorithm.*

```
void nspsLmsInit(NSPLmsType lmsType, const float *tapVals,
      int tapsLen, const float *dlyVals, float step, float leak,
      int errDly, NSPLmsState *statePtr);
void nspsLmsInitMr(NSPLmsType lmsType, const float *tapVals,
      int tapsLen, const float *dlyVals, float step, float leak,
      int errDly, int downFactor, int downPhase,
      NSPLmsState *statePtr);
      /* real delay line, real taps; single precision */

void nspcLmsInit(NSPLmsType lmsType, const SCplx *tapVals,
      int tapsLen, const SCplx *dlyVals, float step, float leak,
      int errDly, NSPLmsState *statePtr);
void nspcLmsInitMr(NSPLmsType lmsType, const SCplx *tapVals,
      int tapsLen, const SCplx *dlyVals, float step, float leak,
      int errDly, int downFactor, int downPhase,
      NSPLmsState *statePtr);
      /* complex delay line, complex taps; single precision */

void nspscLmsInit(NSPLmsType lmsType, const SCplx *tapVals,
      int tapsLen, const float *dlyVals, float step, float leak,
      int errDly, NSPLmsState *statePtr);
void nspscLmsInitMr(NSPLmsType lmsType, const SCplx *tapVals,
      int tapsLen, const float *dlyVals, float step, float leak,
      int errDly, int downFactor, int downPhase,
      NSPLmsState *statePtr);
      /* real delay line, complex taps; single precision */

void nspdLmsInit(NSPLmsType lmsType, const double *tapVals,
      int tapsLen, const double *dlyVals, float step, float leak,
      int errDly, NSPLmsState *statePtr);
```

```
void nspdLmsInitMr(NSPLmsType lmsType, const double *tapVals,
      int tapsLen, const double *dlyVals, float step, float leak,
      int errDly, int downFactor, int downPhase,
      NSPLmsState *statePtr);
      /* real delay line, real taps; double precision */

void nspzLmsInit(NSPLmsType lmsType, const DCplx *tapVals,
      int tapsLen, const DCplx *dlyVals, float step, float leak,
      int errDly, NSPLmsState *statePtr);
void nspzLmsInitMr(NSPLmsType lmsType, const DCplx *tapVals,
      int tapsLen, const DCplx *dlyVals, float step, float leak,
      int errDly, int downFactor, int downPhase,
      NSPLmsState *statePtr);
      /* complex delay line, complex taps; double precision */

void nspdzLmsInit(NSPLmsType lmsType, const DCplx *tapVals,
      int tapsLen, const double *dlyVals, float step, float leak,
      int errDly, NSPLmsState *statePtr);
void nspdzLmsInitMr((NSPLmsType lmsType, const DCplx *tapVals,
      int tapsLen, const double *dlyVals, float step, float leak,
      int errDly, int downFactor, int downPhase,
      NSPLmsState *statePtr);
      /* real delay line, complex taps; double precision */
void nspLmsFree(NSPLmsState *statePtr);
```

| | |
|---|---|
| *dlyVals* | Pointer to the array containing the delay line values. |
| *downFactor* | The factor used by the `nsp?LmsInitMr()` function for down-sampling multi-rate signals. |
| *downPhase* | The phase value used by the `nsp?LmsInitMr()` function for down-sampling multi-rate signals. |
| *errDly* | The delay (in samples) from the output signal of the LMS filter to the error signal input. The *errDly* argument is used by the `nsp?LmsInit()` and `nsp?LmsInitMr()` functions. |
| *leak* | How much the tap values “leak” towards 0.0 on each iteration. The value must be between 0.0 and 1.0. The *leak* argument is used by the `nsp?LmsInit()` and `nsp?LmsInitMr()` functions. |

| | |
|---|---|
| `lmsType` | Specifies the adaptation scheme to use with the filter description given. The value for `lmsType` must currently be set to `NSP_LmsDefault`. The `lmsType` argument is used by the `nsp?LmsInit()` and `nsp?LmsInitMr()` functions. |
| `statePtr` | Pointer to the `NSPLmsState` structure. |
| `step` | The convergence step size. This value must be between 0.0 and 1.0. A non-zero value enables adaptation while a value of 0.0 disables the filter adaptation. The `step` argument is used by the `nsp?LmsInit()` and `nsp?LmsInitMr()` functions. |
| `tapVals` | Pointer to the array containing the filter coefficient (taps) values. |
| `tapsLen` | The number of values in the array containing the filter coefficients (taps). |

## Description

The `nsp?LmsInit()` and `nsp?LmsInitMr()` functions initialize a single-rate LMS filter and a multi-rate LMS filter respectively. They are intended for cyclic processing. The `nspLmsFree()` function releases dynamic memory associated with the filter.

The `nsp?LmsInit()` and `nsp?LmsInitMr()` functions provide a higher-level interface than the corresponding low-level LMS functions (see `nsp?LmslInit()` and `nsp?LmslInitMr()` on page 8-47). In particular, `nsp?LmsInit()` and `nsp?LmsInitMr()` bundle the taps and delay line into a single state. They also dynamically allocate memory for the taps and delay line; thus the arrays `tapVals[n]` and `dlyVals[n]` are not accessed after initialization and need not exist while the filter exists.

Many combinations of input (`x(n)`) types and filter coefficient types are possible. Real or complex input can be mixed with real or complex filter coefficients. This is indicated by the `s`, `c`, `sc`, `d`, `z`, and `dz`, type codes following the `nsp` prefix in the function names above.

---

**NOTE.** *The different data types of* `nsp?LmsInit()` *and* `nsp?LmsInitMr()` *correspond to different combinations of real/complex inputs versus real/complex outputs. The complex input, real tap types (*`cs`*,* `zd`*), are not provided. While it is possible to implement such constrained LMS filters by projecting the error term onto the real taps, the method of projection is application-dependent. In contrast, the real input, complex tap types (*`sc`*,* `dz`*), are provided, but may have convergence problems depending on the input signal.*

---

**`nsp?LmsInit().`** The function `nsp?LmsInit()` initializes `statePtr` to describe a single-rate LMS filter. The argument *`lmsType`* specifies the adaptation scheme and must currently be `NSP_LmsDefault`. The *`tapsLen`* length array *`tapVals[n]`* specifies the initial filter coefficients. The array *`dlyVals[n]`* specifies the initial delay line contents. If *`dlyVals`* is non-`NULL`, it must have a length of *`tapsLen`* - 1. If *`dlyVals`* is `NULL` the delay line is initialized to zero.

The argument *`leak`* controls how much the tap values “leak” towards 0 on each iteration. Its value should be between 0.0 and 1.0. A value of 0.0 yields a traditional non-leaky LMS filter. A typical leaky filter uses a small value for *`leak`*.

The argument *`errDly`* specifies the delay (in samples) from the output signal of the LMS filter to the error signal input. In principle, the value of *`errDly`* must be at least 1. A delay of 1 corresponds to subtracting the filter output sample *`y(n)`* from the desired signal *`d(n)`* to obtain the error signal *`e(n)`*. The value of the error signal is then passed to the next invocation of the filter. In the general case, *`errDly`* is chosen as follows:

*`e(n)`* = *`d(n`* - *`errDly)`* - *`y(n`* - *`errDly)`*

where *`e(n)`* is the error signal.

The *`step`* argument specifies the convergence step size. This value must be between 0.0 and 1.0. A non-zero value enables adaptation while a value of 0.0 disables the filter adaptation.

**`nsp?LmsInitMr().`** The function `nsp?LmsInitMr()` initializes *`statePtr`* to describe a multi-rate LMS filter. The *`errDly`*, *`dlyVals`*, *`lmsType`*, *`tapVals`*, *`tapsLen`*, *`statePtr`*, *`step`*, and *`leak`*, arguments for the `nsp?LmsInitMr()` function are defined the same as for the `nsp?LmsInit()` function above.

The *`downFactor`* argument is the factor by which the signal is down-sampled. That is, `downFactor` - 1 samples are discarded from the signal. The argument *`downPhase`* determines which of the samples within each block are not discarded. The value of *`downPhase`* is required to be 0 ≤ *`downPhase`* < *`downFactor`*. For more information on down-sampling, see "DownSample" in Chapter 3 for the description of the `nsp?DownSample` function.

**`nspLmsFree().`** The `nspLmsFree()` function releases dynamic memory associated with the LMS adaptive filters created with the `nsp?LmsInit()` and `nsp?LmsInitMr()` functions.

**Application Notes:** The *`step`* and *`leak`* parameters are single-precision (`float`) in all data types of the functions. The extra precision available with double-precision (`double`) is not required for these parameters.

### Related Topics

| | |
|---|---|
| `LmslInit` | Initializes a low-level, single-rate LMS filter (see page 8-47). |
| `LmslInitMr` | Initializes a low-level, multi-rate LMS filter (see page 8-47). |

## Lms, bLms

*Filters samples through an LMS filter.*

```
float nspsLms(NSPLmsState *statePtr, float samp, float err);
float nspsbLms(NSPLmsState *statePtr, const float *inSamps,
      float err);
      /* real input, real error signal; single precision */

SCplx nspcLms(NSPLmsState *statePtr, SCplx samp, SCplx err);
SCplx nspcbLms(NSPLmsState *statePtr, const SCplx *inSamps,
      SCplx err);
      /* complex input, complex error signal; single precision */

SCplx nspscLms(NSPLmsState *statePtr, float samp, SCplx err);
SCplx nspscbLms(NSPLmsState *statePtr, const float *inSamps,
      SCplx err);
      /* real input, complex error signal; single precision */

double nspdLms(NSPLmsState *statePtr, double samp, double err);
double nspdbLms(NSPLmsState *statePtr, const double *inSamps,
      double err);
      /* real input, real error signal; double precision */

DCplx nspzLms(NSPLmsState *statePtr, DCplx samp, DCplx err);
DCplx nspzbLms(NSPLmsState *statePtr, const DCplx *inSamps,
      DCplx err);
      /* complex input, complex error signal; double precision */

DCplx nspdzLms(NSPLmsState *statePtr, double samp, DCplx err);
DCplx nspdzbLms(NSPLmsState *statePtr, const double *inSamps,
      DCplx err);
      /* real input, complex error signal; double precision */
```

| | |
|---|---|
| `err` | The error signal sample. |
| `inSamps` | Pointer to the array containing the input samples for the `nsp?bLms()` function. |
| `samp` | The input sample for the `nsp?Lms()` function. |

*`statePtr`* Pointer to the `NSPLmsState` structure.

## Description

The `nsp?Lms()` and `nsp?bLms()` functions perform a single iteration of LMS adaptation and filtering.

Many combinations of input (*`x(n)`*) types and filter coefficient types are possible. Real or complex input can be mixed with real or complex filter coefficients. This is indicated by the `s`, `c`, `sc`, `d`, `z`, and `dz`, type codes following the `nsp` prefix in the function names above.

---

**NOTE.** *The complex input, real error signal types (`cs`, `zd`), are not provided. While it is possible to implement such constrained LMS filters by projecting the error term onto the real taps, the method of projection is application-dependent. In contrast, the real input, complex error signal types (`sc`, `dz`), are provided, but may have convergence problems depending on the input signal.*

---

**Previous Tasks:** Before using `nsp?Lms()` or `nsp?bLms()`, you must initialize *`statePtr`* by calling either `nsp?LmsInit()` or `nsp?LmsInitMr()`.

**`nsp?Lms().`** The `nsp?Lms()` function filters a sample through a single-rate filter. The input sample *`samp[n]`* is *`x(n)`*, *`err`* is the error sample *`e(n)`*, and the output sample *`y(n)`* is returned, as specified for `nsp?Lmsl()`.

**`nsp?bLms().`** The `nsp?bLms()` function filters samples through a multi-rate filter to produce a single output sample. The argument *`statePtr`* uses the *`downFactor`* argument specified by `nsp?LmsInitMr()`. The argument *`err`* is the error signal *`e(n)`* and the *`downFactor`* length array *`inSamps[n]`* provides samples of *`x(n)`*. The filtered result *`y(n)`* is returned.

Even though `nsp?bLms()` has the `b` prefix flag to indicate a blocked function, `nsp?bLms()` does not perform more than one iteration. This is because doing so would introduce excess delay into the error signal. Instead, this function is provided for multi-rate filtering, which requires a vector (blocked) input array.

Example 8-9 illustrates the use of the LMS functions to initialize and filter a signal sample.

### Example 8-9 Filtering with the LMS Filter Functions

```
/*
 * standard single-rate
 * filtering
 */
NSPLmsState  lmsSt;
double       taps[32];
int          i;
double       xval, yval, dval, eval = 0.0;

/* insert code here to initialize taps */

nspdLmsInit(NSP_LmsDefault, taps, 32, NULL, 0.01, 0.0, 1, &lmsSt);
for (i=0; i<2000; i++) {
      xval = /* insert code here to get next value of x(n) */;
      yval = nspdLms(&tapSt, xval, eval);
      dval = /* insert code here to get next value of d(n) */;
      eval = dval - yval;
}
```

### Related Topics

| | |
|---|---|
| `bLmsl` | Filters samples through a multi-rate, low-level LMS filter to produce a single sample (see page 8-58). |
| `LmsInit` | Initializes a single-rate, low-level LMS filter (see page 8-73). |
| `LmsInitMr` | Initializes a multi-rate, low-level LMS filter (see page 8-73). |

| | |
|---|---|
| `Lmsl` | Filters a sample through a single-rate, low-level LMS filter(see page 8-58). |

## LmsGetTaps, LmsSetTaps

*Gets and sets the taps coefficients of an LMS filter.*

```
void nspsLmsGetTaps(const NSPLmsState *statePtr, float *outTaps);
void nspsLmsSetTaps(const float *inTaps, NSPLmsState *statePtr);
      /* real values; single precision */
void nspcLmsGetTaps(const NSPLmsState *statePtr, SCplx *outTaps);
void nspcLmsSetTaps(const SCplx *inTaps, NSPLmsState *statePtr);
      /* complex values; single precision */
void nspdLmsGetTaps(const NSPLmsState *statePtr, double *outTaps);
void nspdLmsSetTaps(const double *inTaps, NSPLmsState *statePtr);
      /* real values; double precision */
void nspzLmsGetTaps(const NSPLmsState *statePtr, DCplx *outTaps);
void nspzLmsSetTaps(const DCplx *inTaps, NSPLmsState *statePtr);
      /* complex values; double precision */
```

| | |
|---|---|
| *`inTaps`* | Pointer to the array holding copies of the tap coefficients for the `nsp?LmsSetTaps()` function. |
| *`outTaps`* | Pointer to the array holding copies of the tap coefficients for the `nsp?LmsGetTaps()` function. |
| *`statePtr`* | Pointer to the `NSPLmsState` structure. |

### Description

The `nsp?LmsGetTaps()` and `nsp?LmsSetTaps()` functions get and set the taps of an LMS adaptive filter. The data type of the function used here must match the data type for the taps used during initialization.

**Previous Tasks:** Before calling either `nsp?LmsGetTaps()` or `nsp?LmsSetTaps()`, you must initialize the state structure `NSPLmsState` pointed to by *`statePtr`* by calling either `nsp?LmsInit()` or `nsp?LmsInitMr()`. You must also specify the tap length *`tapsLen`* and the taps *`h(0)...h(tapsLen`* - 1*`)`*.

**`nsp?LmsGetTaps().`** The `nsp?LmsGetTaps()` function copies the tap coefficients from *`statePtr`* to the *`tapsLen`* length array *`outTaps[n]`*, unpermuting them if required so that *`outTaps[n]=h(n)`*.

**`nsp?LmsSetTaps().`** The `nsp?LmsSetTaps()` function copies the *`tapsLen`* tap coefficients from the *`inTaps[n]`* array into *`statePtr`*, permuting them if required so that *`h(n)=inTaps[n]`*.

### Related Topics

| | |
|---|---|
| `LmslGetTaps` | Gets the filter coefficient (taps) values for a low-level LMS adaptive filter (see page 8-52). |
| `LmslSetTaps` | Sets the filter coefficient (taps) values for a low-level LMS adaptive filter (see page 8-52). |
| `LmslInit` | Initializes a low-level, single-rate LMS filter (see page 8-47). |
| `LmslInitMr` | Initializes a low-level, multi-rate LMS filter (see page 8-47). |

## LmsGetDlyl, LmsSetDlyl

*Gets and sets the delay line contents of an LMS filter.*

```
void nspsLmsGetDlyl(const NSPLmsState *statePtr, float *outDlyl);
void nspsLmsSetDlyl(const float *inDlyl, NSPLmsState *statePtr);
      /* real values; single precision */

void nspcLmsGetDlyl(const NSPLmsState *statePtr, SCplx *outDlyl);
```

```
void nspcLmsSetDlyl(const SCplx *inDlyl, NSPLmsState *statePtr);
      /* complex values; single precision */

void nspdLmsGetDlyl(const NSPLmsState *statePtr, double *outDlyl);
void nspdLmsSetDlyl(const double *inDlyl, NSPLmsState *statePtr);
      /* real values; double precision */

void nspzLmsGetDlyl(const NSPLmsState *statePtr, DCplx *outDlyl);
void nspzLmsSetDlyl(const DCplx *inDlyl, NSPLmsState *statePtr);
      /* complex values; double precision */
```

| | |
|---|---|
| *inDlyl* | Pointer to the array holding copies of the delay line values for the `nsp?LmsSetDlyl()` function. |
| *outDlyl* | Pointer to the array holding copies of the delay line values for the `nsp?LmsGetDlyl()` function. |
| *statePtr* | Pointer to the `NSPLmsState` structure. |

## Description

The `nsp?LmsGetDlyl()` and `nsp?LmsSetDlyl()` functions get and set the delay line of an adaptive LMS filter. The data type of the function used here must match the data type for the delay line used during initialization.

