SigLib

Signal Processing Library

User's Guide

Version 10.60

11 January, 2026

Email : mailto:support@numerix-dsp.com
WWW : https://www.numerix-dsp.com/

Table of Contents

Introduction

SigLib is an ANSI C source signal processing library that is designed to be both portable and efficient. The library is available in two formats, the full source code package or the Binary (Formerly just the Dynamic Link Library (DLL)) variant. The full source code variant includes pre-built libraries for many of the wide variety of devices that are supported by SigLib.

The full source code package includes all of the C source code for the library and the demonstration programs. The code is supplied in C source format and the math / signal processing section of the library will compile using any ANSI C compiler.

The SigLib package is updated frequently and our web site (https://www.numerix-dsp.com/) is also updated frequently with information about the latest version of SigLib and also DSP applications, consequently you might like to bookmark the site.

The SigLib library has benefited from many suggestions from customers, who provide feedback, advice, suggestions and code. As a token of our appreciation, Numerix often provides extended maintenance to anyone who has helped us improve the library, so that they can benefit from the use of the library as it grows in the future.

Each different version of SigLib includes many new features and all new and existing customers are recommended to read the SigLib Overview section of this manual.

Navigating the SigLib installed directory structure is made easy through the welcome.html file located in the SigLib root directory.

Software Upgrades

New functionality is frequently added to SigLib. Customers who have a valid support contract will be informed of these updates. Should you wish to be informed of all available updates then please feel free to view our web site or contact us directly.

Documentation Overview

The SigLib documentation is split in to three sections, a User's Guide gives an overview of the SigLib library, whilst the Reference Manual gives a function by function description of the library and the Host Function Reference Manual. Users will probably find it beneficial to read the user's guide to get an understanding of how SigLib functions, they will then probably find that the reference manual is sufficient guidance in every day usage. The on-line nature of the documentation allows it to be used in parallel with the development tools.

Additional documentation is also supplied for the SigLib utility programs.

This manual is primarily written for the complete C source package however it also includes full details of the binary version.

Documentation Conventions

The SigLib documentation uses the following conventions :

The ANSI C standard conventions have been followed, for example hexadecimal numbers are prefixed by '0x'.

Names of directories, files and functions are given in italics.

Important programming information is indicated with the symbol : 

Software Installation

The easiest way to install SigLib is to clone it from:
https://github.com/Numerix-DSP/siglib

```
git clone https://github.com/Numerix-DSP/siglib.git
```

The installed directory structure is described below.

Before trying to use SigLib, please ensure that the *include and lib* files are in the appropriate library and include file search paths for your C compiler.

Source Package Installed Directory Structure

The following table shows an abridged extract of the installed directory structure, to highlight the key folders.

rootpath\SigLib
- AppNotes - Applications notes related to the use of SigLib
- Benchfft - SigLib FFT benchmark examples for many different processors
- ChBuild - Build files for Ch
- Docs - Documentation files
- Examples
  - C#Examples - C# SWIG examples
  - CEExamples - SigLib Windows Mobile 2003 examples
  - CExamples - Examples written to support the Gnuplot/C API (https://www.numerix-dsp.com/files).
    - Beamforming
    - CostasQAM
    - iir_coeff_generate
    - ImageExamples
    - MachineLearning
    - MachineLearning_PortAudio (works on Raspberry Pi)
    - OrderAnalysis
    - ToneLevels
    - SynchronousSampleRateConversion
    - VibrationAnalysis
  - ChExamples - Ch examples
  - DLLExamples - SigLib DLL examples for C++, Vee, Excel and Visual Basic
    - BenchIIR
    - CPPTest
    - Delphi
    - DTMFWav
    - Histogram
    - Matrix
    - testifs2
    - VBTest
    - Vee
    - Xltest
    - ZoomFFT
  - JavaExamples - Java SWIG examples
  - PerlExamples - Perl SWIG examples
  - PortAudioExamples - Real-time examples that run under Windows and Linux
- gnuplot_c - Gnuplot/C graphing library
- include - SigLib header files
- lib - Library files
  - 21k - ADSP-21xxx compiled libraries
  - ARMV4WinCE - ARM Windows Mobile 2003 libraries

	C6x	- TI TMS320C6000 libraries
	Ch	- Ch
	IAR	- IAR Embedded Workstation libraries for ARM
	linux	- Linux libraries
	Microsoft	- Microsoft Visual C/C++ satic and DLL libraries
	Raspberry Pi	- Raspberry Pi libraries
	sc100	- StarCore SC100 libraries
	XMOS	- XMOS libraries
	ZSP	- LSILogic ZSP libraries
ngl		- Numerix graphics library
AudioFiles		- Sampled audio and speech files
src		- SigLib source files
SWIG		- Files for building the SWIG interface
utils		- SigLib utility files
WebResources		- Resource files used in the html documentation

SigLib Quick Start Guide

Quick start batch files are included for Windows, Linux and the Raspberry Pi.
Use the following instructions to configure the paths and build the library, you might like to add the SigLib configuration to your system environment variables (e.g. .bashrc in Linux) the SetEnv batch file or shell scripts are provided to show you how to do this.

Microsoft Windows

Ensure you have a configured Visual Studio command prompt and type:

```
configure.bat
```

Linux/GCC

Type:

```
./configure.sh
```

Raspberry Pi Linux/GCC

Type:

```
./configureRPi.sh
```

Mac OSX/GCC

Instructions are included below to build the library for OSX.

SigLib Coding Standards (ISO/IEC 9899:1999)

SigLib is coded according to the (ISO/IEC 9899:1999) standard, also commonly referred to as C99. This is supported by almost all modern C/C++ compilers however some do not support C99 by default and require a compiler option (typically "`-c99`" or "`-std=c99`", for gcc) to switch the compiler into C99 compatible mode.

If you find that compiling the library generates compiler warnings then please ensure that you have set the compiler options correctly.