**Previous Tasks:** Before calling either `nsp?LmsGetDlyl()` or `nsp?LmsSetDlyl()`, you must initialize the state structure `NSPLmsState` pointed to by *statePtr* by calling either `nsp?LmsInit()` or `nsp?LmsInitMr()`. You must also specify the tap length *tapsLen* and the delay line values. For single-rate filters, the delay line values are denoted as $x(n - tapsLen - errDly + 1)...x(n - 1)$; for multi-rate filters they are denoted as $x(n - tapsLen - errDly - downFactor + 1)...x(n - 1)$.

**`nsp?LmsGetDlyl()`.** The `nsp?LmsGetDlyl()` function takes the delay line values in *statePtr* and stores them into the *tapsLen* length array *outDlyl[n]*. The `nsp?LmsGetDlyl()` function unpermutes the delay line values if necessary so that $outDlyl[k]=x(n - k - 1)$, where $x(n - 1)$ is the last filtered sample. For single-rate filters *outDlyl[n]* must be $tapsLen + errDly - 1$ long; for multi-rate filters it must be $tapsLen + errDly + downFactor - 1$ long.

**nsp?LmsSetDlyl().** The nsp?LmsSetDlyl() function permutes the values in the array *inDlyl[n]* and stores them into *statePtr*. For single-rate filters, *inDlyl[n]* must be *tapsLen* + *errDly* - 1 long; for multi-rate filters it must be *tapsLen* + *errDly* + *downFactor* - 1 long. If *inDlyl* is NULL, the delay line is initialized to all zeros.

### Related Topics

| | |
|---|---|
| LmslGetDlyl | Gets the delay line values for a low-level LMS filter (see page 8-54). |
| LmslSetDlyl | Sets the delay line values for a low-level LMS filter (see page 8-54). |
| LmslInit | Initializes a low-level, single-rate LMS filter (see page 8-47). |
| LmslInitMr | Initializes a low-level, multi-rate LMS filter (see page 8-47). |

## LmsGetStep, LmsSetStep, LmsGetLeak, LmsSetLeak

*Gets and sets the leak and step values of an LMS filter.*

```
float nspsLmsGetStep(const NSPLmsState *statePtr);
void nspsLmsSetStep(float step, NSPLmsState *statePtr);
      /* real values; single precision */

float nspsLmsGetLeak(const NSPLmsState *statePtr);
void nspsLmsSetLeak(float leak, NSPLmsState *statePtr);
      /* real values; single precision */
```

| | |
|---|---|
| *leak* | How much the tap values “leak” towards 0.0 on each iteration. The value must be between 0.0 and 1.0. |
| *statePtr* | Pointer to the NSPLmsState structure. |

| | |
|---|---|
| *`step`* | The convergence step size. This value must be between 0.0 and 1.0. |

## Description

The `nspsLmsGetLeak()` and `nspsLmsSetLeak()`functions allow your application to get and set the *`leak`* parameter of an LMS filter described by *`statePtr`*.

The `nspsLmsGetStep()` and `nspsLmsSetStep()` functions allow your application to get and set the *`step`* parameters of an LMS filter described by *`statePtr`*.

These functions can be used for filters of any type since only single-precision *`step`* and *`leak`* parameters are supported.

**Application Notes:** On most platforms, the `nspsLmsGetLeak()`, `nspsLmsSetLeak()`, `nspsLmsGetStep()`, and `nspsLmsSetStep()` functions are implemented by calling the corresponding low-level functions `nspsLmslGetLeak()`, `nspsLmslSetLeak()`, `nspsLmslGetStep()`, and `nspsLmslSetStep()`.

## Related Topics

| | |
|---|---|
| `LmslGetLeak` | Gets the *`leak`* parameter for a low-level LMS adaptive filter (see page 8-57). |
| `LmslGetStep` | Gets the *`step`* parameter for a low-level LMS adaptive filter (see page 8-57). |
| `LmslSetLeak` | Sets the *`leak`* parameter for a low-level LMS adaptive filter (see page 8-57). |
| `LmslSetStep` | Sets the *`step`* parameter for a low-level LMS adaptive filter (see page 8-57). |

## LmsDes, bLmsDes

*Filters samples through an LMS filter using a desired-output signal for adaptation instead of an error signal.*

```
float nspsLmsDes(NSPLmsState *statePtr, float samp, float des);
void nspsbLmsDes(NSPLmsState *statePtr, const float *inSamps,
      const float *desSamps, float *outSamps, int numIters);
      /* real input, real desired signal; single precision */

SCplx nspcLmsDes(NSPLmsState *statePtr, SCplx samp, SCplx des);
void nspcbLmsDes(NSPLmsState *statePtr, const SCplx *inSamps,
      const SCplx *desSamps, SCplx *outSamps, int numIters);
      /* complex input, complex desired signal; single precision */

SCplx nspscLmsDes(NSPLmsState *statePtr, float samp, SCplx des);
void nspscbLmsDes(NSPLmsState *statePtr, const float *inSamps,
      const SCplx *desSamps, SCplx *outSamps, int numIters);
      /* real input, complex desired signal; single precision */

double nspdLmsDes(NSPLmsState *statePtr, double samp, double des);
void nspdbLmsDes(NSPLmsState *statePtr, const double *inSamps,
      const double *desSamps, double *outSamps, int numIters);
      /* real input, real desired signal; double precision */

DCplx nspzLmsDes(NSPLmsState *statePtr, DCplx samp, DCplx des);
void nspzbLmsDes(NSPLmsState *statePtr, const DCplx *inSamps,
      const DCplx *desSamps, DCplx *outSamps, int numIters);
      /* complex input, complex desired signal; double precision */

DCplx nspdzLmsDes(NSPLmsState *statePtr, double samp, DCplx des);
void nspdzbLmsDes(NSPLmsState *statePtr, const double *inSamps,
      const DCplx *desSamps, DCplx *outSamps, int numIters);
      /* real input, complex desired signal; double precision */
```

| | |
|---|---|
| *des* | A single sample of the desired signal. |
| *desSamps* | Pointer to the array containing samples of the desired signal. |

| | |
|---|---|
| *inSamps* | Pointer to the array containing samples of the input signal. For a multi-rate filter, the length of the array inSamps is *numIters * downFactor*. |
| *numIters* | The length of the arrays *inSamps*, *desSamps*, and *outSamps* for a single-rate filter. |
| *outSamps* | Pointer to the array containing samples of the output signal. |
| *samp* | A single sample of the input signal. |
| *statePtr* | Pointer to the `NSPLmsState` structure. |

## Description

The `nsp?LmsDes()` and `nsp?bLmsDes()` functions perform LMS adaptation and filtering. They also compute the error signal *e(n)* from the desired signal *d(n)*. This is different from the functions `nsp?Lms()` and `nsp?bLms()` which assume the presence of a pre-computed error signal.

The low-level LMS filtering functions first perform adaptation using the input argument *err* and then perform one iteration of filtering. The high-level functions `nsp?LmsDes()` and `nsp?bLmsDes()` use the desired signal *d(n)* after each iteration of filtering and compute the error value for the next filter adaptation. The computed error is stored in the *errVal* field in *statePtr*.

Many combinations of input (*x(n)*) types and filter coefficient types are possible. Real or complex input can be mixed with real or complex sample signals. This is indicated by the `s`, `c`, `sc`, `d`, `z`, and `dz`, type codes following the `nsp` prefix in the function names above.

**NOTE.** *The complex input, real signal sample types (`cs`, `zd`), are not provided. In contrast, the real input, complex signal sample types (`sc`, `dz`), are provided.*

**Previous Tasks:** Before using `nsp?LmsDes()` or `nsp?bLmsDes()`, you must initialize *`statePtr`* by calling either `nsp?LmsInit()` or `nsp?LmsInitMr()`.

**`nsp?LmsDes().`** The `nsp?LmsDes()` function filters a sample through a single-rate filter. The input sample *`samp`* is *`x(n)`*, the desired signal sample *`des`* is *`d(n)`* and the result *`y(n)`* is returned. Since adaptation is performed before the filter convolution, the error value for adaptation is obtained from the *`errVal`* field of *`statePtr`*. The `nsp?LmsDes()` function uses the desired signal sample *`des`* to compute the error value *`e(n)`* for the next adaptation and to update *`errVal`*.

**`nsp?bLmsDes().`** The function `nsp?bLmsDes()` filters a block of samples through a single-rate or multi-rate filter. For a single-rate filter, the *`numIters`* length arrays *`inSamps[n]`* and *`desSamps[n]`* contain samples of the input signal *`x(n)`* and the desired signal *`d(n)`* respectively. The *`numIters`* output samples are returned in *`outSamps[n]`*. The result is the same as *`numIters`* consecutive calls to `nsp?LmsDes()`.

For a multi-rate filter, the *`numIters * downFactor`* length array *`inSamps[n]`* contains samples of *`x(n)`* and the *`numIters`* length array *`desSamps[n]`* contains samples of *`d(n)`*. The *`numIters`* output samples are returned in *`outSamps[n]`*.

As with `nsp?LmsDes()`, the error value for the first adaptation of the sample block is taken from the *`errVal`* field of *`statePtr`*. The *`errVal`* field is updated using the last output sample and the last desired signal sample before the function exits.

**NOTE.** *The `nsp?LmsDes()` and `nsp?bLmsDes()` functions are different from `nsp?Lms()` and `nsp?bLms()`. Since `nsp?Lms()` and `nsp?bLms()` do not update `errVal`, it is up to the application to check or update `errVal` (using `nsp?LmsGetErrVal()` or `nsp?LmsSetErrVal()`) when switching from one mode to the other.*

Example 8-10 illustrates the use of the single-rate filtering with the `nsp?LmsDes()` function, and Example 8-11 illustrates the use of the LMS filtering using nsp?Lms() and `nsp?LmsDes()` functions.

**Example 8-10 Single-Rate Filtering with the nsp?LmsDes() Function**

```
/*
 * standard single-rate
 * filtering
 */
NSPLmsState  lmsSt;
double       taps[32];
int          i;
double       xval, yval, dval;
/* insert code here to initialize taps */
nspdLmsInit(NSP_LmsDefault, taps, 32, NULL, 0.01, 0.0, 1, &lmsSt);
for (i=0; i<2000; i++) {
   xval = /* insert code here to get next value of x(n) */;
   dval = /* insert code here to get next value of d(n) */;
   yval = nspdLmsDes(&tapSt, xval, dval);
}
```

**Example 8-11 LMS Filtering Using nsp?Lms() and nsp?LmsDes()**

```
/*
 * LMS filtering using both
 * nsp?Lms() and nsp?LmsDes() */
 */
NSPLmsState  lmsSt;
double       taps[32];
int             i,j;
double       xval, yval, dval, eval=0.0;

/* insert code here to initialize taps */
}
```

continued ☞

**Example 8-11 LMS Filtering Using nsp?Lms() and nsp?LmsDes()** (continued)

```
nspdLmsInit(NSP_LmsDefault, taps, 32, NULL, 0.01, 0.0, 1, &lmsSt);
for (j=0; j<10; j++) {
   for (i=0; i<2000; i++) {
      xval = /* insert code to get next value of x(n) */
      yval = nspdLms(&tapSt, xval, eval);
      dval = /* insert code to get next value of d(n) */;
      eval = dval - yval;
   }
   nspdLmsSetErrVal(eval, &lmsSt);
   for (i=0; i<2000; i++) {
      xval = /* insert code to get next value of x(n) */;
      dval = /* insert code to get next value of d(n) */;
      yval = nspdLmsDes(&tapSt, xval, dval);
   }
   eval = nspdLmsGetErrVal(&lmsSt);
```

### Related Topics

| | |
|---|---|
| `LmsInit` | Initializes a single-rate adaptive LMS filter (see page 8-73). |
| `LmsInitMr` | Initializes a multi-rate adaptive LMS filter (see page 8-73). |

## LmsGetErrVal, LmsSetErrVal

*Gets and sets the error signal of an LMS adaptive filter if computed from the desired signal by the Signal Processing Library.*

```
float nspsLmsGetErrVal(const NSPLmsState *statePtr);
void nspsLmsSetErrVal(float err, NSPLmsState *statePtr);
      /* real input; single precision */
```

```
SCplx nspcLmsGetErrVal(const NSPLmsState *statePtr);
void nspcLmsSetErrVal(SCplx err, NSPLmsState *statePtr);
      /* complex input; single precision */

double nspdLmsGetErrVal(const NSPLmsState *statePtr);
void nspdLmsSetErrVal(double err, NSPLmsState *statePtr);
      /* real input; double precision */

DCplx nspzLmsGetErrVal(const NSPLmsState *statePtr);
void nspzLmsSetErrVal(DCplx err, NSPLmsState *statePtr);
      /* complex input; double precision */
```

| | |
|---|---|
| *err* | The error signal sample. |
| *statePtr* | Pointer to the `NSPLmsState` structure. |

## Description

The `nsp?LmsGetErrVal` and `nsp?LmsSetErrVal` functions allow the application to get and set the error value (*errVal*) used by `nsp?LmsDes()` and `nsp?bLmsDes()`. The data type of the function used here must match the data type of the filter output. For more information on *errVal*, see "LmsInit" on page 8-73 for `nsp?LmsInit()` and "LmsDes" on page 8-86 for `nsp?LmsDes()`.

**Previous Tasks:** The filter state *statePtr* must have been previously initialized by `nsp?LmsInit()` or `nsp?LmsInitMr()`.

**`nsp?LmsGetErrVal().`** The `nsp?LmsGetErrVal()` function returns the value of *errVal* which is stored in *statePtr*.

**NOTE.** *The error value (*errVal*) field is initialized to zero when *statePtr* is initialized. Thus, `nsp?LmsGetErrVal()` will return zero if *errVal* has not been set with `nsp?LmsSetErrVal()`, or if a new *errVal* has not been computed by `nsp?LmsDes()` or `nsp?bLmsDes()`.*

**`nsp?LmsSetErrVal().`** The function `nsp?LmsSetErrVal()` copies the error signal sample *err* into *errVal* in *statePtr*.

### Related Topics

| | |
|---|---|
| `LmsInit` | Initializes a single-rate adaptive LMS filter (see page 8-73). |
| `LmsInitMr` | Initializes a multi-rate adaptive LMS filter (see page 8-73). |

## Low-Level IIR Filter Functions

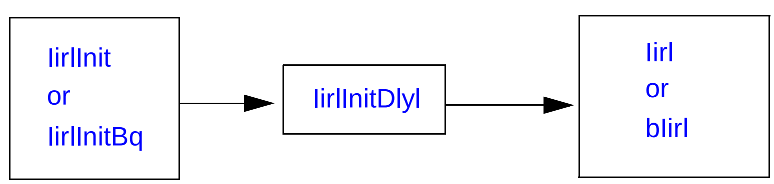
The functions described in this section initialize a low-level, infinite impulse response (IIR) filter.

To initialize and use a low-level IIR filter, follow this general scheme:

1. Call either `nsp?IirlInit()` to initialize the filter as an arbitrary order IIR filter or call `nsp?IirlInitBq()` to initialize the filter as a cascade of biquads.
2. Call `nsp?IirlInitDlyl()` to initialize a delay line for the IIR filter.
3. Call the `nsp?Iirl()` function to filter a single sample through a low-level IIR filter and/or call `nsp?bIirl()` to filter a block of consecutive samples through a low-level IIR filter.

Figure 8-5 illustrates the order of use of the low-level IIR filter functions.

**Figure 8-5 Order of Use of the Low-Level IIR Functions**



## IirlInit, IirlInitGain, IirlInitBq, IirlInitDlyl

*Initializes a low-level infinite impulse response filter.*

```
void nspsIirlInit(NSPIirType iirType, float *taps, int order,
      NSPIirTapState *tapStPtr);
void nspsIirlInitBq(NSPIirType iirType, float *taps, int numQuads,
      NSPIirTapState *tapStPtr);
void nspsIirlInitDlyl(const NSPIirTapState *tapStPtr, float *dlyl,
      NSPIirDlyState *dlyStPtr);
      /* real values; single precision */

void nspcIirlInit(NSPIirType iirType, SCplx *taps, int order,
      NSPIirTapState *tapStPtr);
void nspcIirlInitBq(NSPIirType iirType, SCplx *taps, int numQuads,
      NSPIirTapState *tapStPtr);
void nspcIirlInitDlyl(const NSPIirTapState *tapStPtr, SCplx *dlyl,
      NSPIirDlyState *dlyStPtr);
      /* complex values; single precision */

void nspdIirlInit(NSPIirType iirType, double *taps, int order,
      NSPIirTapState *tapStPtr);
void nspdIirlInitBq(NSPIirType iirType, double *taps, int numQuads,
      NSPIirTapState *tapStPtr);
void nspdIirlInitDlyl(const NSPIirTapState *tapStPtr, double *dlyl,
      NSPIirDlyState *dlyStPtr);
      /* real values; double precision */

void nspzIirlInit(NSPIirType iirType, DCplx *taps, int order,
      NSPIirTapState *tapStPtr);
void nspzIirlInitBq(NSPIirType iirType, DCplx *taps, int numQuads,
      NSPIirTapState *tapStPtr);
void nspzIirlInitDlyl(const NSPIirTapState *tapStPtr, DCplx *dlyl,
      NSPIirDlyState *dlyStPtr);
      /* complex values; double precision */
```

```
void nspwIirlInit(NSPIirType iirType, float *taps, int order,
      NSPIirTapState *tapStPtr);
void nspwIirlInitGain(NSPIirType iirType, float *taps, int order,
      NSPIirTapState *tapStPtr, float gain, int InputRange);
void nspwIirlInitBq(NSPIirType iirType, float *taps, int numQuads,
      NSPIirTapState *tapStPtr);
void nspwIirlInitDlyl(const NSPIirTapState *tapStPtr, long int *dlyl,
NSPIirDlyState *dlyStPtr);
      /* real values; short integer */
```

| | |
|---|---|
| *dlyl* | Pointer to the array storing the delay line values. The *dlyl* argument is used by the `nsp?IirlInitDlyl()` function. |
| *dlyStPtr* | Pointer to the `NSPIirDlyState` structure. |
| *iirType* | Specifies the filter structure to use with the filter description given. The value for *iirType* must currently be `NSP_IirDefault`. This argument is used by the `nsp?IirlInit()` and `nsp?IirlInitBq()` functions. |
| *numQuads* | The number of cascades of biquads (second-order IIR sections). The *numQuads* argument is used by the `nsp?IirlInitBq()` function. |
| *order* | The order of the IIR filter. This argument is used by the `nsp?IirlInit()` function. |
| *taps* | Pointer to the array of filter coefficients used by the `nsp?IirlInit()` and `nsp?IirlInitBq()` functions. |
| *tapStPtr* | Pointer to the `NSPIirTapState` structure. |
| *gain* | The gain coefficient for the filter output signal. Must have positive value. This argument is used by the `nspwIirlInitGain()` function. |
| *InputRange* | Specifies the bit range of the input signal (from 4 to 16 bit). This argument is used by the `nspwIirlInitGain()` function. |

## Discussion

The `nsp?IirlInit()`, `nspwIirlInitGain()`, `nsp?IirlInitBq()`, and `nsp?IirlInitDlyl()` functions initialize a low-level IIR filter. The choice of `nsp?IirlInit()` or `nsp?IirlInitBq()` selects whether the filter is described as an arbitrary-order IIR filter or as a cascade of biquads.

**`nsp?IirlInit()`.** The `nsp?IirlInit()` function initializes *`tapStPtr`* to describe a low-level IIR filter of order *`order`*. The argument *`iirType`* selects the filter structure to use with the filter description given. The filter structure is the organization of delay elements, gain elements, and adders that make up the filter (common filter structures are, for example, "direct form 1," "direct form 2," and so on). Multiple structures can implement the same filter, but the contents of the delay line will have different meaning depending on the structure you choose to use. Both the choice of initialization function and the value of *`iirType`* combine to select the desired filter implementation. The value of *`iirType`* must currently be set to `NSP_IirDefault`, meaning that the library is free to use whichever filter structure is most natural.

The array *`taps[n]`* describes a filter with the following transfer function:

$$H(z) = \frac{\sum_{k=0}^{N-1} \mathtt{taps}[k] \cdot z^{-k}}{\sum_{k=0}^{N-1} \mathtt{taps}[N+k] \cdot z^{-k}}, \quad N = \mathtt{order} + 1$$

Thus, there are 2(*`order`* + 1) elements in the array *`taps[N]`*. The value of *`taps[N]`* must not be 0.0, and is generally 1.0. If the value of *`taps[N]`* is not equal to 1.0, the initialization function (that is, either `nsp?IirlInit()` or `nsp?IirlInitBq()`) will typically normalize the taps so that it is 1.0.

**`nspwIirlInitGain()`.** Use this function to initialize the integer flavor of the IIR filter if the input signal representation requires less than 16 bits. This provides opportunity for optimal conversion of filter taps from the floating-point to the internal short format. The *`InputRange`* argument specifies the actual bit range of the input signal, which can be from 4 to 16 bits. The effect of calling `nspwIirlInitGain()`with *`gain`*=1.0 and *`InputRange`*=16 is the same as of calling the `nspwIirlInit()` function.

**`nsp?IirlInitBq()`.** The `nsp?IirlInitBq()` function initializes *`tapStPtr`* to reference a cascade of biquads (second-order IIR sections). The filter type argument *`iirType`* describes the filter structure to use. As described above, this must currently be `NSP_IirDefault`. The array *`taps[n]`* describes a set of filters as follows:

$$H_i(z) = \frac{taps[6\times i+0] + taps[6\times i+1]\cdot z^{-1} + taps[6\times i+2]\cdot z^{-2}}{taps[6\times i+3] + taps[6\times i+4]\cdot z^{-1} + taps[6\times i+5]\cdot z^{-2}}$$

$$H(z) = \prod_{i=0}^{numQuads-1} H_i(z)$$

Note that *`taps`*[6 x *`i`*] and *`taps`*[6 x *`i`* + 3] must not be 0.0. Most implementations normalize the taps so that *`taps`*[6 x *`i`* + 0] and *`taps`*[6 x *`i`* + 3] are 1.0; this requires a separate gain term.

**`nsp?IirlInitDlyl()`.** The `nsp?IirlInitDlyl()` function initializes *`dlyStPtr`* to reference a delay line for an IIR filter. For the arbitrary-order IIR filter, *`dlyl`* contains *`order`* elements, and for the biquad IIR filter, *`dlyl`* must contain 2 x *`numQuads`* elements. For *`iirType`* = `NSP_IirDefault`, the delay line is set to all zeros. The data type of the delay line initialization must match the data type of the filter output.

Do not deallocate or overwrite the arrays *`taps[n]`* and *`dlyl[n]`* during the life of the filter. Your application must not directly access these arrays because the `nsp?IirlInit()`, `nsp?IirlInitBq()` and `nsp?IirlInitDlyl()` functions can permute their contents in an implementation-dependent way.

**Application Notes:** The `nsp?IirlInit()` and `nsp?IirlInitBq()` functions can use any filter structure to implement the transfer function. For efficiency, the implementation can permute the taps and delay line. For better accuracy of the w-flavor functions, float taps and long int delay line are used. The internal usage and representation of *`taps[n]`* and *`dlyl[n]`* are implementation- and processor-dependent.

The filter structure `NSP_IirDefault` is implementation-dependent and might or might not permute and/or normalize the taps and delay line.

### Related Topics

| | |
|---|---|
| `bIirl` | Filters a block of samples through a low-level IIR filter (see page 8-97). |
| `Iirl` | Filters a single sample through a low-level IIR filter (see page 8-97). |

## Iirl, bIirl

*Filters a signal through a low-level IIR filter.*

```
float nspsIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, float samp);
void nspsbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const float *inSamps,
      float *outSamps, int numIters);
      /* real input, real taps; single precision */

SCplx nspcIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, SCplx samp);
void nspcbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const SCplx *inSamps,
      SCplx *outSamps, int numIters);
      /* complex input, complex taps; single precision */

SCplx nspscIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, float samp);
void nspscbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const float *inSamps,
      SCplx *outSamps, int numIters);
      /* real input, complex taps; single precision */

SCplx nspcsIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, SCplx samp);
```

```
void nspcsbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const SCplx *inSamps,
      SCplx *outSamps, int numIters);
      /* complex input, real taps; single precision */

double nspdIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, double samp);
void nspdbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const double *inSamps,
      double *outSamps, int numIters);
      /* real input, real taps; double precision */
DCplx* nspzIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, DCplx samp);
void nspzbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const DCplx *inSamps,
      DCplx *outSamps, int numIters);
      /* complex input, complex taps; double precision */

DCplx nspdzIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, double samp);
void nspdzbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const double *inSamps,
      DCplx *outSamps, int numIters);
      /* real input, complex taps; double precision */

DCplx nspzdIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, DCplx samp);
void nspzdbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const DCplx *inSamps,
      DCplx *outSamps, int numIters);
      /* complex input, real taps; double precision */

short nspwIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, short samp, int ScaleMode,
      int *ScaleFactor);
void nspwbIirl(const NSPIirTapState *tapStPtr,
      NSPIirDlyState *dlyStPtr, const short *inSamps,
      short *outSamps, int numIters, int ScaleMode,
      int *ScaleFactor);
      /* real input, real taps; short integer */
```

| | |
|---|---|
| *dlyStPtr* | Pointer to the `NSPIirDlyState` structure. |

| | |
|---|---|
| *inSamps* | Pointer to the array containing the input samples for the nsp?bIirl() function. |
| *numIters* | The number of samples to be filtered by the nsp?bIirl() function. |
| *outSamps* | Pointer to the array containing the output samples for the nsp?bIirl() function. |
| *samp* | The input sample for the nsp?bIirl() function. |
| *tapStPtr* | Pointer to the NSPIirTapState structure. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

The nsp?Iirl() and nsp?bIirl() functions filter samples through a low-level IIR filter. The different types of functions correspond to different combinations of real/complex taps and input samples. The data type of the delay line must match the data type of the filter output. The data type codes following the nsp prefix in the function names and the real/complex combinations are described in Table 8-6.

**Table 8-6 Delay Line and Output Data Types for nsp?Iirl() and nsp?bIirl() Functions**

| Type Code | Input Type | Filter Coefficient Type | Delay Line Type | Output Type |
|---|---|---|---|---|
| s | float | float | float | float |
| c | SCplx | SCplx | SCplx | SCplx |
| sc | float | SCplx | SCplx | SCplx |
| cs | SCplx | float | SCplx | SCplx |
| d | double | double | double | double |
| z | DCplx | DCplx | DCplx | DCplx |
| dz | double | DCplx | DCplx | DCplx |
| zd | DCplx | double | DCplx | DCplx |
| w | short | float | long int | short |

**Previous Tasks:** Before using either `nsp?Iirl()` or `nsp?bIirl()`, you must initialize the structure pointed to by *`tapStPtr`* by calling either `nsp?IirlInit()` or `nsp?IirlInitBq()`. You must also initialize the structure pointed to by *`dlyStPtr`* by calling `nsp?IirlInitDlyl()`.

**`nsp?Iirl()`.** The `nsp?Iirl()` function filters a single sample *`samp[n]`* through a low-level IIR filter and returns the result.

**`nsp?bIirl()`.** The `nsp?bIirl()` function filters a block of *`numIters`* samples in the array *`inSamps[n]`* through a low-level IIR filter and returns the result in the array *`outSamps[n]`*.

Example 8-12 illustrates the use of the low-level IIR functions in filtering a signal sample.

### Example 8-12 Using the Low-Level IIR Functions to Filter a Sample

```
/* filter a single sample through an IIR filter */
NSPIirTapState tapSt;
NSPIirDlyState  dlySt;
double       taps[6] = { 1.0, -2.0, 1.0, 1.0, -1.732, 1.0};
double       dlyl[2];
int             i;
double       xval, yval, inSamps[2000], outSamps[2000];

nspdIirlInitBq(NSP_IirDefault, taps, 1, &tapSt);
nspdIirlInitDlyl(&tapSt, dlyl, &dlySt);
for (i=0; i < 2000; i++) {
      xval = /* insert code here to get the
               * next value of x
               */;
      yval = nspdIirl(&tapSt, &dlySt, xval);
      /* yval has the output sample */
}
```

Example 8-13 illustrates the use of the low-level IIR functions in filtering a block of samples.

**Example 8-13 Using the Low-Level IIR Functions to Filter a Block of Samples**

```
/* standard block filtering */
nspdIirlInit(NSP_IirDefault, taps, 2, &tapSt);
nspdIirlInitDlyl(&tapSt, dlyl, &dlySt);

/* Insert code here to get values of inSamps[] */

nspdbIirl(&tapSt, &dlySt, inSamps, outSamps, 2000);
```

**Application Notes:** Call the `nsp?Iirl()` function to invoke either the arbitrary-order IIR filter (`nsp?IirlInit()`) or the biquad cascade structure (`nsp?IirlInitBq()`).

### Related Topics

| | |
|---|---|
| `IirlInit` | Initializes a low-level IIR filter. This function describes the filter as an arbitrary-order IIR filter (see page 8-93). |
| `IirlInitBq` | Initializes a low-level IIR filter. This function describes the filter as a cascade of biquads (second-order IIR sections, see page 8-93). |
| `IirlInitDlyl` | Initializes the delay line contents for a low-level IIR filter (see page 8-93). |

## IIR Filter Functions

The functions described in this section initialize an infinite impulse response (IIR) filter and perform the filtering function. They are intended for cyclic processing.

These functions provide a higher-level interface than the corresponding low-level IIR functions (see “Iirl” on page 8-97 for a description of `nsp?Iirl()`). In particular, they bundle the taps and delay line into a single state. Also, the IIR filter functions dynamically allocate memory for the taps and delay line; thus the arrays storing the taps and delay line values are not accessed after initialization and need not exist while the filter exists.

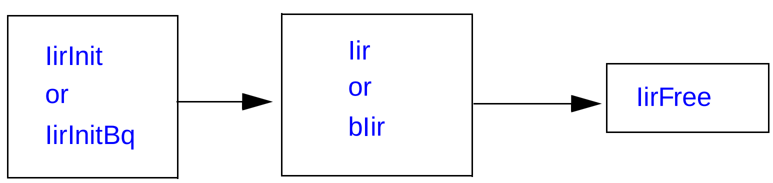
To initialize and use an IIR filter, follow this general scheme:

1. Call `nsp?IirInit()` to initialize the filter as an arbitrary order IIR filter, or call `nsp?IirInitBq()` to initialize the filter as a cascade of biquads.
2. Call `nsp?Iir()` to filter a single sample through an IIR filter or call `nsp?Iir()` repeatedly to filter consecutive samples one at a time. You can also call `nsp?bIir()` repeatedly to filter consecutive blocks of samples through an IIR filter.
3. After all filtering is complete, call `nspIirFree()` to release dynamic memory associated with the filter.

Real and complex filter coefficients can be mixed with real and complex input (that is, all four combinations are allowable). However, filter coefficients and input of different precision must not be mixed. It is the application's responsibility to call the correct function for the given type combination. This is not checked at compile time nor is it required to be checked at run-time.

Figure 8-6 illustrates the order of use of the IIR filter functions.