Installing And Using SigLib Under Microsoft Windows

Install the software into your chosen directory (e.g. `c:\siglib`)

Microsoft

Make the following additions to the environment variable list :

```
set PATH=%PATH%;c:\siglib\utils
set INCLUDE=%INCLUDE%;c:\siglib\include;c:\siglib\ngl;C:
\siglib\gnuplot_c;c:\portaudio_v19\include
set LIB=%LIB%;C:\siglib\lib\Microsoft\static_library\Release;c:
\siglib\ngl;C:\siglib\gnuplot_c;C:
\portaudio_v19\build\msvc\Win32\Release
```

The easiest way to use SigLib under Microsoft Windows is to use the free Visual Studio Community Edition and the workspace file "siglib.sln". This workspace allows the library to be rebuilt as a static or dynamic linked library, in either debug or release mode. An example program is provided in C:\siglib\DLLExamples\CPPTest and the program can be built using the workspace file "CPPTest.dsw".

A batch files (SetEnv.bat) is included in the SigLib root directory for configuring the environment variables.

A batch file (remakeMSVC.bat) is provided in C:\siglib\src that will rebuild the library in the following models using Visual Studio Community Edition :

```
Static library, Debug model
Static library, Release model
Dynamic library, Debug model
Dynamic library, Release model
```

If you would like to link a SigLib program with a static library then you can define the following in the project options :
-DSIGLIB_STATIC_LIB=1
This will tell the compiler that it is linking your application against the static library version of SigLib. The static libraries are located in C:\siglib\lib\Microsoft\static_library_64\Debug or C:\siglib\lib\Microsoft\static_library_64\Release, depending on which version you would like to use.

To use the graphics functionality within the examples, with the Microsoft compilers, you will need to install Gnuplot (http://gnuplot.info/). Further details are available in the Gnuplot/C User's Guide (C:\siglib\gnuplot_c\Gnuplot_C Users Guide.pdf).

Installing And Using SigLib Static Library Under Linux

Install the software into your chosen directory (e.g. ~/siglib)
SigLib is provided as a pre-built static library (libsiglib.a) which is located in ~/siglib/lib/linux. If you would like to rebuild the SigLib static library then use the following commands :

```
$ cd ~/siglib/src
$ gmake -f makefile.lx clean
$ gmake -f makefile.lx
```

Note : for the Raspberry Pi the makefile is called makefile.lx.rpi. This will save the library in the folder siglib/lib.RaspberryPi.

To build one of the examples you can use gb.sh script, as follows :

```
$ cd ~/siglib/Examples/CExamples
$ ./gb.sh filename
```

Where filename is the name of the source file.

There are also several examples provided with makefiles, for example the matrix.c example and is used as follows :

```
$ gmake -f makefile_matrix clean
$ gmake -f makefile_matrix
```

To use the graphics functionality within the examples, under Linux, you will need to install Gnuplot (http://gnuplot.info/). Further details are available in the Gnuplot/C User's Guide (`C:\siglib\gnuplot_c\Gnuplot_C Users Guide.pdf`).

Several of the SigLib examples use the traditional MS-DOC compiler conio.h functionality which is not supported by Gcc. SigLib includes a collection of these functions (from various Internet sources) in the include file `gconio..h` - please refer to this file for source references.

Rebuilding SigLib Shared Object Library Under Linux

Install the software into your chosen directory (e.g. `~/siglib`)
SigLib is provided as a pre-built shared object library (`siglib.so.1.0`) which is located in `~/siglib/lib/linux`. If you would like to rebuild the SigLib shared object library then use the following commands :

```
$ cd ~/siglib/src
$ gmake -f makefile.lxso clean
$ gmake -f makefile.lxso
```

Note : for the Raspberry Pi the makefile is called `makefile.lxso.rpi`. This will save the library in the folder `siglib/lib.RaspberryPi`.

To build one of the examples we will use the `matrix.c` example :

To use the shared object file you will need to use the following procedure :

```
$ cd ~/siglib/lib/linux
$ ldconfig -v -n .
$ ln -sf siglib.so siglib.so.1
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

Installing And Using SigLib Static Library Under MAC OS

Install the software into your chosen directory (e.g. `~/siglib`)
SigLib is provided as a pre-built static library (`libsiglib.a`) which is located in `~/siglib/lib/macos`. If you would like to rebuild the SigLib static library then use the following commands :

```
$ cd ~/siglib/src
$ make -f makefile.macos clean
$ make -f makefile.macos
```

To build one of the examples we will use the `test_mat.c` example :

```
$ cd ~/siglib/Examples
$ gcc -I ../include -I ../ngl -I ../gnuplot_c test_mat.c
-L../lib/linux -l siglib -o test_mat -L../ngl -L../gnuplot_c -lngl -
lgnuplot_c -lm
```

A `makefile` is also provided to rebuild the `matrix.c` example and is used as follows :