**Figure 8-6 Order of Use of the IIR Functions**



## IirInit, IirInitBq, IirFree

*Initializes an infinite impulse response filter.*

```
void nspsIirInit(NSPIirType iirType, const float *tapVals,
      int order, NSPIirState *statePtr);
void nspsIirInitBq(NSPIirType iirType, const float *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* real input, real taps; single precision */

void nspcIirInit(NSPIirType iirType, const SCplx *tapVals,
      int order, NSPIirState *statePtr);
void nspcIirInitBq(NSPIirType iirType, const SCplx *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* complex input, complex taps; single precision */
void nspscIirInit(NSPIirType iirType, const SCplx *tapVals,
      int order, NSPIirState *statePtr);
void nspscIirInitBq(NSPIirType iirType, const SCplx *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* real input, complex taps; single precision */

void nspcsIirInit(NSPIirType iirType, const float *tapVals,
      int order, NSPIirState *statePtr);
void nspcsIirInitBq(NSPIirType iirType, const float *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* complex input, real taps; single precision */

void nspdIirInit(NSPIirType iirType, const double *tapVals,
      int order, NSPIirState *statePtr);
void nspdIirInitBq(NSPIirType iirType, const double *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* real input, real taps; double precision */

void nspzIirInit(NSPIirType iirType, const DCplx *tapVals,
      int order, NSPIirState *statePtr);
void nspzIirInitBq(NSPIirType iirType, const DCplx *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* complex input, complex taps; double precision */
```

```
void nspdzIirInit(NSPIirType iirType, const DCplx *tapVals,
      int order, NSPIirState *statePtr);
void nspdzIirInitBq(NSPIirType iirType, const DCplx *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* real input, complex taps; double precision */

void nspzdIirInit(NSPIirType iirType, const double *tapVals,
      int order, NSPIirState *statePtr);
void nspzdIirInitBq(NSPIirType iirType, const double *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* complex input, real taps; double precision */

void nspIirFree(NSPIirState *statePtr);
void nspwIirInit(NSPIirType iirType, const float *tapVals,
      int order, NSPIirState *statePtr);
void nspwIirInitBq(NSPIirType iirType, const float *tapVals,
      int numQuads, NSPIirState *statePtr);
      /* real input, real taps; short integer */
```

| | |
|---|---|
| *iirType* | Specifies the filter structure to use with the filter description given. The value for *iirType* must currently be `NSP_IirDefault`. This argument is used by the `nsp?IirInit()` and `nsp?IirInitBq()` functions. |
| *numQuads* | The number of cascades of biquads (second-order IIR sections). The *numQuads* argument is used by the `nsp?IirInitBq()` function. |
| *order* | The order of the IIR filter. This argument is used by the `nsp?IirInit()` function. |
| *statePtr* | Pointer to the `NSPIirState` structure. |
| *tapVals* | Pointer to the array which stores the filter coefficients. |

## Description

The `nsp?IirInit()` and `nsp?IirInitBq()` functions initialize an infinite impulse response filter. They are intended for cyclic processing. The `nspIirFree()` function frees dynamic memory associated with an infinite impulse response filter.

Many combinations of real and complex input and filter coefficients are possible. This is indicated by the `s`, `c`, `sc`, `cs`, `d`, `z`, `dz`, `zd`, and `w` type codes following the `nsp` prefix in the function names above. For both of the functions, `nsp?IirInit()` and `nsp?IirInitBq()`, the allowed combinations of real and complex input and filter coefficients are described in Table 8-7.

**Table 8-7 Input and Filter Coefficient Combinations for nsp?IirInit() and nsp?IirInitBq() Functions**

| Type Codes | Input Type | Filter Coefficient Type | Output Type |
|---|---|---|---|
| `s` | `float` | `float` | `float` |
| `c` | `SCplx` | `SCplx` | `SCplx` |
| `sc` | `float` | `SCplx` | `SCplx` |
| `cs` | `SCplx` | `float` | `SCplx` |
| `d` | `double` | `double` | `double` |
| `z` | `DCplx` | `DCplx` | `DCplx` |
| `dz` | `double` | `DCplx` | `DCplx` |
| `zd` | `DCplx` | `double` | `DCplx` |
| `w` | `short` | `float` | `short` |

continued ☞

**`nsp?IirInit()`**. The `nsp?IirInit()` function initializes an arbitrary order IIR filter. The argument *`iirType`* selects the filter structure to use with the filter description given. The filter structure is the organization of delay elements, gain elements, and adders that make up the filter (common filter structures are, for example, “direct form 1,” “direct form 2,” and so on). Multiple structures can implement the same filter, but the contents of the delay line will have different meaning depending on the structure you choose to use. Both the choice of initialization function and the value of *`iirType`* combine to select the desired filter implementation. The value of *`iirType`* must currently be set to `NSP_IirDefault`, meaning that the library is free to use whichever filter structure is most natural.

The 2 * (*order* + 1) length array *tapVals[n]* specifies the filter coefficients as discussed for the low-level IIR function nsp?IirlInit(). See "IirlInit" on page 8-103 (for nsp?IirlInit()) for more information on how the filter coefficients are specified.

The delay line is initialized as in nsp?IirlInitDlyl(). See "IirlInitDlyl" on page 8-93 (for nsp?IirlInitDlyl()) for more information on how the delay line is initialized.

**nsp?IirInitBq().** The function nsp?IirInitBq() initializes an IIR filter defined by a cascade of biquads. The argument *iirType* describes the filter structure to use. As described above, this must be NSP_IirDefault. The 6 * *numQuads* length array *tapVals[n]* specifies the filter coefficients as described for the low-level IIR function nsp?IirlInitBq(). See "IirlInitBq" on page 8-103 (for nsp?IirlInitBq()) for more information on how the filter coefficients are specified.

The delay line is initialized in the same way as for the low-level IIR function nsp?IirlInitDlyl(). See "IirlInitDlyl" on page 8-93 (for nsp?IirlInitDlyl()) for more information on how the delay line is initialized.

**nspIirFree().** The nspIirFree() function frees all dynamic memory associated with a filter created by nsp?IirInit() or nsp?IirInitBq(). You should call nspIirFree() after the application has finished filtering with *statePtr*. After calling nspIirFree(), you should not reference *statePtr* again.

**Application Notes:** The contents of NSPIirState is implementation-dependent, but it does include a NSPIirTapState structure and a NSPIirDlyState structure. In addition, it includes a dynamically allocated array for the taps and the delay line. The integer flavor of IIR filter uses the float format for initializing the taps; the internal usage and representation of taps is implementation- and processor-dependent. For more information, see for the "Application Notes" on page 8-96 for the "IirlInit" and "IirlInitBq" sections (that is, for the low-level functions nsp?IirlInit() and nsp?IirlInitBq()).

### Related Topics

| | |
|---|---|
| `IirlInit` | Initializes a low-level IIR filter. This function describes the filter as an arbitrary-order IIR filter (see page 8-93). |
| `IirlInitBq` | Initializes a low-level IIR filter. This function describes the filter as a cascade of biquads (second-order IIR sections, see page 8-93). |

## Iir, bIir

*Filters a signal through an IIR filter.*

```
float nspsIir(NSPIirState *statePtr, float samp);
void nspsbIir(NSPIirState *statePtr, const float *inSamps,
float *outSamps, int numIters);
      /* real input, real taps; single precision */

SCplx nspcIir(NSPIirState *statePtr, SCplx samp);
void nspcbIir(NSPIirState *statePtr, const SCplx *inSamps,
SCplx *outSamps, int numIters);
      /* complex input, complex taps; single precision */

SCplx nspscIir(NSPIirState *statePtr, float samp);
void nspscbIir(NSPIirState *statePtr, const float *inSamps,
SCplx *outSamps, int numIters);
      /* real input, complex taps; single precision */

SCplx nspcsIir(NSPIirState *statePtr, SCplx samp);
void nspcsbIir(NSPIirState *statePtr, const SCplx *inSamps,
SCplx *outSamps, int numIters);
      /* complex input, real taps; single precision */

double nspdIir(NSPIirState *statePtr, double samp);
void nspdbIir(NSPIirState *statePtr, const double *inSamps,
double *outSamps, int numIters);
      /* real input, real taps; double precision */

DCplx nspzIir(NSPIirState *statePtr, DCplx samp);
void nspzbIir(NSPIirState *statePtr, const DCplx *inSamps,
DCplx *outSamps, int numIters);
      /* complex input, complex taps; double precision */

DCplx nspdzIir(NSPIirState *statePtr, double samp);
void nspdzbIir(NSPIirState *statePtr, const double *inSamps,
      DCplx *outSamps, int numIters);
      /* real input, complex taps; double precision */
```

```
DCplx nspzdIir(NSPIirState *statePtr, DCplx samp);
void nspzdbIir(NSPIirState *statePtr, const DCplx *inSamps,
DCplx *outSamps, int numIters);
      /* complex input, real taps; double precision */

short nspwIir(NSPIirState *statePtr, short samp, int ScaleMode,
      int *ScaleFactor);
void nspwbIir(NSPIirState *statePtr, const short *inSamps,
      short *outSamps, int numIters, int ScaleMode,
      int *ScaleFactor);
      /* real input, real taps; short integer */
```

| | |
|---|---|
| *inSamps* | Pointer to the array containing the input samples for the `nsp?bIir()` function. |
| *numIters* | The number of samples to be filtered by the `nsp?bIir()` function. |
| *outSamps* | Pointer to the array containing the output samples for the `nsp?bIir()` function. |
| *samp* | The input sample for the `nsp?bIir()` function. |
| *statePtr* | Pointer to the `NSPIirState` structure. |
| *ScaleMode*, *ScaleFactor* | Refer to "Scaling Arguments" in Chapter 1. |

## Description

The `nsp?Iir()` and `nsp?bIir()` functions filter samples through an IIR filter. The different data types of functions correspond to different combinations of real/complex taps and delay line, as described by the table under `nsp?IirInit()`. The data type of the function used here must match the data type of the function used for initialization.

**Previous Tasks:** You must initialize the `NSPIirState` structure pointed to by *statePtr* by calling either `nsp?IirInit()` or `nsp?IirInitBq()`.

**`nsp?Iir()`.** The `nsp?Iir()` function filters a single sample *samp* through an IIR filter and returns the result.

**nsp?bIir()**. The nsp?bIir() function filters a block of *numIters* samples in the array *inSamps[n]* through an IIR filter and returns the result in the array *outSamps[n]*.

Example 8-14 illustrates using nsp?IirInit() to initialize an arbitrary-order IIR filter and then using nsp?Iir() to filter the samples.

### Example 8-14 Arbitrary Order IIR Filtering With the nsp?IirInit() and nsp?Iir() Functions

```
/*
 * arbitrary order
 * IIR filtering
 */
NSPIirState  iirSt;
double       taps[10], xval, yval;
int          i;
/* insert code here to initialize taps */
nspdIirInit(NSP_IirDefault, taps, 4, &iirSt);
for (i=0; i<2000; i++) {
   xval = /* insert code here to get the next value of x */;
   yval = nspdIir(&iirSt, xval);
   /* yval has the output sample */
}
```

Example 8-15 illustrates using nsp?IirInitBq() to initialize an IIR filter as a cascade of biquads and then using nsp?Iir() to filter the samples.

**Example 8-15 Cascaded Biquad Filtering With the nsp?IirInitBq() and nsp?Iir() Functions**

```
/*
 * cascaded biquad
 * IIR filtering
 */
NSPIirState  iirSt;
double       taps[30], xval, yval;
int          i;
/* insert code here to initialize taps */
nspdIirInitBq(NSP_IirDefault, taps, 5, &iirSt);
for (i=0; i<2000; i++) {
   xval = /* insert code here to get the next value of x */;
   yval = nspdIir(&iirSt, xval);
   /* yval has the output sample */
}
```

## Related Topics

| | |
|---|---|
| `bIirl` | Filters a block of samples through a low-level IIR filter and returns the result (see page 8-97). |
| `IirInit` | Initializes an IIR filter. This function describes the filter as an arbitrary-order IIR filter (see page 8-103). |
| `IirInitBq` | Initializes an IIR filter. This function describes the filter as a cascade of biquads (second-order IIR sections, see page 8-103). |
| `Iirl` | Filters a single sample through a low-level IIR filter and returns the result (see page 8-97). |

## Median Filter functions

Median filters are nonlinear rank-order filters based on replacing each element of the input 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 SPL median function implementation, the mask is always centered at the input element for which the median value is computed. You can use an even mask size in function calls as well, but internally it will be changed to odd by subtracting 1. Another specific feature of the median function implementation in SPL is that elements outside the input vector, which are needed to determine the median value for “border” elements, are set to be equal to the corresponding edge element of the input vector, i.e. are padded by cloning the edge element.

## bMedianFilter1

*Computes median values for each input vector element (in-place)*

```
void nspsbMedianFilter1( float *vec, int len,
   int masksize);
        /* real values; single precision */
void nspdbMedianFilter1( double *vec, int len,
   int masksize);
        /* real values; double precision */
void nspwbMedianFilter1( short *vec, int len,
   int masksize );
        /* real values; short integer */
```

| | |
|---|---|
| *vec* | The pointer to the input vector |
| *len* | The input vector length (number of elements) |

*masksize* The mask length, must be a positive integer. If an even value is specified, the function subtracts 1 and uses the corresponding odd value for median filtering.

## bMedianFilter2

*Computes median values for each input vector element, and writes the result to the output vector.*

```
void nspsbMedianFilter2( const float *src, float *dst,
   int len, int masksize );
        /* real values; single precision */
void nspdbMedianFilter2( const double *src, double *dst,
   int len, int masksize );
        /* real values; double precision */
void nspwbMedianFilter2( const short *src, short *dst,
   int len, int masksize );
        /* real values; short integer */
```

*src* The pointer to the input vector

*dst* The pointer to the output vector

*len* The vector length (number of elements). The same for input and output vectors.

*masksize* The mask length, must be a positive integer. If an even value is specified, the function subtracts 1 and uses the corresponding odd value for median filtering.

### Example 8-16 Using the Median Filter Function

```
#define nsp_UsesMedian
#include <nsp.h>
#define LEN 12
int main( void ) {
   short y[LEN];
   short x[LEN] = { 1,2,3,15,5,6,7,8,15,15,15,16 };
   nspwbMedianFilter2( x,y,LEN,3 );
   return NSP_StsOk == nspGetErrStatus();
}
/// y ={ 1 2 3 5 6 6 7 8 15 15 15 16  }
```

Input signal increases uniformly at the beginning with the single spike impulse reaching 15, and steeply raises in magnitude towards the end of the input vector. The median filter removes the impulsive spike without smoothing the further sharp signal increase. The use of averaging filter instead would distort the signal by keeping trace of the spike noise impulse, together with smoothing the final signal gain.

# Convolution Functions

9

Library
function lists

This chapter describes the Signal Processing Library functions that perform convolution operations. Convolution is an operation used to define an output signal from any linear time-invariant (LTI) processor in response to any input signal [Lyn89].

The convolution operation is performed for one- and two-dimensional signals.

## One-Dimensional Convolution

The Signal Processing Library provides an `nsp?Conv()` function to perform finite linear convolution of two sequences for one-dimensional signals.

## Conv

*Performs finite, linear convolution of two sequences.*

```
void nspsConv(const float *x, int xLen, const float *h, int hLen,
      float *y); /* real first signal, real second signal;
                    single precision */
void nspcConv(const SCplx *x, int xLen, const SCplx *h, int hLen,
      SCplx *y); /* complex first signal, complex second signal;
                    single precision */
void nspscConv(const float *x, int xLen, const SCplx *h, int hLen,
      SCplx *y); /* real fist signal, complex second signal;
                    single precision */
void nspcsConv(const SCplx *x, int xLen, const float *h, int hLen,
      SCplx *y); /* complex first signal, real second signal;
                    single precision */
void nspdConv(const double *x, int xLen, const double *h, int hLen,
      double *y); /* real first signal, real second signal;
                    double precision */
void nspzConv(const DCplx *x, int xLen, const DCplx *h, int hLen,
      DCplx *y); /* complex first signal, complex second signal;
                    double precision */
void nspdzConv(const double *x, int xLen, const DCplx *h, int hLen,
      DCplx *y); /* real first signal, complex second signal;
                    double precision */
void nspzdConv(const DCplx *x, int xLen, const double *h, int hLen,
      DCplx *y); /* complex first signal, real seconf signal;
                    double precision */
void nspwConv(const short *x, int xLen, const short *h, int hLen
      short *y, int ScaleMode, int *ScaleFactor);
      /* real first signal, real second signal; short integer */
```

| | |
|---|---|
| `h`, `x` | Pointers to the arrays to be convolved. |
| `hLen` | Number of samples in the array `h[n]`. |
| `xLen` | Number of samples in the array `x[n]`. |

*y* Pointer to the array which stores the result of the convolution.

*ScaleMode*, *ScaleFactor* Refer to "Scaling Arguments" in Chapter 1.

## Discussion

The different types of the nsp?Conv() function correspond to the different real/complex combinations of the input signals. This is indicated by the s, c, sc, cs, d, z, dz, and zd type codes following the nsp prefix in the function names above. The allowed combinations of real and complex signals are described in Table 9-1.

**Table 9-1 Signal Types Combinations for nsp?Conv() Function**

| Type Codes | First Signal Type | Second Signal Type | y(n) (or output) Type |
|---|---|---|---|
| s | float | float | float |
| c | SCplx | SCplx | SCplx |
| sc | float | SCplx | SCplx |
| cs | SCplx | float | SCplx |
| d | double | double | double |
| z | DCplx | DCplx | DCplx |
| dz | double | DCplx | DCplx |
| zd | DCplx | double | DCplx |

**NOTE.** *The data type of the function used here must match the data type of the function used for* nsp?Fir().

The nspscConv() and nspcsConv() functions and the nspdzConv() and nspzdConv() functions are essentially identical and are included for convenience.

The `nsp?Conv()` function performs single-rate convolution. The `xLen`-length array `x` is convolved with the `hLen`-length array `h` to produce an `xLen` + `hLen` - 1 length array `y`. The argument names `x`, `h`, and `y` are chosen to suggest FIR filtering. The result of the convolution is defined as follows:

$$y[n] = \sum_{k=0}^{hLen-1} h[k] \cdot x[n-k], \quad 0 \le n < xLen + hLen - 1$$

This finite-length convolution is related to infinite-length by:

$$x'(n) = \begin{cases} x[n] & , 0 \le n < xLen \\ 0 & , otherwise \end{cases}$$

$$h'(n) = \begin{cases} h[n] & , 0 \le n < hLen \\ 0 & , otherwise \end{cases}$$

$y'(n) = x'(n) * h'(n)$

In the above equations, `x'(n)` and `h'(n)` are the zero-padded (infinite-length) versions of `x(n)` and `h(n)`; `y'(n)` is the infinite-length output version of `y(n)`.

Then `y'(n)` is zero everywhere except over:

$y'[n] = y(n), 0 \le n < xLen + hLen - 1$

Example 9-1 shows the code for the convolution of two vectors using `nsp?Conv()`.

### Example 9-1 Using nsp?Conv() to Convolve Two Vectors

```
/* convolve two vectors */
double  x[32], h[16];
double  y[47]; /* 32 + 16 - 1 = 47 */

/* insert code here to put data in x and h */

nspdconv(x, 32, h, 16, y);
/* y has the finite convolution of x and h */
```

## Two-Dimensional Convolution

For two-dimensional signals, the Signal Processing Library provides two functions: `nsp?Conv2D()` and `nsp?Filter2D()`. The functions are basically identical for image processing except that `nsp?Filter2D()` stores the result of the convolution in the array that is the same as input array, while `nsp?Conv2D()` stores the result of the convolution in a new output array.

## Conv2D

*Performs finite, linear convolution of two two-dimensional signals.*

```
void nspsConv2D (float *x, int xCols, int xRows, float *h,
      int hCols, int hRows, float *y);
      /* real values; single precision */
void nspdConv2D (double *x, int xCols, int xRows, double *h,
      int hCols, int hRows, double *y);
      /* real values; double precision */
void nspwConv2D (short *x, int xCols, int xRows, short *h,
      int hCols, int hRows, short *y, int ScaleMode,
      int *ScaleFactor);
      /* real values; short integer */
```

| | |
|---|---|
| `h`, `x` | Pointers to the two-dimensional arrays to be convolved. |
| `hCols, hRows` | Dimensions of the `h[n, m]` array. |
| `xCols, xRows` | Dimensions of the `x[n, m]` array. |
| `y` | The array which stores the result of the convolution. |
| `ScaleMode`, `ScaleFactor` | Refer to “Scaling Arguments” in Chapter 1. |

## Discussion

The `nsp?Conv2D()` function performs a convolution of two-dimensional signals. The *xCols* by *xRows* array *x* is convolved with the *hCols* by *hRows* array *h* to produce an *xCols+hCols-1* by *xRows+hRows-1* array *y*.

The result of the convolution is defined as follows:

$$y[n,m] = \sum_{k=0}^{hRows-1} \sum_{j=0}^{hCols-1} h(j,k) \cdot x(n-j, m-k)$$

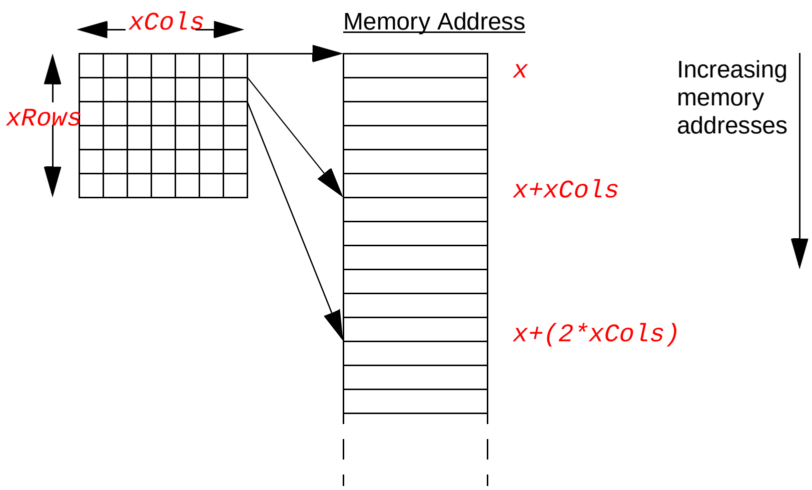
$$0 \le n < xCols + hCols - 1, \quad 0 \le m < xRows + hRows - 1$$

$$x(n,m) = \begin{cases} x[n,m] & 0 \le n < xCols, \quad 0 \le m < xRows \\ 0 & otherwise \end{cases}$$

In the above expressions, *x[n, m]* is a shorthand for *x[n + m*xCols]*, *h[n, m]* is a shorthand for *h[n + m*hCols]*, and *y[n, m]* is a shorthand for *y[n + m*(xCols + hCols - 1)]*.

This function treats consecutive array element addresses as horizontally adjacent samples (that is, they are in the same row), and elements that are a distance of *xCols* (or *hCols*) apart as vertically adjacent (that is, they are in the same column) as shown in Figure 9-1.

**Figure 9-1 Consecutive Array Element Addresses**




## Filter2D

*Filters a two-dimensional signal.*

```
void nspsFilter2D (const float *x, int xCols, int xRows,
      const float *h, int hCols, int hRows, float *y);
      /* real values for line and taps; single precision */
void nspdFilter2D (const double *x, int xCols, int xRows,
      const double *h, int hCols, int hRows, double *y);
      /* real values for line and taps; double precision */
void nspwFilter2D (const short *x, int xCols, int xRows,
      const short *h, int hCols, int hRows, short *y,
      int ScaleMode, int *ScaleFactor);
      /* real values for line and taps; short integer */
```

| | |
|---|---|
| `x` | Two-dimensional array (input image) to be filtered. The size of the input image is `xCols` by `xRows` |
| `h` | Two-dimensional array of filter coefficients. The filter size is `hCols` by `hRows`. |
| `y` | The array which stores the result of the filtering (output image). The size of the input image is `xCols` by `xRows`. |
| `ScaleMode`, `ScaleFactor` | Refer to "Scaling Arguments" in Chapter 1. |

## Discussion

The `nsp?Filter2D()` function filters two-dimensional signal `x` using coefficients in the array `h`. It is intended for image processing and is identical to the function `nsp?conv2D()`, except that it returns an output array that is the same as the input array. This prevents image dimensions from being effected by filtering.

The arguments `xCols` and `xRows` specify the width and height, respectively, of the input image `x` and output image `y`. The arguments `hCols` and `hRows` specify the width and height, respectively, of the coefficient matrix `h`. The input and coefficients are convolved as follows:

$$y[n,m] = \sum_{j=0}^{hCols-1}\sum_{k=0}^{hRows-1} h[j,k]\cdot x\left(n-j+\left\lfloor\frac{hCols}{2}\right\rfloor, m-k+\left\lfloor\frac{hRows}{2}\right\rfloor\right)$$

$$0 \le n < xCols,\quad 0 \le m < xRows$$

$$x(n,m) = \begin{cases} x[n,m] & 0 \le n < xCols,\quad 0 \le m < xRows \\ 0 & otherwise \end{cases}$$

In the above expressions, `x[n, m]` is a shorthand for `x[n + m*xCols]`, `h[n, m]` is a shorthand for `h[n + m*hCols]`, and `y[n, m]` is a shorthand for `y[n + m*xCols]` .

This function treats consecutive array element addresses as horizontally adjacent samples (that is, they are in the same row), and elements that are a distance of `xCols` (or `hCols`) apart as vertically adjacent (that is, they are in the same column) as discussed for `nsp?Conv2D`.

# Wavelet Functions

# 10

Library function lists

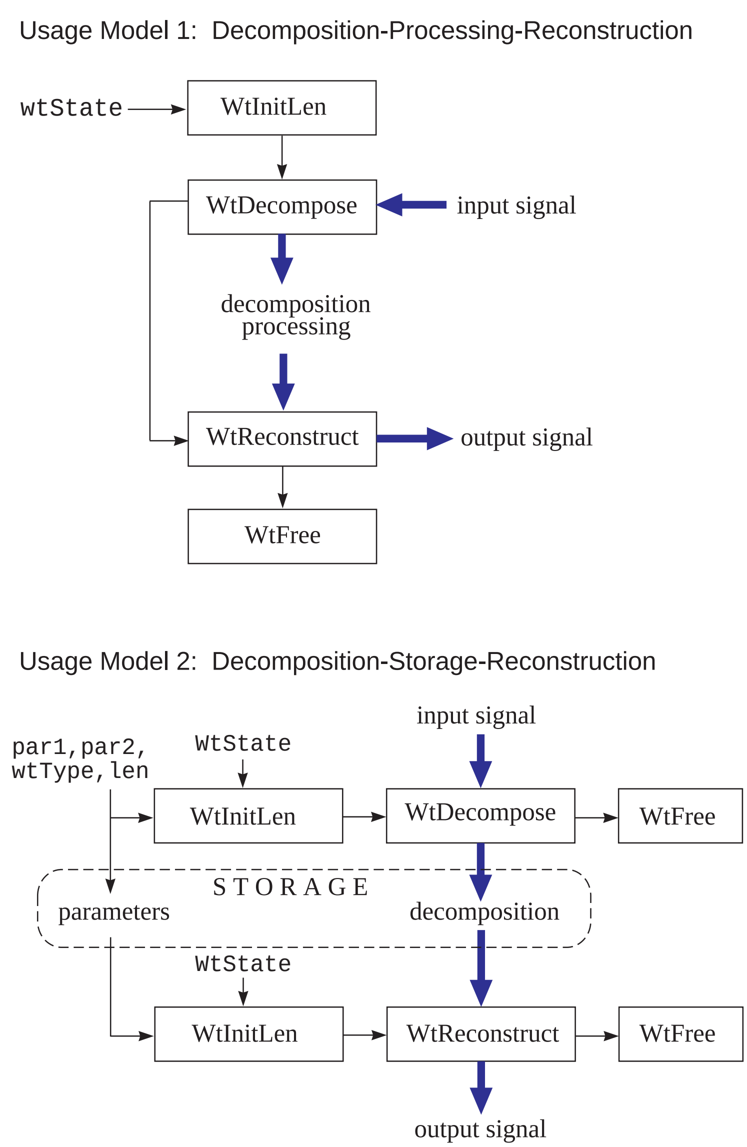
This chapter describes the wavelet functions in the Signal Processing Library. The importance of wavelet-based representation of signals is primarily due to its capability of "zooming." It helps you observe rapidly changing functions by using shorter time windows, and low-frequency components by using longer windows. (This is different from the Fourier transform, in which the bases are characterized by an infinite time window.)

The wavelet functions in the SPL provide the necessary building blocks for wavelet-based decomposition and reconstruction of signals. To take advantage of these functions, even basic understanding of wavelet concepts is sufficient. The SPL wavelet functions support most commonly used wavelet bases, both orthogonal and biorthogonal; you can also specify wavelets by setting the corresponding filters directly. The traditional dyadic-tree-structured decompositions are implemented.

The library includes the following functions for wavelet decomposition and reconstruction of signals:

- the functions `nsp?WtInit()`, `nsp?WtInitLen()`, `nsp?WtInitUserFilter()`, `nsp?WtGetState()`, and `nsp?WtSetState()` that perform initialization operations, in particular, create filters to be used in the decomposition and reconstruction, set and retreive the wavelet parameters
- the `nsp?WtDecompose()` function that performs wavelet-based decomposition of signals
- the `nsp?WtReconstruct()` function that performs wavelet-based reconstruction of signals.
- the `nspWtFree()` function that frees the memory used by the above functions for internal purposes.

**Figure 10-1 Order of Use of the Wavelet Functions**



All information about wavelets needed for the decomposition and reconstruction of data is contained in the `NSPWtState` structure. This information is in a computation-oriented format; therefore, there are special functions for initializing the `NSPWtState` structure and setting all its parameters: the wavelet type, the wavelet order, the filter order, the taps, and so on.

Figure 10-1 shows the order of use of the wavelet transform functions. All these functions are described in the sections that follow.

## WtInit

*Initializes the NSPWtState structure for transforming signals of length $2^N$.*

```
void nspsWtInit(int par1, int par2, int dataOrder, int level,
NSPWtState *wtState, int wtType);
            /* real values; single precision */
void nspdWtInit(int par1, int par2, int dataOrder, int level,
NSPWtState *wtState, int wtType);
            /* real values; double precision */
void nspwWtInit(int par1, int par2, int dataOrder, int level,
NSPWtState *wtState, int wtType);
            /* real values; short integer */
```

| | |
|---|---|
| *par1, par2* | The wavelet parameters (see *Discussion* below). |
| *dataOrder* | The logarithm $\log_2 L$, where $L$ is the length of data. $L$ must be a power of 2 so that $\log_2 L$ is integer. |
| *level* | The level of decomposition (see *Discussion* below). |
| **wtState* | The structure that stores all information needed for wavelet decomposition and reconstruction. |
| *wtType* | The type of wavelet; can be one of the following:<br>`NSP_Haar`<br>`NSP_Daublet`<br>`NSP_Symmlet` |

NSP_Coiflet
NSP_Vaidyanathan
NSP_BSpline
NSP_BSpline_Dual
NSP_LinSpline
NSP_QuadSpline

## Discussion

The `nsp?WtInit()` function initializes the `NSPWtState` structure according to the specified *par1*, *par2*, *dataOrder*, *level*, and *wtType* arguments.

The *par1* and *par2* arguments have the following meaning:

for *wtType* = `NSP_Daublet` (Daubechies wavelets)

*par1* is the order of the wavelet (1 to 10) and
*par2* is dummy;

for *wtType* = `NSP_Coiflet` (Coifman orthonormal filters)

*par1* is the filter length parameter (1 to 5) and
*par2* is dummy;

for *wtType* = `NSP_Symmlet`,

*par1* is the order of the wavelet (1 to 7) and
*par2* is dummy;

for *wtType* = `NSP_BSpline` and *wtType* = `NSP_BSplineDual`,
*par1* is the order of the wavelet analysis basis,
*par2* is the order of the dual (synthesis) basis.

for `NSP_Haar`, `NSP_Vaidyanathan`, `NSP_LinSpline`, and `NSP_QuadSpline`, both *par1* and *par2* are dummy.

The following (*par1*, *par2*) argument pairs are admissible for `NSP_BSpline` and `NSP_BSplineDual`:

(1,1), (1,3), (1,5) for box splines;
(2,2), (2,4), (2,6), (2,8) for linear splines;
(3,1), (3,3), (3,5), (3,7), (3,9) for quadratic splines.

The case `NSP_LinSpline` is equivalent to `NSP_BSpline` with the parameters (2,2), and the case `NSP_QuadSpline` is equivalent to `NSP_BSpline` with the parameters (3,3).

The *level* argument specifies the decomposition level, that is, the number of decomposition iterations to be performed. The value of *level* ranges from 0 (no decomposition) to *dataOrder* - 1 (full decomposition).

Consider the following example: if the number of elements in the data vector is 16, *dataOrder* is 4. If you specify *level* = 2, two iterations of decomposition will be performed (see Figure 10-1).

You can also specify wavelets directly by using the corresponding filters. Such wavelets have *wtType* = `NSP_WtByFilter`. You cannot call the `WtInit()` function with this value of `wtType`; use the `WtSetState()` function instead.

## Related Topics

| | |
|---|---|
| `WtGetState` | Returns the wavelet and filter parameters of the `NSPWtState` structure (see page 10-11). |
| `WtSetState` | Sets the wavelet and filter parameters (see page 10-11). |
| `WtDecompose` | Decomposes the input signal into wavelet series (see page 10-15). |
| `WtReconstruct` | Reconstructs the signal from the wavelet decomposition (see page 10-16). |
| `WtFree` | Frees the memory used by the wavelet functions for internal purposes (see page 10-17). |

## WtInitLen

*Initializes the NSPWtState structure for signals of an arbitrary length.*

```
void nspsWtInitLen(int par1, int par2, int len, int level,
NSPWtState *wtState, int wtType, int *len_dec);
            /* real values; single precision */
void nspdWtInitLen(int par1, int par2, int len, int level,
NSPWtState *wtState, int wtType, int *len_dec);
            /* real values; double precision */
void nspwWtInitLen(int par1, int par2, int len, int level,
NSPWtState *wtState, int wtType, int *len_dec);
            /* real values; short integer */
```

| | |
|---|---|
| *par1, par2* | The wavelet parameters (see *Discussion* below). |
| *len* | The signal length (may or may not be a power of 2). |
| *level* | The level of decomposition (see *Discussion* below). |
| *wtState* | Pointer to the structure that stores all information needed for wavelet decomposition and reconstruction. |
| *wtType* | The type of wavelet; can be one of the following:<br>`NSP_Haar`<br>`NSP_Daublet`<br>`NSP_Symmlet`<br>`NSP_Coiflet`<br>`NSP_Vaidyanathan`<br>`NSP_BSpline`<br>`NSP_BSplineDual`<br>`NSP_LinSpline`<br>`NSP_QuadSpline` |
| *len_dec* | Pointer to an integer variable which will store the actual decomposition signal length. |

## Discussion

The `nsp?WtInitLen()` function initializes the `NSPWtState` structure according to the specified *`par1`*, *`par2`*, *`dataOrder`*, *`level`*, and *`wtType`* arguments. The *`par1`*, *`par2`*, *`level`*, and *`wtType`* arguments have the same meaning as the corresponding arguments of `nsp?WtInit()`; see page 10-3.