```
$ make -f makefile_matrix clean
$ make -f makefile_matrix
```

To use the graphics functionality within the examples, under MACOS, you will need to install Gnuplot (http://gnuplot.info/). Further details are available in the Gnuplot/C User's Guide (`C:\siglib\gnuplot_c\Gnuplot_C Users Guide.pdf`).

Cmake

A CMakeLists.txt file is included in the root folder, to build SigLib with the cmake utility and is designed to work with any compiler toolset that supports cmake.
This is a minimalistic cmake project which builds SigLib library itself only and has been tested under Windows and Linux.
It does not build tests, utilities and example programs.

Here are steps to build SigLib:
1) cd to siglib root folder

2) cmake -S ./ -B ./build
This creates a ./build folder and puts all the build scripts in it.

3) cmake --build ./build
This runs the build tools against the build scripts prepared in the configuration step. The built library (siglib.lib for Windows and libsiglib.a for Linux) in ./build/Debug subfolder.

SigLib Overview

The SigLib library provides a complete resource for the DSP developer. The SigLib library functions are split into several sections, as follows :

Frequency domain
Windowing
Fixed coefficient filter (IIR and FIR)
Acoustics and beamforming
Adaptive coefficient filter (LMS FIR)
Convolution and Correlation
Image processing and coding
Signal generation
Modulation, Demodulation and Communications
DSP utility
Digital effects
Control systems
Speech processing
And many more.

In general, all the information required by a function is passed as parameters (including data array pointers, array sizes etc.) either on the stack or via registers, depending on the compilation options. In order to maximise the program efficiency, the majority of the data arrays are all linear and real however some functions do support strided data.

Individual complex data variables are stored in a structure containing the real and imaginary components. Complex data arrays are handled in one of two ways :

1/ Two separate arrays for separate real and imaginary components
2/ One single array for interleaved real and imaginary components

The reason for these two different approaches is that the first often gives higher performance but for some functions, the second is more natural.

Some of the functions work on buffers of data (these functions begin `SDA_` . .) and others on single samples (these functions begin `SDS_` . .). The block handling functions, like the buffer oriented filter variants, have been written to so that they can receive buffered streams of data and they ensure that there are no discontinuities in the output data streams.

Where possible the SigLib functions have been tested with large buffers and no limits have been found. We realise that many users will push the limits of the software and any feedback on this subject would be very welcome.

Some of the functions require specific initialisation operations and these have been placed into separate initialisation functions so that the run-time performance of the primary routines is not compromised.

All of the functions in SigLib are re-entrant and multi-thread safe. As a result, all status information and any information that is required to be shared with other functions is managed by the calling function. Managing the data in this format allows the library to process as many different incidences of a function as required by the application. Each of the incidences may have completely

independent configurations. For arrays that are passed to functions through pointers, it is important that the arrays are large enough for the amount of data that is written into them.

The SigLib source code has been written using an editor with a tab spacing of 4 spaces.

The SigLib Applications Programming Interface

The SigLib API uses a modular naming convention. All functions that process arrays are prefixed with the letters 'SDA_', where as all sample centric functions are prefixed with the letters 'SDS_'. This allows the array version of the function and the sample oriented version to use the same naming convention and the two are differentiated by the prefix. The initialisation operation for all functions use the SIF_ prefix. An example of how the new convention operates is the FIR filter, as follows :

SIF_Fir	Initialise the FIR filter functionality
SDS_Fir	Perform the FIR filter on the sample oriented data stream
SDA_Fir	Perform the FIR filter on the array oriented data stream

The complete set of function prefixes are :

SDS_	Data sample oriented operations
SDA_	Data array oriented operations
SCV_	Complex vector sample oriented operations
SCA_	Complex vector array oriented operations
SMX_	Multi-dimensional matrix array oriented operations
SIM_	Image processing and coding operations
SUF_	Utility functions
SIF_	Initialisation function – Initialise look up tables and state arrays
SRF_	Reset function – Only reset critical data e.g. state arrays
SAI_	Functions that return data of type SLArrayIndex_t. Note : SigLib can be configured (in the file *siglib_processors.h*) to support different word lengths for fixed point data and array indices.

Note : Not all functions require initialization and/or reset functions.

All SigLib constants start with the string SIGLIB_.

 Many SigLib functions can operate in different modes, depending on the parameter selection. Some modes require different configurations of input and output arrays. For example, some may require pointers to a data array or look up table in some modes of operation but not in others. For modes that do not require particular pointers to be used, SigLib includes the following NULL pointer macros :

SIGLIB_NULL_FLOAT_PTR for data of type SLData_t
SIGLIB_NULL_FIX_PTR for data of type SLFixData_t

For further information on SigLib data types, please see the SigLib Data Types section of this manual.

SigLib Data Types

SigLib defines the following standard user data types :

SLData_t	SigLib data values – generally floating point data
SLChar_t	Character based fixed point values
SLFixData_t	Fixed point data values
SLUFixData_t	Unsigned fixed point data values
SLArrayIndex_t	Array index / offset / length values - must be a signed variable
SLImageData_t	Image data values
SLBool_t	Boolean values - not used at present
SLError_t	Error code values
SLStatus_t	Status code values - not used at present

To ease portability across different processors and systems, SigLib defines the following base data types :

SLInt8_t	Signed 8 bit integer values
SLUint8_t	Unsigned 8 bit integer values
SLInt16_t	Signed 16 bit integer values
SLUInt16_t	Unsigned 16 bit integer values
SLInt32_t	Signed 32 bit integer values
SLUInt32_t	Unsigned 32 bit integer values
SLInt64_t	Signed 64 bit integer values - not supported by this compiler
SLUInt64_t	Unsigned 64 bit integer values - not supported by this compiler
SLFloat32_t	32 bit floating point values
SLFloat64_t	64 bit floating point values

These base data types are defined for each compiler / processor combination in the file *siglib_processors.h*. These types should not generally be used within the application code base because the standard user data types should be used.

These user data types are also defined in the file *siglib_processors.h*. Some functions use specific word length types for example the scramblers use registers up to 32 bits long so always use the SLUInt32_t data type. Details of the specific data types required by each function are included in the SigLib function reference manual (*ref.pdf*).

The default SigLib data types are shown in the following table :

Device	Compiler	Vector Data Type	Integer Data Type	Array Index Type	Image Data Type
80x86	Microsoft / GNU	double	long	long	Unsigned char
TMS320C6000	TI	float	long	long	Unsigned char
ADSP2106X	Analog Devices	float	long	long	Unsigned char
ADSP2116X	Analog Devices	float	long	long	Unsigned char
SPARC	Sun / GNU	double	long	long	Unsigned char
PowerPC	GNU / DiabData / Wind River	double	long	long	Unsigned char
StarCore	StarCore	float	long	long	Unsigned char
ZSP	LSILogic	Float	long	long	Unsigned char

Extra devices and compilers can be supported by adding additional definition statements to the top of the file *siglib_processors.h*. SigLib users are advised to use the `SLData_t` and `SLFixData_t` data types at all times, to assist themselves with portability and future upgrades.

Many of the user data types can be modified by defining constant values on the command line, as follows :

To declare the integer data type (`SLFixData_t`) as short (16 bits) then define the value `SIGLIB_FIX_SHORT` on the compiler command line as a non zero value. The default data type is long (32 bits).