Unlike `nsp?WtInit()`, the function `nsp?WtInitLen()` works for signals of an arbitrary length *`len`* > 1. The lengths of both the original and reconstructed signals are equal to *`len`*. Note, however, that the *decomposition length may differ from the source signal length* when *`len`* is not proportional to $2^{level}$. The output parameter *`len_dec`* is a pointer to a variable which will store the actual decomposition signal length.

The value of *`level`* must satisfy the condition $2^{level} < 2$ *`len`*. For example, if *`level`* = 2 and the input vector length is *`len`* = 10, then two iterations of decomposition will be performed (see Example on page 10-18).

## Related Topics

| | |
|---|---|
| `WtGetState` | Returns the wavelet and filter parameters of the `NSPWtState` structure (see page 10-11). |
| `WtSetState` | Sets the wavelet and filter parameters (see page 10-11). |
| `WtDecompose` | Decomposes the input signal into wavelet series (see page 10-15). |
| `WtReconstruct` | Reconstructs the signal from the wavelet decomposition (see page 10-16). |
| `WtFree` | Frees the memory used by the wavelet functions for internal purposes (see page 10-17). |

## WtInitUserFilter

*Initializes the NSPWtState structure using the user's filter bank.*

```
NSPStatus nspsWtInitUserFilter(float *tap_filt[4], int len_filt[4],
   int ofs_filt[4], int len, int level, NSPWtState *wtState,
   int *len_dec));
             /* real values; single precision */
NSPStatus nspdWtInitUserFilter(double *tap_filt[4], int len_filt[4],
   int ofs_filt[4], int len, int level, NSPWtState *wtState,
   int *len_dec));
             /* real values; double precision */
NSPStatus nspwWtInitUserFilter(float *tap_filt[4], int len_filt[4],
   int ofs_filt[4], int len, int level, NSPWtState *wtState,
   int *len_dec));
             /* real values; short integer */
```

| | |
|---|---|
| *tap_filt* | Array of pointers to arrays of filter taps. |
| *len_filt* | Array of filter lengths. |
| *ofs_filt* | Array of filter offsets. |
| *len* | The length of the signal vector. |
| *level* | The level of decomposition. |
| *wtState* | Pointer to the structure that contains the data for wavelet decomposition and reconstruction. |
| *len_dec* | Pointer to an integer variable which will store the actual decomposition signal length. |

### Discussion

You can initialize the wavelet transform functions to perform decomposition and reconstruction using any appropriate filters with finite impulse response. The `nsp?WtInitUserFilter()` function initializes the `NSPWtState` structure using your own filter bank (see Example 10-1). This initialization requires the arrays *tap_filt*[4] (pointers to arrays of filter taps), *len_filt*[4] (filter lengths), *ofs_filt*[4] (filter offsets). Elements of these arrays correspond to the following filters:

0: decomposition low-pass filter;
1: decomposition high-pass filter;
2: reconstruction low-pass filter;
3: reconstruction high-pass filter.

Each of the offset values *ofs_filt*[*i*] must be zero or a positive integer less than the corresponding filter length. The offsets specify that, in the periodic filtering mode, (*len_filt*[*i*]-*ofs_filt*[*i*]-1) elements will be added at the beginning of the signal data.

Elements of the array *tap_filt* point to four vectors of filter taps. You can provide the tap values by using the following code:

```
const double fltLowDecom[12]  = {...};
const double fltHighDecom[12] = {...};
const double fltLowRecon[12]  = {...};
const double fltHighRecon[12] = {...};
double *tap_filt[4] = { fltLowDecom, fltHighDecom,
                        fltLowRecon, fltHighRecon };
nspdWtInitUserFilter(tap_filt, ...);
```

Each filter vector will be copied to the memory allocated by the function `nsp?WtInitUserFilter()`. When you call the function `nsp?WtFree()`, it will free this memory.

## Related Topics

| | |
|---|---|
| `WtGetState` | Returns the wavelet and filter parameters of the `NSPWtState` structure (see page 10-11). |
| `WtSetState` | Sets the wavelet and filter parameters (see page 10-11). |
| `WtDecompose` | Decomposes the input signal into wavelet series (see page 10-15). |
| `WtReconstruct` | Reconstructs the signal from the wavelet decomposition (see page 10-16). |
| `WtFree` | Frees the memory used by the wavelet functions for internal purposes (see page 10-17). |

### Example 10-1 Initializing Wavelet Transforms with Beylkin Filters

```
/* Beylkin wavelet filters (18 taps) */
int   lenFlt[4] = {18, 18, 18, 18};
/* offset value pointing to maximum of absolute value
    of low-pass decomposition filter taps */
int   ofsFlt[4] = { 2,  2, 15, 15};
/* taps of low-pass decomposition filter */
float lowDecom[18]  =    {   7.021978E-2,  2.999656E-1,
 4.948512E-1,  3.179988E-1, -7.843766E-2, -1.870278E-1,
 1.902139E-2,  1.099825E-1, -1.238904E-2, -6.260980E-2,
 1.391577E-2,  3.034647E-2, -1.234637E-2, -1.015816E-2,
 7.099643E-3,  1.049512E-3, -1.934667E-3,  4.528915E-4};
NSPWtState wtState;
int   i, j;
int   len_dec;
float highDecom[18]; /* high-pass decomposition  filter taps */
float lowRecon[18];  /* low-pass  reconstruction filter taps */
float highRecon[18]; /* high-pass reconstruction filter taps */
float *filtAll[4] = { lowDecom, highDecom, lowRecon, highRecon };

 /* given the above low-pass decomposition filter taps,
    compute the taps for all other filters
    to form an orthogonal wavelet basis          */

for(i = 0, j = 17; i < 18; i+=2, j-=2)
    highDecom[i] =   lowDecom[j];
for(i = 1, j = 16; i < 18; i+=2, j-=2)
    highDecom[i] = - lowDecom[j];
for(i = 0, j = 17; i < 18; i++, j--) {
    lowRecon[i]  = 2.0 * lowDecom[j];
    highRecon[i] = 2.0 * highDecom[j];
}
/* initialization of wtState structure */
nspsWtInitUserFilter(filtAll, lenFlt, ofsFlt,
                        110, 3, &wtState, &len_dec);

/* here you can perform wavelet decomposition/reconstruction */

nspWtFree(&wtState);
```

## WtGetState
## WtSetState

*Returns or sets all wavelet parameters in the NSPWtState structure.*

```
void nspsWtGetState(NSPWtState *wtState, int *wtType, int *par1, int
*par2, int *dataOrder, int *level, float **fTaps, int *fLen,
int *fOffset);   /* real values; single precision */
void nspdWtGetState(NSPWtState *wtState, int *wtType, int *par1, int
*par2, int *dataOrder, int *level, double **fTaps, int *fLen,
int *fOffset);   /* real values; double precision */
void nspwWtGetState(NSPWtState *wtState, int *wtType, int *par1, int
*par2, int *dataOrder, int *level, float **fTaps, int *fLen,
int *fOffset);   /* real values; short integer */
void nspsWtSetState(NSPWtState *wtState, int wtType, int par1, int
par2, int dataOrder, int level, const float **fTaps, const int
*fLen, const int *fOffset);   /* real values; single precision */
void nspdWtSetState(NSPWtState *wtState, int wtType, int par1, int
par2, int dataOrder, int level, const double **fTaps, const int
*fLen, const int *fOffset);   /* real values; double precision */
void nspwWtSetState(NSPWtState *wtState, int wtType, int par1, int
par2, int dataOrder, int level, const float **fTaps, const int
*fLen, const int *fOffset);   /* real values; short integer */
```

*wtState* — The structure that stores all information needed for wavelet decomposition and reconstruction.

*wtType* — The type of wavelet; can be one of the following:
`NSP_Haar`
`NSP_Daublet`
`NSP_Symmlet`
`NSP_Coiflet`
`NSP_Vaidyanathan`
`NSP_BSpline`
`NSP_BSpline_Dual`

| | |
|---|---|
| | `NSP_LinSpline`<br>`NSP_QuadSpline`<br>`NSP_WtByFilter` |
| *`par1, par2`* | The wavelet parameters (see *Discussion* of `WtInit`). |
| *`dataOrder`* | The logarithm $\log_2 L$, where $L$ is the length of data. $L$ must be a power of 2 so that $\log_2 L$ is integer. |
| *`level`* | The last level of decomposition (see *Discussion* of `WtInit`). |
| *`*fTaps`* | Pointers to taps of the analysis low-pass filter, analysis high-pass filter, synthesis low-pass filter, and synthesis high-pass filter (see *Discussion* below). |
| *`fLen`* | The array of lengths of these filters. |
| *`fOffset`* | The array of offsets for these filters. Offsets are nonnegative integers less than the corresponding filter lengths (see *Discussion* below).<br>In the *`fLen`* and *`fOffset`* arrays,<br>0th elements correspond to the low-pass analysis filter,<br>1st elements to high-pass analysis filter,<br>2nd elements to the low-pass synthesis filter, and<br>3rd elements to the high-pass synthesis filter. |

## Discussion

The `nsp?WtSetState()` function sets all wavelet parameters in the `NSPWtState` structure. The `nsp?WtGetState()` function returns the current values of these parameters (including all applicable filter taps and lengths).

If *`wtType`* = `NSP_WtByFilter`, both *`par1`* and *`par2`* are dummy. For more information about the *`par1`* and *`par2`* arguments, see *Discussion* of the `nsp?WtInit()` function.

The *`level`* argument specifies the decomposition level, that is, the number of decomposition iterations to be performed. The value of *`level`* ranges from 0 (no decomposition) to *`dataOrder`* `-` 1 (full decomposition); see Figure 10-2.

If the wavelet type is `NSP_WtByFilter`, you must specify the taps lengths and offsets for all filters when calling the `WtSetState()` function. To better understand the meaning of *fLen* and *fOffset* parameters, consider the following two equivalent forms of the wavelet filter equation:

$$y[n] = \sum_{i=-K_1}^{K_2} x[n-i] \cdot h[i] \qquad (K_1 > 0,\ K_2 > 0)$$

$$y[n] = \sum_{k=0}^{fLen-1} x[n + fOffset - k] \cdot fTaps[k]$$

Here $x[n]$ is the input signal, $y[n]$ is the output signal, $fLen = K_1 + K_2 + 1$ is the filter length, $fOffset = K_1$, and therefore $fTaps[0] = h[-K_1]$, $fTaps[fLen-1] = h[K_2]$. SPL wavelet functions use the second form of the equation, which is more convenient in computation.

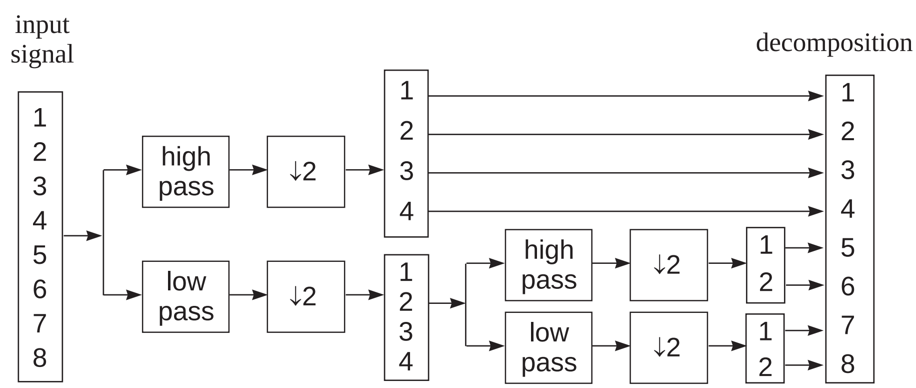
For wavelet types other than `NSP_WtByFilter`, the filters are determined internally from the *par1* and *par2* arguments.

## Related Topics

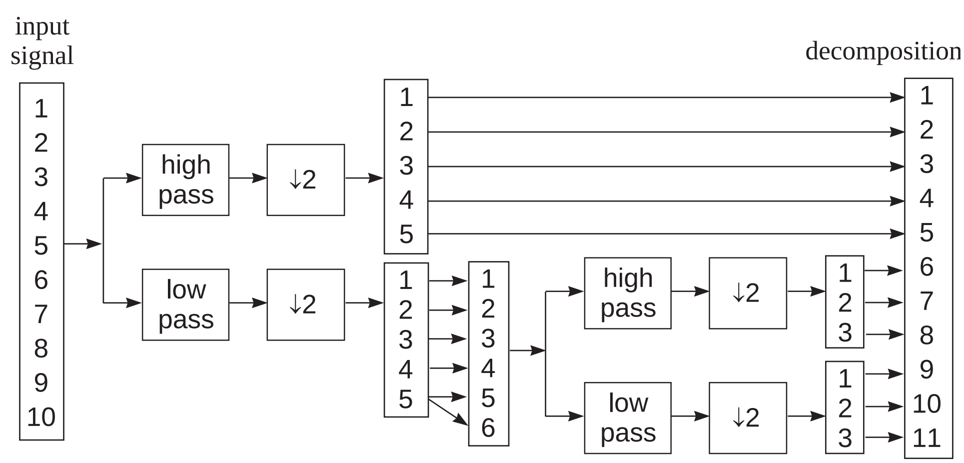
| | |
|---|---|
| `WtInit` | Initializes the `NSPWtState` structure (see page 10-3). |
| `WtDecompose` | Decomposes the input signal into wavelet series (see page 10-15). |
| `WtReconstruct` | Reconstructs the signal from the wavelet decomposition (see page 10-16). |
| `WtFree` | Frees the memory used by the wavelet functions for internal purposes (see page 10-17). |

**Figure 10-2 Wavelet Decomposition Scheme**

Signal of Length $2^N$ (length = 8; 2 levels)




Signal of an Arbitrary Length (length = 10; 2 levels)



## WtDecompose

*Decomposes the input signal into wavelet series.*

```
void nspsWtDecompose(NSPWtState *wtState, float *src, float *dst);
            /* real values; single precision */
void nspdWtDecompose(NSPWtState *wtState, double *src, double *dst);
            /* real values; double precision */
void nspwWtDecompose(NSPWtState *wtState, short *src, short *dst);
            /* real values; short integer */
```

| | |
|---|---|
| **wtState* | The structure that stores all information needed for wavelet decomposition and reconstruction. |
| **src* | Pointer to the original signal. |
| **dst* | Pointer to the output decomposed signal. |

### Discussion

The `nsp?WtDecompose()` function decomposes the input signal specified by the **src* argument into wavelet series (see Figure 10-1). It uses the wavelet and filter parameters stored in the `NSPWtState` structure. To set these parameters, call the `nsp?WtInit()` and `nsp?WtSetState()` functions. To obtain the current values, use `nsp?WtGetState()`.

### Related Topics

| | |
|---|---|
| `WtInit` | Initializes the `NSPWtState` structure (see page 10-3). |
| `WtGetState` | Returns the current wavelet and filter parameters of the `NSPWtState` structure (see page 10-11). |
| `WtSetState` | Sets the wavelet and filter parameters (see page 10-11). |
| `WtReconstruct` | Reconstructs the signal from the wavelet decomposition (see page 10-16). |
| `WtFree` | Frees the memory used by the wavelet functions for internal purposes (see page 10-17). |

## WtReconstruct

*Reconstructs signals from the wavelet decomposition.*

```
void nspsWtReconstruct(NSPWtState *wtState, float *src, float *dst);
            /* real values; single precision */
void nspdWtReconstruct(NSPWtState *wtState, double *src, double *dst);
            /* real values; double precision */
void nspwWtReconstruct(NSPWtState *wtState, short *src, short *dst);
            /* real values; short integer */
```

| | |
|---|---|
| `*wtState` | The structure that stores all information needed for wavelet decomposition and reconstruction. |
| `*src` | Pointer to the wavelet decomposition data. |
| `*dst` | Pointer to the output reconstructed signal. |

### Discussion

The `nsp?WtReconstruct()` function reconstructs the signal from the wavelet decomposition given in the `*src` argument. It uses the wavelet and filter parameters stored in the `NSPWtState` structure. To set these parameters, use the `nsp?WtInit()` and `nsp?WtSetState()` functions. To obtain the current values, use the `nsp?WtGetState()` function.