To declare the SigLib data variables (SLData_t) as a short (16 bits) fixed point then define the value `SIGLIB_DATA_SHORT` on the compiler command line as a non zero value. To declare it as 32 bit floating point then define the value `SIGLIB_DATA_FLOAT` on the compiler command line as a non zero value.

If the application only uses arrays that are shorter than 32767 samples long then the array index typedef (`SLArrayIndex_t`) can be set to short (16 bits) by defining the value `SIGLIB_INDEX_SHORT` on the compiler command line as a non zero value. The default array index type is long (32 bits).

For further information on data types for the TMS320C62x devices, please refer to the section of this manual titled Compiling And Using SigLib Functions.

SigLib fixed point data type `SLArrayIndex_t` is used for array indexing and other memory access processes. When using small memory models in some segmented memory architectures these pointers are truncated to 16 bits and

compiler warnings are often given. The solution is to use a `#pragma` to disable the warnings.

Before using SigLib please read the "Compiling And Using SigLib Functions" and "Optimising The Performance Of SigLib Functions" sections of the Users Guide.

The file *siglib.h* should be included in all source files that access the SigLib functionality. It is important to ensure that *siglib.h* is the last header file included because it will use information supplied by the compiler's standard header files.

When building the SigLib library it is important that the compiler options for memory models etc. are the same as those used for the application program AND the standard run-time support libraries.

SigLib normalises all frequencies to a sample rate of 1 Hz, therefore particular frequencies can be calculated as follows :

$$\text{frequency parameter} = \frac{\text{required frequency}}{\text{sample rate}}$$

SigLib Data Files

SigLib allows for a wide range of files to be accessed and examples are provided that support many of these functions. The primary file used by SigLib is the .sig format, which is an ascii file containing samples stored in floating-point format, one sample per line. Examples are : *aa.sig*, *ee.sig*, *er.sig*, *i.sig* and *oo.sig* are all sampled at 10 KHz, 16 bits. The image files are 256 x 256 images with 8 bit pixels.

SigLib .sig files can easily viewed or modified with a regular text editor or imported into programs like Excel and Matlab.

File formats supported are :

File Extension	Description
.bin	Contiguous 16 bit binary data
.csv	Comma Separated Variable format for importing into a spreadsheet
.dat	Two column format, with header. Column 1 : sample timestamp Column 2 : data sample This format is used by gnuplot
.sig	A single column of floating point numbers that represent the data sequence
.wav	16 bit multi-channel .wav file

Functions are included in SigLib to translate these file formats and can be found in the Examples/ FileIO folder. If you wish to compile these applications and use them in scripts for converting file formats then a system environment variable path to this folder can be created.

SigLib Example Programs

The SigLib library is supplied with many example source files, which are located in the `/siglib/Examples` folder and listed in the file *index.html*.

The most complete collection of examples is located in : `/siglib/Examples/CExamples`.

All of the standard examples in C:\siglib\Examples\CExamples are can be compiled with the free Microsoft Visual Studio Community Edition compiler and GCC.

Real time I/O and DSP processing can now be performed using PortAudio (http://www.portaudio.com/) and examples to show this are provided in the PortAudioExamples directory.

If you wish to use the graphical examples on a DSP or embedded processor then the only modification that is required is to remove the graphics functions from the example programs. Some modified examples are included in the DSPExamples directory. Most modern DSP development systems support the stdio standard console and file I/O functions via JTAG debug facilities.

SigLib examples are also included for the DLL library, these are located in the "DLLExamples" directory. Examples are supplied for Visual C++ and some also compile with the Express Edition, Visual Basic (includes an example application and complete example globals.bas file for all the SigLib functions), Excel and HP VEE.

File I/O examples are located in : `/siglib/Examples/FileIO`.

Optimising The Performance Of SigLib Functions

For the purposes of performance and re-entrancy, all functions that require internal arrays must have these allocated by the calling function prior to being used. If it is necessary to initialise these arrays to specific values prior to calling the function then initialisation functions are provided or the documentation will indicate that these arrays must be cleared to zero (for example using the SDA_Zeros() function).

When compiling the SigLib source code to run on an embedded system or DSP, many application specific optimisations can be made to the code and the following guide lines may be found to be useful :

- Declare buffers and variables globally, where possible, to avoid double indirection.
- If the program does not initially run, disable all code optimisation.
- Take care when choosing to use pointers vs. offsets in to arrays. The best choice is not always obvious and will depend on the compiler, processor architecture and specific function being used.
- Re-code SigLib functions “in-line”. To remain inside your license and warranty conditions, when doing this, you must also copy the SigLib file header.
- Use on-chip memory (if available).
- Turn on the cache (if available).
- Pointers to source buffers can be declared as “const *” pointers - this can assist the compiler with CPU data flow optimisation.
- Local variables can be declared “static”, to locate them on the heap.

When compiling under Microsoft C, problems have been found with maximum optimisation enabled, all the functions have been compiled and tested with the following options : -Otin -Gs -G*N* -W4, where *N* represents the processor family. Users will find that compiling with higher levels of optimisation will often run faster, however the operation can not be guaranteed.

Further information relating to this subject can also be found on our web site at : https://www.numerix-dsp.com/c_coding.html.

SigLib Library Processing Modules Overview

SigLib is split into separate modules according to the source files listed in the following table. Each module is also discussed in further detail below.

Module	Functionality
acoustic.c	Acoustic processing and beamforming
adaptive.c	Adaptive filtering
arbfft.c	Arbitrary length FFT
chirpz.c	Chirp z-transform
comms.c	General telecommunications
complex.c	Complex mathematics on individual complex numbers
complexa.c	Complex numbers processed as complex arrays
control.c	Control
convolve.c	Convolution
correlate.c	Correlation
datatype.c	Convert between different data types – fixed, float etc.
decint.c	Decimation and interpolation
delay.c	Delay functions
dsputils.c	General utility functions for processing data
dsputil2.c	More general utility functions
dsputil3.c	More general utility functions
dtmf.c	DTMF generation and detection
fdfilter.c	Frequency domain filtering
ffourier.c	Fast Fourier transforms
filter.c	General filters
firfilt.c	FIR filtering functions
fourier.c	Discrete Fourier transforms
icoder.c	Image coding
iirfilt.c	IIR filters
image.c	Image processing
machinelearning.c	Machine Learning
matrix.c	Matrix manipulation
minmax.c	Minimum and maximum detection and setting
mod_a.c	Analog modulation
mod_d.c	Digital modulation
modem.c	Digital modulation
mux.c	Multiplexing / demultiplexing
prbs.c	Pseudo random binary sequences
pspect.c	Power spectrum analysis
regress.c	Regression analysis
siggen.c	Signal analysis
siglib.c	Generic SigLib functions
smath.c	General mathematics
speech.c	Speech processing
stats.c	Statistics
timing.c	Timing detection and extraction
trig.c	Trigonometry
viterbi.c	Trellis encoding and Viterbi decoding
window.c	Windowing

Fourier Transform Functions

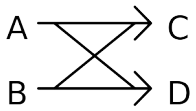
SigLib provides functions for Fast Fourier Transforms (FFTs) for both radix-2 and radix-4 FFTs, within the file (*ffourier.c*), with supporting functions located in the file (*fourier.c*) . Arbitrary length FFT functions are located in the file *arbfft.c* and frequency domain filtering functions are located in *fdfilter.c*.

Radix-2 FFT Functions

There are three basic Fourier transform functions, they are SDA_Rfft (real FFT), SDA_Cfft and SDA_Cifft (complex forward and inverse FFTs). Each of the FFT functions perform in-place transforms, i.e. the result data is placed back in the source data arrays. The FFT functions use transform lengths that must be a power of 2 and they all use the Radix-2 butterfly. Tests have shown that the real transforms are almost twice as fast as the complex ones, the actual percentage difference however depends on the compiler used. The FFT functions do not scale their output, different applications may require different scaling, therefore the function SDA_Multiply should be used to ensure that the correct scaling is maintained for the application.

Prior to using any of the FFT functions, the function SIF_Fft must be called, this, amongst other things initialises the twiddle factor tables. In order to be able to support FFT different FFT lengths simultaneously it is necessary to initialise each length required with a separate call to SIF_Fft function, with the coefficients and, if required, bit reverse address tables being located in separate arrays.

Different text books use different notations for the sign of the sine term, when performing forward and inverse FFTs, SigLib uses the following notation :



- Radix 2 butterfly

Cr = Ar + Br
Ci = Ai + Bi
Dr = (Ar-Br) * Cos (Theta) + (Ai-Bi) * Sin (Theta)
Di = (Ai-Bi) * Cos (Theta) - (Ar-Br) * Sin (Theta)

It is recommended that users verify before hand that this is the notation, required, for their application. The phase differences, between the different notations is irrelevant, when performing a square magnitude sum on the results.

For an example of how to use the FFT functions, please see the example *test_fft.c*. A function SDA_ZoomFft is supplied, to allow frequency zooming, by modulating and decimating the signal, this function is very useful, in applications, where a high frequency resolution is required, over part of the bandwidth and where performing a large FFT would be computationally too expensive.

The arbitrary length FFT functions use Bluestein's algorithm to calculate the FFT via the chirp z-transform for arbitrary length data sets.

FFT bit reverse re-ordering happens in one of 3 ways, that are configured by passing the following to the function parameter pBitReverseAddressTable:

pBitReverseAddressTable	Bit Reverse Addressing Mode
SIGLIB_BIT_REV_NONE	No bit reverse addressing is performed
SIGLIB_BIT_REV_STANDARD	Bit reverse addressing is handled via computation of the addresses to swap
A valid memory address	Bit reverse addressing is handled via a look-up table so is faster but requires the use of the additional look-up table, in memory. Note: for this mode to work the look-up table must not be located at memory addresses 0x0….00 or 0x0….01

Window Weighting Functions

The window weighting functions generate an array of *n* coefficients, to weight a time-series data array to improve the spectral response characteristics of the FFT.

After generating a window array with one of the window functions, the SDA_Window() or SDA_ComplexWindow() functions will apply that window to the time domain data array(s), prior to performing the FFT. The window application functions requires a pointer to source and destination data arrays, however if an in-place windowing is required then the destination can be the same as the source.

The window functions available are : Hanning, Hamming, Generalized Cosine, Blackman, Bartlett / Triangle, Kaiser, 4th order Blackman-Harris, Rectangle and flat top.

Each window attenuates a signal, to which it is applied, this attenuation can be reversed by using the function SDA_WindowInverseCoherentGain(), which returns the gain that must be applied to the signal after the window has been applied.

Filtering Functions

SigLib provides functions for finite impulse response (*firfilt.c*), infinite impulse response (*iirfilt.c*) and generic (*filter.c*) fixed coefficient filters and adaptive filter functions (*adaptive.c*).

Versions of the FIR and IIR filter routines are supplied to be applied to both complete data arrays (SDA_...) and to individual sample streams (SDS_ ...).

SigLib also includes a range of filter design functions, including IIR and FIR filter design, including windowing and bilinear transform.

A standalone digital filter design program is available at : http://numerix-dsp.com/dfplus/.

FIR Filter Overview

The coefficients for the FIR filter are in the form of a linear array (h_0, H_1, … $h_{(N-1)}$)

Three basic versions of the FIR filter are provided, for both real and complex implementations. These different versions are provided to allow efficient computation on a range of different compiler/processor combinations.

_FIR	FIR with integrated circular buffering. Some compilers can optimize the circular buffering and use hardware based zero overhead modulo addressing, if the DSP supports this.
_FIRWithStore	Bucket brigade FIR shifting the data down the state array after computation. Where modulo addressing is not supported in hardware this can be more efficient than the _FIR mode.
_FirExtendedArray	Uses double length state and coefficient arrays to ensure that neither circular buffering nor state variable storage are required during the dot product calculation. This can be the most efficient method on traditional microprocessors or microcontrollers. Note : Additional functions are provided for allocating the extended arrays.

IIR Filter Overview

The IIR filter coefficients are stored in a linear array, as follows :

stage 1 A0, A1, A2, B1, B2
stage 2 A0, A1, A2, B1, B2
.
.
stage N A0, A1, A2, B1, B2

The IIR filter form is Direct Form II and has been chosen for the best compromise between processing efficiency and stability.

The defined constant IIR_COEFFS_PER_BIQUAD defines the size of the memory space to store the coefficients for each biquad section. This can be used to allocate the necessary memory space.

Two basic versions of the IIR filter are provided :

_Iir	The feedback coefficients are negated in the IIR function.
_IirMac	The feedback coefficients are NOT negated in the IIR function. This requires the coefficients to be negated at design time. This allows for more efficient run-time implementation on processor architectures that only support a single cycle Multiply-Accumulate function and not a single cycle Multiply-Subtract function.

Efficient forms of the IIR filter construct are available via the SDA_OnePole() and SDA_OnePolePerSample() filter functions.

Other Filter Functions

Several forms of comb filter are supported in SigLib, each gives a different response, and each are applicable for different applications.

Various delay functions are also supplied, to allow for delays of samples and delay of streams (across buffers).

The function SIF_HilbertTransformer() initialises the coefficients of an FIR Hilbert transformer filter.

The Goertzel algorithm is supported with two functions; SDA_GoertzelFilter() just filters the data stream while SDA_GoertzelDetect() returns the power of the signal in the pass band of the filter.

Image Processing Functions

The image processing section of the SigLib library contains many two dimensional FFT and convolution functions. Unlike the data in the remainder of the library, the image processing functions accept fixed point data, declared as unsigned char, to save memory however the internal operations are all floating-point.

The conv3x3 function convolves an arbitrary n x m image with a 3x3 kernel. In addition to these functions, the *dsputils.c,dsputil2.c* and *dsputil3.c* files also contains many routines, which are useful for image processing, for example the histogram, clipping and scaling functions.

The supplied example program uses the coefficients :

0.0	0.1	0.0
0.1	0.5	0.1
0.0	0.1	0.0

This is a low pass filter, which may be passed over the image several times to clean it up. After cleaning up the supplied image, the coefficients :

0.0	0.1	0.0
0.1	-0.4	0.1
0.0	0.1	0.0

can be applied, this is an edge detecting kernel. Another useful filter is the Gaussian low pass filter :

0.1	0.2	0.1
0.2	0.4	0.2
0.1	0.2	0.1

For an example of how to use the image processing functions, please see the example *test_img.c*, which must be linked with the large memory model SigLib library.

The two dimensional FFT function is SIM_Fft2d and any calls to this function must be proceeded by a call to : SIF_Fft2d.

The sobel3x3 function performs a two dimensional Sobel edge detection filter on the image. This filter gives better performance if the image has been cleaned up by low pass filtering and thresholding. A 3x3 median filter is also included in the library.