### Related Topics

| | |
|---|---|
| `WtInit` | Initializes the `NSPWtState` structure (see page 10-3). |
| `WtGetState` | Returns the current wavelet and filter parameters of the `NSPWtState` structure (see page 10-11). |
| `WtSetState` | Sets the wavelet and filter parameters (see page 10-11). |
| `WtDecompose` | Decomposes the input signal into wavelet series (see page 10-15). |
| `WtFree` | Frees the memory used by the wavelet functions for internal purposes (see page 10-17). |

# WtFree

*Frees the memory used by wavelet functions for internal purposes.*

```
void nspWtFree(NSPWtState *wtState);
```

| | |
|---|---|
| `*wtState` | The structure that stores all information needed for wavelet decomposition and reconstruction. |

## Discussion

The `nspWtFree()` function frees the memory used by wavelet functions for internal purposes.

## Related Topics

| | |
|---|---|
| `WtInit` | Initializes the `NSPWtState` structure (see page 10-3). |
| `WtGetState` | Returns the current wavelet and filter parameters of the `NSPWtState` structure (see page 10-11). |
| `WtSetState` | Sets the wavelet and filter parameters (see page 10-11). |
| `WtDecompose` | Decomposes the input signal into wavelet series (see page 10-15). |
| `WtReconstruct` | Reconstructs the signal from the wavelet decomposition (see page 10-16). |

**Example 10-2 Decomposing a Signal of an Arbitrary Length**

```
#include <math.h>
#define   nsp_UsesWavelet
#define   nsp_UsesVector
#include "nsp.h"
/* MaxLevel : return maximum decomposition level            */
int MaxLevel(
    int len)        /* - length of Ethe y  signal               */
{
    int shift = 0;
    for(len--; len > 0; len >>= 1) shift++;
    return shift;
}
/* AbsThresh : limits the samples amplitude by given value  */
void AbsThresh(
    double *vec,    /* pointer to vector processed in place  */
    int     len,    /* length of vector                      */
    double  thresh) /* threshold value                       */
{
    int i;
    for(i = 0; i < len; i++)
        if(fabs(vec[i]) < thresh) vec[i] = 0;
}
/* NoiseReduction: perform noise reduction under signal      */
void NoiseReduction(
    double *src,        /* - pointer to noised signal        */
    double *dst,        /* - pointer to cleared destination  */
    double *pattern,    /* - noise pattern                   */
    int     len,        /* - signal length                   */
    int     lenPtr)     /* - length of noise pattern         */
{
    NSPWtState state;   /* - wavelet functions structure     */
    double    *dec;     /* - pointer to signal decomposition */
    double    *dec_lev; /* - pointer to current level of dec.*/
    double    *thresh;  /* - thresholds of noise gate        */
    int        len_dec; /* - length of signal decomposition  */
    int        level;   /* - decomposition level             */
```

continued

**Example 10-2 Decomposing a Signal of an Arbitrary Length** (continued)

```
    int        i;       /* - index (current level)          */
    /* maximum level of decomposition                        */
    level  = MaxLevel(MIN(lenPtr, len));
    thresh = nspdMalloc(level + 1);
    /* thresholds calculation by analysis of noise pattern   */
    nspdWtInitLen(0, 0, lenPtr, level, &state,
                    NSP_Vaidyanathan, &len_dec);
    dec = nspdMalloc(len_dec);
    nspdWtDecompose(&state, pattern, dec);
    nspdbAbs1(dec, len_dec);
    thresh[0]  =  nspdMax(dec, state.tree[level]);
    dec_lev    =  dec + state.len_dec;
    for(i = 1; i <= level; i++) {
        dec_lev  -= state.tree[i];
        thresh[i] = nspdMax(dec_lev, state.tree[i]);
    }
    nspFree(dec);
    nspWtFree(&state);
    /* noise reduction */
    nspdWtInitLen (0,0,len,level,&state,NSP_Vaidyanathan,&len_dec);
    dec = nspdMalloc(len_dec);
    nspdWtDecompose(&state, src, dec);
    /* ))decomposition discriminated by noise threshold((    */
    AbsThresh(dec, state.tree[level], thresh[0]);
    dec_lev    =  dec + state.len_dec;
    for(i = 1; i <= level; i++) {
        dec_lev  -= state.tree[i];
        AbsThresh(dec_lev, state.tree[i], thresh[i]);
    }
    nspdWtReconstruct(&state, dec, dst);
    nspFree(dec);
    nspWtFree(&state);
    nspFree(thresh);
}
```

*This page is left blank for double-sided printing*

*This page is left blank for double-sided printing*

# Library Information

# 11

Library function lists

This chapter describes a function that can be used to query the version number and the name of the current Signal Processing Library. The function returns a pointer to the data structure `NSPLibVersion` containing the required information.

## GetLibVersion

*Returns a pointer to the library version data structure.*

```
const NSPLibVersion *nspGetLibVersion (void);
```

### Discussion

This function returns a pointer to a static data structure `NSPLibVersion` that contains information about the current version of the Signal Processing Library. This structure is defined as follows:

```
typedef struct {
    const int major;
    const int minor;
    const int build;
    const char * Name;
    const char * Version;
    const char * InternalVersion;
```

```
    const char * BuildDate;
    const char * CallConv;
} NSPLibVersion;
```

where:

| | |
|---|---|
| *major* | is the major number of the current library version. |
| *minor* | is the minor number of the current library version. |
| *build* | is the build number of the current library version. |
| *Name* | is the name of the current library version. |
| *Version* | is the library version string. |
| *InternalVersion* | is the library version detail. |
| *BuildDate* | is the library version actual build date. |
| *CallConv* | is the library calling convention: DLL, Microsoft* or Borland*. |

For example, if the library version is 3.0, build 14, then the fields in this structure are set as

*major* = 3, *minor* = 0, *build* = 14.

Any of the fields in `NSPLibVerion` is returned as -1 if there is an error in retrieving the information.

Example 10-1 shows how to use the function `nspGetLibVersion()`.

### Example 11-1 Using the GetLibVersion Function

```
NSPLibVersion *p
p = nspGetLibVersion ()
printf (“Library Name: %s\n”, p->Name);
printf (“library Version: %s\n”, p->Version);
```

# *Fast Fourier Transforms* A

This appendix provides notes and hints on using fast Fourier transforms. For a more complete discussion, see Chapter 8 of [Mit93], and the references cited there.

The standard fast Fourier transform presented in most textbooks is a complex FFT; that is, its input is a complex vector and its output is a complex vector. However, typical signal processing applications need to take the FFT of real time-domain signals, not complex signals. While it is possible to promote the real signal to a complex signal by setting the imaginary part to zero and then apply a standard complex FFT, this approach is not efficient. A large family of real FFTs have been developed which operate directly on real inputs. The functions in the Signal Processing Library which operate on real inputs are `nsp?RealFftl()` (see page 7-23); `nsp?CcsFftl()` (see page 7-35); `nsp?RealFft()` (see page 7-38); `nsp?CcsFft()` (see page 7-45); `nsp?Real2Fft()` (see page 7-48); and `nsp?Ccs2Fft()` (see page 7-52).

The FFT of a real signal produces complex conjugate-symmetric (CCS) values, and the FFT of a CCS signal produces real values. Given this, there are four possible combinations of real input versus CCS input and time-domain versus frequency-domain:

1. real `x(n)` $\xrightarrow{\mathit{FFT}}$ CCS `X(k)`

2. real `X(k)` $\xrightarrow{\mathit{IFFT}}$ CCS `x(n)`

3. CCS `x(n)` $\xrightarrow{FFT}$ real `X(k)`

4. CCS `X(k)` $\xrightarrow{IFFT}$ real `x(n)`

where `x(n)` is a time-domain signal and `X(k)` is a frequency-domain signal.

Operations 1 and 4 are the most commonly used in typical signal processing. Operations 2 and 3 are much less common, but do appear in some filter design algorithms.

The functions `nsp?RealFft()` (and the other functions with `Real` in their names) implement operation 1 (with the *`flags`* value `NSP_Forw`) and operation 2 (with the *`flags`* value `NSP_Inv`), while the functions `nsp?CcsFft()` (and the other functions with `Ccs` in their names) implement operation 3 (with the *`flags`* value `NSP_Forw`) and operation 4 (with the *`flags`* value `NSP_Inv`).

One consequence of this arrangement is that `nsp?RealFft()` is not its own functional inverse. That is, composing operation 1 with operation 2 is not meaningful. Instead, operation 1 must be composed with operation 4 to get an identity operation. Similarly, `nsp?CcsFft()` is not its own functional inverse. While it would be natural to define a function that combines operations 1 and 4, this is not done by the Signal Processing Library because the type declaration problems that arise with such an arrangement cannot be adequately resolved.

For `Real` and `Ccs` functions, refer to Chapter 7.

# Digital Filtering

B

This appendix provides background about information on digital filtering and introduces the concepts of the following filters used by the Signal Processing Library:

- Finite impulse response (FIR) and infinite impulse response (IIR) filters
- Multi-rate filters
- Adaptive FIR filters using the least mean squares (LMS) algorithm

A digital filter is a system with frequency-selective capability. It can be used to modify, reshape, or manipulate a digital signal according to a specified requirement. Thus a specified range of frequencies can be attenuated, rejected or isolated from a signal. This capability ensures the use of digital filters in the following areas:

- Noise removal
- Compensation for signal distortion due to channel characteristics
- Separating signals
- Demodulation
- Digital to analog conversion
- Rate Conversion

## FIR and IIR Filters

The filters implemented in the Signal Processing Library belong to a class known as linear time-invariant (LTI) systems. A general LTI system is described by this differential equation:

$$y(t) = -\sum_{j=1}^{M-1} a_j \frac{d^j y(t)}{dt^j} + \sum_{i=0}^{N-1} b_i \frac{d^i x(t)}{dt^i},$$

where $x(t)$ is an input signal, $y(t)$ is an output signal, and $a_j$, $b_j$ are constants.

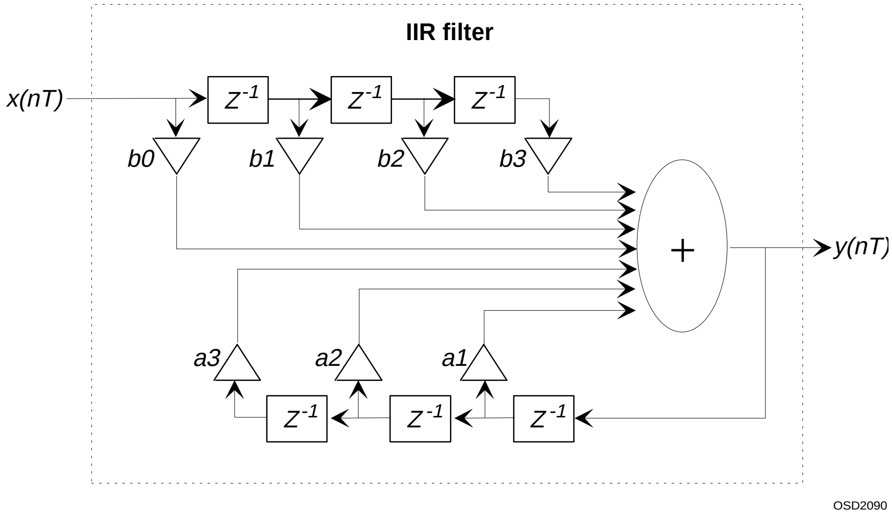
A corresponding digital LTI system is described by this equation:

$$y(nT) = -\sum_{j=1}^{M-1} a_j y((n-j)T) + \sum_{i=0}^{N-1} b_i x((n-i)T) \qquad (1),$$

where $T$ is the sampling step, $x(nT)$ is the sample of the input signal, $y(nT)$ is the sample of the output signal, and $a_j$, $b_j$ are constant coefficients (taps).

Equation (1) describes an IIR filter. In this equation, output values $y(nt)$ depend on input values $x((n-i)T)$ and $y((n-i)T)$, the latter being a feedback value of the previous time-step. In other words, the IIR filter uses a feedback loop. Such a filter is called an arbitrary order IIR filter. Figure B-1 provides an example of an IIR filter structure.

**Figure B-1 Example of an IIR Filter Structure**

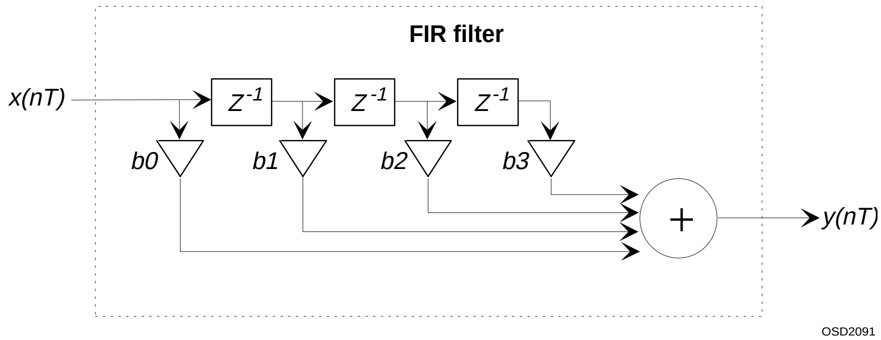



An FIR filter does not use a feedback loop. Assuming in equation (1) $a_j \equiv 0$, the equation for the FIR filter is

$$y(nT) = \sum_{i=0}^{N-1} b_i x((n-i)T) \qquad (2).$$

Figure B-2 illustrates an example of an FIR filter structure.

**Figure B-2 Example of an FIR Filter Structure**




To compute a new output count `y(nT)`, it is necessary to save a part of the previous time-step input `x((n-i)T)` (for an IIR filter, also output `y((n-j)T))`. The previous signal samples are saved in an array called a delay line. In equations (1) and (2), the length of the output signal delay line is `M-1`, and the length of the input signal delay line is `N-1`.

Each of the FIR and IIR filters have their advantages and disadvantages. FIR filters do not require feedback, they are more stable and used more often than IIR filters due to their stability. However, IIR filters provide higher performance because they do much less calculation than FIR filters.

When implementing various versions of equations (1) and (2) directly, the filter is known as a direct-form filter. When using mathematical transformations of equation (1), a different form of an IIR filter known as a biquadratic form (or a cascade form, or biquads) can be obtained. The implementation of cascade form filters consists of cascaded second-order sections. When properly implemented, the cascade form IIRs have better noise immunity than the direct-form filters.

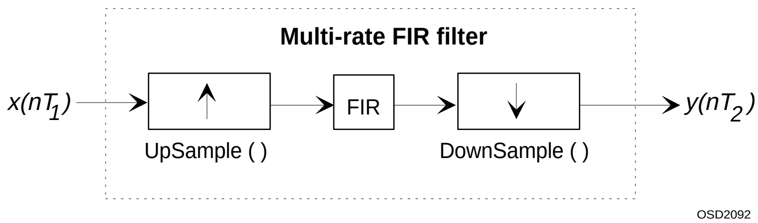
# Multi-Rate Filters

Most signal processing systems use signals of varying frequencies. In order to align or use signals with varying sampling rates, it is necessary to transform the sampling rate of signals to a common value. This transformation is known as sampling rate alteration. An example of such sampling rate alteration is that of an audio signal from 16 KHz to 8 KHz, that is, from frequency $f_1 = \frac{1}{T_1}$ to frequency $f_2 = \frac{1}{T_2}$.

The Signal Processing Library implements a special FIR filter in which the sampling rate alteration of the input signal as well as the output signal alteration is performed. Such filter is called a multi-rate filter. (For more details on multi-rate filters, see Appendix C).

A multi-rate filter can be thought of as consisting of three sequential elements: the up-sampler to increase sampling rate, the FIR filter, and the down-sampler to decrease sampling rate. In addition to the multi-rate FIR filter, the library provides the `nsp?UpSample()` and `nsp?DownSample()` functions to up-sample and down-sample the signals, respectively. Figure B-3 shows an example of a multi-rate FIR filter structure.

**Figure B-3 Example of a Multi-Rate FIR Filter Structure**

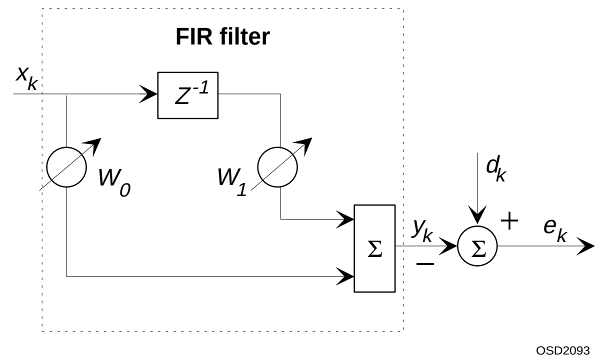



# Adaptive Filters

In some signal processing applications, the taps can be changed at each time-step to provide an optimal control effect. Filters with changeable taps are called adaptive filters.

Figure B-4 provides a scheme of a simple adaptive filter implementation. The scheme shows a delay line with two branches and a summing unit.