Modulation, Demodulation And Communication Functions

The SigLib DSP library includes functions for both analog and digital modulation and data stream (de)multiplexing functions such as : `SDA_Mux_N`, `SDA_Demux_N`. These functions can receive buffered streams of data, to ensure that there are no discontinuities in the input or output data streams. The amplitude modulator functions works equally as a modulator and a de-modulator. There are also several spectrum manipulation algorithms available in the file *dsputil2.c*, including :

SDA_RealSpectralInverse - Spectrum inverse on real time domain data
SDA_ComplexSpectralInverse - Spectrum inverse on complex time domain data
SDA_FdInterpolate - Interpolate a spectrum, to change pitch

The analog modulation functions are found in the file *mod_a.c* and include amplitude, frequency and delta modulation techniques. The following digital modulation functions are found in the file *mod_d.c* and include BPSK, D-BPSK, QPSK, FSK, CPFSK and QAM functions.

SigLib also includes some telecommunications utility functions, including SDA_BitErrorRate, in the file *comms.c*.

The vast majority of the (de)modulation functions use a look up table to store the carrier wave that is modulated. This wave form should be of suitable resolution (i.e. a lower frequency) so that it can adequately represent the carrier wave with minimal distortion. It is common to use make the frequency a multiple of the frequency of the signal in the look up table.

As with most SigLib functions, where there is a requirement to provide a frequency (e.g. a carrier frequency) then this is always normalized to 1.0 Hz.

Notes on detecting and extracting a modulated data stream

The following notes are intended to guide the engineer in understanding how to successfully demodulate and decode a modulated data stream. Since Version 8.0 of SigLib, most of this functionality has been incorporated into the `xxx_CostasQamDemodulate` functions.

The majority of the digital modulation functions modulate or demodulate a single symbol of data. Some functions are also provided with versions that modulate or demodulate a complete byte of data onto carrier. The latter functions are indicated by the function name suffix "Byte".

When demodulating a single symbol it is necessary to synchronize both the timing of the samples so that they align with the individual symbols of modulated data and also the individual bits into bytes. For the synchronization of the samples into symbols, the SigLib functions use a counter called the RxSampleClock. This is initialized in the initialization (`SIF_`) functions and it is used in the demodulation functions. If any parameters of the demodulation function (e.g. filter specifications etc.) then the

RxSampleClock variable must be modified accordingly. Functions that demodulate complete bytes of information also include a bit clock that will also need to be modified should the demodulator or timing be modified in any way.

For the decoding of a modulated data into the bit stream, there are three main parts, as follows.

Part 1 - Extraction of the timing of the modulated symbols. There are several techniques but essentially it is necessary to extract the start of each symbol. The most common method for doing this is to use a Phase Locked Loop (PLL) to track the incoming carrier phase.

With FSK, this tracking can easily be performed with two simple filters that have a centre frequency that is the same as the signals being detected. Threshold detectors are placed on the output of each filter and if either of them go over a threshold then you know when the symbol has started. For short sequences (e.g. Caller ID), once locked on, the application knows the period and just keeps sampling. For longer sequences it is necessary to track the incoming signal to account for differences between the transmit and receive sample clocks. These filters can be called directly within SigLib.

In order to continuously track the timing of an incoming signal it is necessary to use a FIFO buffer arrangement that allows for the adjustment of the timing on a per sample basis. This functionality is supported through the SigLib xxx_FIFODelay functions.

Part 2 - Demodulation of the data. This is a direct call to the SigLib functions. For many applications it is necessary to use a sample rate that results in a non integer number of samples per symbol. Many of the SigLib functions have been written to process data in this way however for each element that it processed it is still necessary to process an integer number of samples. This is easily achieved by using an array to hold successive numbers of integer samples. For example, a common requirement is for a 1200 baud system sampled at 8 kHz. This requires 6.667 samples per symbol, which is a non-integer number. Over 3 symbol periods, the total is 20 samples, which is an integer and can therefore be broken up into the following numbers of samples : 7, 6, 7 to give the nearest integer equivalent. Storing these values in an array and stepping through it on successive samples provides an efficient method of generating the required values.

Part 3 – Re-assembly of the bits into bytes. The first thing is that, in general, start and stop bits are not transmitted, so saving bandwidth. These are extracted at the transmitter and added back at the receiver. A common technique is to use a synchronising pattern at the front of the data stream, which is often a consecutive sequence of ones. There are many different techniques for this and each needs to be coded separately. This part is not strictly speaking DSP but is often performed on the DSP.

One final point to note is that all timing and demodulation functions use filters which delay the signals. One of the most common causes of timing errors is the incorrect allowance for the delays through these filters.

Digital Acoustic Effects

Use of the pitch shifting and echo generation functions can give some very powerful digital sound effects. One other effect (distortion) can be produced, by combining the SDA_Multiply and SDA_Clip functions. The SDA_Multiply function produces a gain on the signal, which is then clipped, producing stronger harmonics of the fundamental. In order to maintain a constant overall gain and an approximate linear distortion effect, the following values can be used :

Distortion factor	Gain	Clip value
0	**1.0**	**1.0**
1	**3.0**	**0.35**
2	**6.0**	**0.3**
3	**8.0**	**0.25**
4	**12.0**	**0.2**
5	**20.0**	**0.2**
6	**50.0**	**0.2**
7	**100.0**	**0.2**
8	**400.0**	**0.2**
9	**1000.0**	**0.2**

Complex Vector Functions

SigLib supports complex vector functionality for creating and processing complex vectors in rectangular and polar formats. These functions are prefixed with the string SCV_ and include square root, inverse, conjugate, magnitude and standard maths functions such as multiply, add, subtract and divide etc.

The complex vector data types (SLComplexRect_s and SLComplexPolar_s) are all defined in the file *siglib_processors.h*.

Compiling And Using SigLib Functions

All the processor specific configuration information is held at the top of the *siglib_processors.h* file and it is here that any processor specific modifications should be made. The vast majority of the supported compilers generate specific defined constants that are used by the SigLib header file to provide device dependent configurations. To create a new configuration for an unsupported processor it is usually best to start from a known working configuration. This should be duplicated within *siglib_processors.h*, then the `#if` comparison and any other requirements can be modified accordingly.

All the data used in the SigLib library uses the typedef'd (within the file *siglib_processors.h*) data type `SLData_t`, for most general purpose processors, the double data type is the standard for the floating point maths library or coprocessor. For DSP devices the float data type tends to be processed in a single processor clock cycle, where as double precision data often requires library function calls. For most embedded applications it will therefore be better to typedef `SLData_t` to `float`, rather than `double`. Fixed point data types are typedef'd to type `SLFixData_t`.

If the SigLib data types are modified within *siglib_processors.h* then it is usually necessary to make other modifications to support this. As an example of these modifications, let us consider changing the TMS320C6x configuration to support double and long data types.

The first step is to change the underlying typedefs, as follows :