### Figure B-4 Simple Adaptive Filter Implementation




In Figure B-4, $x_k$ is the input signal, $y_k$ is the output signal, $W_0$, $W_1$ are filter taps, $Z^{-1}$ is the delay that delays input signal by one clock, $d_k$ is the desired signal, and $e_k$ is the error signal.

In a typical system control application, the filter tap values are updated at each iteration so as to reduce the mean square deviation of the output filter signal $y$ from the desired signal $d$ within a desirable error range $e$.

## LMS Filters

The least mean squares (LMS) filter is an adaptive filter which is based on the least mean squares algorithm to recalculate taps. The LMS filters (see the "Lmsl, bLmsl" section in Chapter 8) are computed using a gradient method. According to this method, each next tap vector is equal to the sum of the previous tap and a component proportional to the gradient value. The estimated gradient components are functions of the partial derivatives of the current vector. Although the algorithm includes the computation of mean square gradients of the error function, the actual implementation eliminates the squaring and differentiation operations which improves performance.

## High- and Low-Level Filters

The Signal Processing Library implements two filter levels: high and low. Functionally the high- and low-level filters are identical. However, the high- and low-level FIR filters differ in their implementation as shown in Table B-1. The low-level filter functions allow you to design your own adaptive filters with the control algorithms of your choice.

**Table B-1 Low- and High-Level Filters Implementation**

| Area of Difference | Low-Level Filters | High-Level Filters |
|---|---|---|
| Data structure | Implement two structures: taps and delay line arrays. | Implement a single structure that provides an access to both taps and delay line arrays. |
| Data owner | Application defines taps and delay line count arrays. | Application has no direct access to the taps and delay line count arrays which are stored in the dynamic memory. To access these arrays, the application uses `nsp?FirSet/GetTaps()` and `nsp?FirSet/GetDlyl()`. |
| Memory usage | Use taps and delay line count arrays of minimal length: *N* and *N*-1, respectively. No more memory is allocated. | Not restricted in memory use either for delay line or for taps or for any other purpose. This condition allows implemention of other computational means, for example, fast Fourier transforms. |

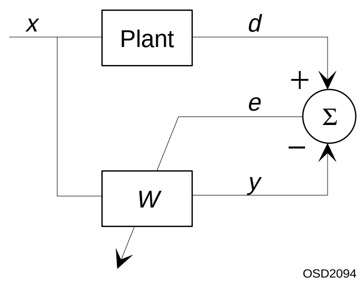
# Using Adaptive Filters

The following sections provide some basic examples of using the adaptive filters.

## System Identification

The adaptive filters can estimate system parameters with unknown transfer characteristics to identify a system. Figure B-5 shows a scheme of computing an adaptive filter in an identification mode. In this scheme, while reducing the error signal to zero, the filter is emulating the transfer characteristics of an unknown device.

**Figure B-5 Using an Adaptive Filter For System Identification**

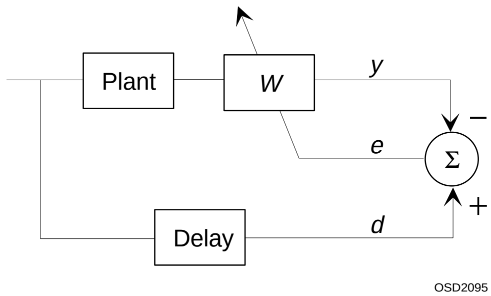



## Equalizing

An adaptive filter can be used for an inverse system parameter search. In this case, the filter serves as a deconvolver of input signal and a polynomial that describes filter parameters.

Figure B-6 shows a scheme of using an adaptive filter for equalizing. In this scheme, the filter restores the delayed input signal version, in particular, the one that has passed through a communication channel.

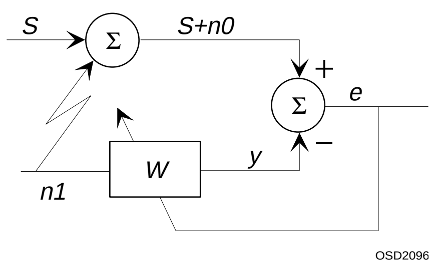
### Figure B-6 Using an Adaptive Filter for Equalizing




## Disturbance compensation

Figure B-7 presents a scheme of an adaptive disturbance compensator.

### Figure B-7 Using Adaptive Filter as Disturbance Compensator



*This page is left blank for double-sided printing*

*This page is left blank for double-sided printing*

# Multi-Rate Filtering

C

This appendix provides a brief overview of multi-rate filtering and the polyphase structure. It also includes a detailed discussion of the number of samples consumed and produced, and the required delay line lengths for finite impulse response (FIR) filter structure and the structure for the finite impulse response filter that uses the least mean squares adaptation (LMS).

Multi-rate filtering and the polyphase structure are not conceptually simple and cannot be adequately described here. Instead, see [Mit93], Chapter 14, *Multirate Signal Processing*.

## Defining Multi-Rate Filtering

Simple signal processing systems and algorithms process signals that are all at the same sampling rate. That is, the length of time between consecutive samples is the same for all signals. Such systems are called single-rate. More advanced systems often contain signals that have different sample rates. That is, the length of time between consecutive samples varies. Such systems are called multi-rate.

The sample rate for multi-rate filters can be increased or decreased by up-sampling or down-sampling. These processes are defined as follows:

- Up-sampling (see “UpSample” for a description of `nsp?UpSample()`) increases the sample rate by an integer factor by inserting zero samples between samples of the original signal. An up-sampling operation is often followed by filtering to “smooth” out the samples; this is called interpolation.

- Down-sampling (see "DownSample" for a description of `nsp?DownSample()`) decreases the sample rate by an integer factor by discarding samples. A down-sampling operation is often preceded by filtering to prevent aliasing; this is called decimation.

Multi-rate filtering, then, can be defined as interpolation and/or decimation; that is, filtering combined with up-sampling and/or down-sampling.

Multi-rate filtering (conceptually) follows this general scheme:

1. Up-sample a signal *`x(n)`* by the factor *`upFactor`* to produce a signal *`x'(n)`*.
2. Filter the signal *`x'(n)`* by the transform *`h(n)`* to produce the signal *`y'(n)`*.
3. Down-sample the signal *`y'(n)`* by the factor *`downFactor`* to produce the signal *`y(n)`*.

When both *`upFactor`* and *`downFactor`* are 1, the filter degenerates into a single-rate filter. Note that up-sampling and down-sampling each have an extra degree of freedom due to the *`upPhase`* and *`downPhase`* arguments.

## Polyphase FIR

In the multi-rate filtering operations described above, two sources of inefficiency may arise. The origin of these inefficiencies is described below.

Consider a direct implementation of the multi-rate filtering operations described above. In the implementation, let *`NP`* = *`upFactor`* * *`downFactor`*.

1. First *`downFactor`* samples of the signal *`x(n)`* are up-sampled to produce *`NP`* samples of a signal *`x'(n)`*.
2. These *`NP`* samples are filtered to produce *`NP`* samples of a signal *`y'(n)`*.
3. These samples are then down-sampled to produce *`upFactor`* samples of the signal *`y(n)`*.

Note that `downFactor` (not `upFactor`) samples are consumed to produce `upFactor` samples of `y(n)`, and that `NP` single-rate filtering operations are required. There are two sources of inefficiency in this direct implementation:

- Of the `NP` input `x'(n)` samples, only every `upFactor`'th sample is non-zero, the rest are zero.
- Of the `NP` output `y'(n)` samples, only every `downFactor`th sample will be kept and the rest discarded.

To optimize these two inefficiencies, the polyphase structure must perform the following tasks:

- The polyphase structure applies the filter `h(n)` only to those input samples which are non-zero.
- The polyphase structure applies the filter `h(n)` only to those output samples which are non-zero.

In order to perform these two optimizations, the polyphase structure makes `upFactor` passes, each pass computing one of the `upFactor` output samples of `y(n)`. Each pass is a dot product between a subset of the filter taps `h(n)` and the input `x(n)`.

The length of each dot product on each pass is called the phase length (`PL`). Then if the number of taps is denoted as `tapsLen`, the minimum phase length can be defined as $\lfloor$ `tapsLen`/`upFactor` $\rfloor$, and the maximum is $\lceil$ `tapsLen`/`upFactor` $\rceil$.

These values differ whenever `tapsLen` is not a multiple of `upFactor`. For the purposes of this discussion, define `PL` = $\lceil$ `tapsLen`/`upFactor` $\rceil$.

This is a valid bound for all up-sampling and down-sampling phases. The filter may be more efficiently implemented when `tapsLen` is a multiple of `upFactor`.

Also of interest is the delay line length. Since the zero samples introduced by up-sampling are never actually stored, the delay line length is closely related to the phase length. Intuitively, if `x(0)` is used in the dot products, then `PL` - 1 previous samples would be required. However, for the correct combination of phases, the first output value `y(0)` may not depend on any of the inputs `x(n)` at all; instead, it depends only on previous values stored in the delay line (for example, `upFactor` = 3, `upPhase` = 2, `downFactor` = 1). Thus the delay line must hold at least the maximum phase length of samples, `PL`. However, the implementation of a filtering operation might be directed to place the new `downFactor` samples from `x(k)` into the delay line before computing the dot products; this requires a delay line of length `PL` + `downFactor`.

Compare this multi-rate delay line length (`PL` + `downFactor`) to the single-rate delay line length of `tapsLen`. The single-rate case is not a trivial reduction of the multi-rate case; instead, it is smaller by one sample. This is because the multi-rate formula does not consider the phase parameters `upPhase` and `downPhase`. For example, when `upFactor` = 1, all outputs depend on the input sample(s), while this is not true for `upPhase` not equal to `upFactor` - 1.

## Polyphase LMS

A polyphase structure can also be used to more efficiently implement multi-rate LMS filters. The filter operation itself can be implemented exactly as described above, and the tap-update algorithm can take advantage of the known zero input samples when up-sampling. However, there are some practical difficulties. Even though the LMS filter is mathematically defined for up-sampling and down-sampling, the LMS filter functions in the Signal Processing Library do not support up-sampling for the following reason.

Up-sampling using a polyphase structure causes output samples to be produced in a block of `upFactor` samples, which introduces additional delay in the error feedback signal.

If your application requires up-sampling in combination with LMS filtering, you can avoid this problem by making an explicit call to `nsp?UpSample()`.

# *Bibliography*

This bibliography provides a list of reference books that might be useful to the application programmer. This list is neither complete nor exhaustive, but serves as a starting point. Of all the references listed, [Mit93] will be the most useful to those readers who already have a basic understanding of signal processing. This reference collects the work of 27 experts in the field and has both great breadth and depth.

The books [Opp75], [Opp89], [Jac89], and [Zie83] are undergraduate signal processing texts. [Opp89] is a much revised edition of the classic [Opp75]; [Jac89] is more concise than the others; and [Zie83] also covers continuous-time systems.


[Cap78] V. Cappellini, A. G. Constantinides, and P. Emilani. *Digital Filters and Their Applications*, Academic Press, London, 1978.

[CCITT] CCITT, Recommendation G.711

[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.

[Fei92] E. Feig and S. Winograd, Fast algorithms for DCT, *IEEE Transactions on Signal Processing*, vol.40, No.9, 1992.

[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.

[Jac89] Leland B. Jackson, *Digital Filters and Signal Processing*, Kluwer Academic Publishers, second edition,1989.

[Lyn89] Paul A. Lynn, *Introductory Digital Signal Processing with Computer Applications*, John Wiley & Sons, Inc., New York, 1993.

[Mit93] Sanjit K. Mitra and James F. Kaiser editors, *Handbook for Digital Signal Processing*, John Wiley & Sons, Inc., New York, 1993.

[NIC91] Nam Ik Cho and Sang Uk Lee, Fast algoritm and implementation of 2D DCT, *IEEE Transactions on Circuits and Systems*, vol. 31, No.3, 1991.

[Opp75] Alan V. Oppenheim and Ronald W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Opp89] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Rab78] L.R. Rabiner and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Rao90] K.R. Rao and P. Yip, *Discrete Cosine Transform. Algorithms, Advantages and Applications*, Academic Press, San Diego, 1990.

[Vai93] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, Englewood Cliffs, New Jersey.

[Wid85] B. Widrow and S.D. Stearns, *Adaptive Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Zie83] Rodger E. Ziemer, William H. Tranter, and D. Ronald Fannin, *Signals and Systems: Continuous and Discrete*, Macmillan Publishing Co., New York, 1983.

# Glossary

| | |
|---|---|
| adaptive filter | An adaptive filter varies its filter coefficients (taps) over time. Typically, the filter's coefficients are varied to make its output match a prototype "desired" signal as closely as possible. Non-adaptive filters do not vary their filter coefficients over time. |
| b | One of the flag values, which indicates the block variety of the function. The block variety of a function is equivalent to multiple invocations of the non-block (scalar) variety of the function. For example, the `nsp?Fir()` function filters a single sample through an FIR filter. The `nsp?bFir()` function filters a block of consecutive samples through a single-rate or multi-rate FIR filter. |
| Bq | One of the "mods," which indicates that the IIR initialization function initializes a cascade of biquads (second-order IIR sections). |
| causal filter | A filter whose response to input does not depend on values of future inputs. |
| 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 $a + b$j is $a - b$j. |
| conjugate-symmetric | See complex conjugate-symmetric. |
| DCplx | A C data structure which defines a double-precision complex data type. |
| decimation | Filtering a signal followed by down-sampling. The filtering prevents aliasing distortion in the subsequent down-sampling. See down-sampling. |
| down-sampling | Down-sampling conceptually decreases a signal's sampling rate by removing samples from between neighboring samples of a signal. See decimation. |
| element-wise | An element-wise operation performs the same operation on each element of a vector, or uses the elements of the same position in multiple vectors as inputs to the operation. For example, the element-wise addition of the vectors $\{x_0, x_1, x_2\}$ and $\{y_0, y_1, y_2\}$ is performed as follows: $\{x_0, x_1, x_2\} + \{y_0, y_1, y_2\} = \{x_0 + y_0, x_1 + y_1, x_2 + y_2\}$. |
| FIR | Abbreviation for finite impulse response filter. Finite impulse response filters do not vary their filter coefficients (taps) over time. For more information, see Chapter 8. |

| | |
|---|---|
| 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. |
| gradient method | A method that assumes that each next tap vector is equal to the sum of the previous tap and a component proportional to the gradient value. The estimated gradient components are functions of the partial derivatives of the current vector. |
| IIR | Abbreviation for infinite impulse response filters. For more information, see Chapter 8. |
| in-place | A function that performs its operation in-place, takes its input from an array and returns its output to the same array. See not-in-place. |
| interpolation | Up-sampling a signal followed by filtering. The filtering gives the inserted samples a value close to the samples of their neighboring samples in the original signal. See up-sampling. |
| leak | A parameter for the LMS filter functions which indicates how much the filter coefficients "leak" (decay) towards zero on each iteration of the function. |
| 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. For more information, see Chapter 8. |

| | |
|---|---|
| LTI | Abbreviation for linear time-invariant systems. In LTI systems, if an input consists of the sum of a number of signals, then the output is the sum of the system's responses to each signal considered separately [Lyn89]. |
| Mr | One of the "mods," indicating the multi-rate variety of the function. For more information on mods, see "Function Name Conventions" in Chapter 1. |
| multi-rate | An operation or signal processing system involving signals with multiple sample rates. Decimation and interpolation are examples of multi-rate operations. |
| Na | One of the "mods," indicating a non-adaptive filter function. For example, `nsp?LmslNa()`. See adaptive filter. For more information on mods, see "Function Name Conventions" in Chapter 1. |
| Nip | Not-in-place. One of the "mods," indicating a function which performs its operation not-in-place. That is, the function takes its input from a source array and puts its output in a second, destination array. For example, `nsp?FftNip()`. For more information on mods, see "Function Name Conventions" in Chapter 1. |
| not-in-place | A function that performs its operation not-in-place takes its input from a source array and puts its output in a second, destination array. |
| polyphase | A computationally efficient method for multi-rate filtering. For example, interpolation or decimation. |
| r | One of the flag values which indicates that the real and imaginary parts of an FFT function are stored in separate arrays. For example, `nsp?rFft()`. |
| RCCcs | A representation of a complex conjugate-symmetric sequence which is easier to use than the RCPack or RCPerm formats. See Table 7-6 in Chapter 7. |

| | |
|---|---|
| RCPack | 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). See “RCPack” in Chapter 7. |
| RCPerm | A format for storing the values for the FFT algorithm. RCPerm format stores the values in the order in which the FFT algorithm uses them. That is, the real and imaginary parts of a given sample need not be adjacent. See “RCPerm” in Chapter 7. |
| SCplx | A C data structure which defines a single-precision complex data type. |
| sinusoid | See tone. |
| step | A parameter for the LMS filter functions which indicates the convergence step size of the filter function. |
| 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

## C

## G

## M

## N

## O



## P



## R

## W