```
typedef long            SLFixData_t;
typedef double          SLData_t;   /* Declare data types */
```

The next step is to modify the numerical limits, for example `SIGLIB_FIX_MAX` will need to be modified as follows :

```
#define SIGLIB_FIX_MAX ((SLFixData_t)2147483647) /* Maximum fixed-
point value */
```

Finally it is necessary to select the desired version of the standard math functions, such as `SDS_Sin`, as follows :

```
#define SDS_Sin(a) ((SLData_t)sin(a)) /* Define standard math
operators */
```

Following these modifications it is necessary to rebuild the appropriate library.

SigLib includes either compilation and librarian batch files or makefiles for all the supported devices. In general, SigLib is compiled with the maximum warning levels possible for each compiler and should not give any warning messages. There are situations however where some compilers will generate warnings for code that is perfectly correct and compiles correctly on other compilers.

A very small number of the SigLib functions use global variables and structures all these variables and structures are prefixed by the string "`siglib_numerix_`", in order to be compliant with the TMS320 Algorithm Standard.

When a standard library function is called by SigLib (for example `malloc`), the return values are verified and the appropriate SigLib error code returned. It is the responsibility of the calling application to manage the error. For Windows, a dialogue box is the usual method of indicating an error, in DOS, a `printf` is usually used and on an embedded DSP, an error flag may need to be set.

The SigLib error messages are :

`SIGLIB_NO_ERROR`	No error occurred in function
`SIGLIB_ERROR`	A generic SigLib error has occurred
`SIGLIB_MEM_ALLOC_ERROR`	A memory allocation error occurred
`SIGLIB_PARAMETER_ERROR`	A function parameter was incorrect
`SIGLIB_FILE_ERROR`	File open error
`SIGLIB_NO_PHASE_CHANGE`	No phase change detected
`SIGLIB_DOMAIN_ERROR`	A domain error has been detected

SigLib allows for memory accesses to be selected between the use of arrays or pointers. The macro `SIGLIB_ARRAY_OR_PTR` can take the following options :

`SIGLIB_POINTER_ACCESS`	Use pointers for memory accesses
`SIGLIB_ARRAY_ACCESS`	Use arrays for memory accesses

Pointer access is the default mode however array accesses are used in some functions for the TMS320C6000 DSPs. This functionality is selected on a per processor basis in the file *siglib_processors.h*. This is a project wide optimisation and so care must be shown when using it. The use of array indexing can help compilers optimise the code execution pipeline to optimise the performance. This performance increase is often at the expense of code size. For a particular application it may be useful to try both methods to find which provides the best performance. SigLib does not provide this option for all functions; for example, initialisation functions only use one method because they are designed to be used during initialisation and not while processing real time data. Code loops that contain conditional testing (e.g. if statements) or contain function calls do not support this optimisation option.

SigLib Memory Management

Embedded processors, such as DSP devices, usually have a segmented memory architecture that contains a mixture of internal memory and external SRAM or DRAM. Each of these sections allow the trade-off of speed for memory size. The internal memory typically provides the highest speed but the smallest size. It is therefore important to optimise the use of internal memory. It should be noted that general purpose PCs and workstations typically have a flat memory architecture so they do not have the same requirements.

When building an application using C, all of the functions in a single source file are treated as a single indivisible module so if the top level program calls one function then all of the functions in the module will be linked into this program. In order to allow the linker to only include the required functions it is necessary to split the functions into separate files but with over 1000 functions in the SigLib library, this would lead to an unmanageable number of files.
The Numerix' solution is to have the functions coded into a small number of files but we provide a utility (splitfile) to split the functions into separate files so that the linker does not need to load all of the functions that were located in an original source file. Splitfile is located in the siglib\utils directory and can be compiled to run under Windows and Unix operating systems.

Splitfile uses the ASCII file separator character (0x1c) to delineate the functions. This character is located within the original source files at the appropriate locations. The output filenames are given a number, starting at 0 and are placed in the SplitFiles directory.

Splitfile also generates a makefile to build the split library. The makefile is generated from a prototype makefile that is specified as the first command line parameter. There are a number of prototype makefiles provided for most architectures, these are located in the siglib\src\SplitPropotypes directory. The prototypes include a place holder character (also 0x1c) located at the position that stores the object files. Please ensure that the SplitFiles directory is a sub-directory of siglib\src before executing splitfile. The prototype makefiles can be modified as required but it is important not to delete the place holder character.

The second command line parameter for splitfile is the extension for the required object files. Most Windows compilers use .obj, Linux uses .o and the Analog Devices compilers use .doj.

The original SigLib source files to be split are listed in the file "srcfiles.txt", which is located in the siglib\src directory.

Please note that the splitfile utility does not split every single function into a separate file because some functions use shared data. A good example of this is the modulation and demodulation functions that use shared look-up tables.

The following example shows how to use splitfile with the Texas Instruments' TMS320C6x C compiler :

```
C:\SigLib\src>..\utils\splitfile SplitPrototypes\makefile.c6x obj

C:\SigLib\src\SplitFiles>gmake
```

For further details on specific memory issues for your DSP please refer to the appropriate section of this manual.

Using Standard C Library Functions

Some of the SigLib functions call the standard library functions, for example sin, cos, log, malloc, free etc. All of these stdio functions are accessed through SigLib macros and this allows ease of portability between platforms, processors and between different word lengths on a particular processor (e.g. between sin() or sinf()). The required stdio function can be chosen, for a particular application, by changing the appropriate definition in *siglib_processors.h*. The complete list of SigLib macros is included in the SigLib Reference Manual.

SigLib Debugging

Many of the SigLib library functions include additional debug logging information that can be useful for understanding how the functions work.
This information can be enabled via the #defining `SIGLIB_ENABLE_DEBUG_LOGGING` to '`1`' in *siglib.h*.

The logging information is written to the file *siglib_debug.log*.

Rebuilding The SigLib Library

The SigLib directory includes the following batch files to recompile and rebuild the library.

Batch files to rebuild the entire library - call *cfxx.bat* and *lfxx.bat*.
remakeMSVC.bat - Microsoft MSVC 64 bit compiler
remake6x.bat - TI TMS320C6000 - little endian
remake6xe.bat - TI TMS320C6000 - big endian

Other Build Files
SigLib.pjt - CodeWright project file
SigLib.dsw - Microsoft Visual C/C++ project file for rebuilding the static and dynamic (DLL) libraries
SigLib.dpj - Analog Devices Visual DSP project file

The examples directory also contains compilation files for Microsoft and GCC compilers.

If the library is ever being rebuilt for a new microprocessor, memory model or parameter passing model then it is essential that all existing object and library files are removed from the SigLib directory to avoid incompatibilities.

Using SigLib With The Texas Instruments Inc. TMS320C6000 Family

SigLib has been tested with the TI Code Generation Tools V8.x.

In order to use SigLib with the TI CGT it is necessary to configure the necessary environment vairables. If, for example under Windows, SigLib is installed in the folder `c:\siglib` then set the following environment variables in a batch file use the following :

```
SET SIGLIB_PATH=c:\siglib
```

Ensure the standard TI environment variable `C6X_C_DIR` points to the install directory for the CGT.

To rebuild SigLib under Windows, use the following commands :

```
> cd C:\siglib\src
> remake6x
```

To build an example, use the following commands :

```
> cd C:\siglib\Examples\CExamples
> cl6x -mv6600 --c99 -q -o3 -op3 -mt -mh -mi -pds1111 matrix.c -I
%SIGLIB_PATH%\include -I %C6X_C_DIR%\include -z -l %SIGLIB_PATH%
\lib\C6x\siglib.lib -l %C6X_C_DIR%\lib\rts6600_elf.lib -o matrix.out
```

SigLib uses the "memory section" header file (*siglib_ti_memory_sections.h*) that allows individual functions to be located into different memory sections. By default, all of the SigLib functions are located in the memory section "text" however alternative sections can be created in the linker command file and the functions placed in them. For many applications, initialisation functions are only executed during start-up, at which time execution performance is not as critical as during run-time. In order to save internal memory it is possible to move initialisation functions to external memory.

Please remember that if code is relocated in this way then it might be necessary to use the large memory model through the use of the compiler option `-ml`*n*.

SigLib also allows for memory arrays to be aligned on suitable boundaries for parallel data reads and writes. The macro `SIGLIB_ARRAYS_ALIGNED` can be set to either `SIGLIB_FALSE` or `SIGLIB_TRUE` to disable or enable this functionality, as appropriate. This functionality currently only supported by TMS320C6000 compiler and arrays are aligned on a 64 bit boundary so that the compiler can use the `LDDW` instruction for loading two parallel 32 bit floating point values. Please be aware that this is a project wide optimisation and so care must be shown when using it. By default, this functionality is disabled. For a discussion of aligned and non aligned memory accesses please refer to the TMS320C6000 Programmer's Guide (SPRU198D) Chapter 8 pages 37 to 39. For a discussion of memory bank conflicts please refer to the TMS320C6000 Programmer's Guide (SPRU198D) Appendix A page 12.

The TI C6000 compiler allows the properties of pointers to be more strictly defined than is possible using, for example, the const key word. SigLib uses the "restrict" keyword to inform the compiler that there is no possibility of address conflicts with any of the pointers used in the function being compiled. This functionality is enabled or disabled using the `SIGLIB_PTR_DECL` macro that is defined in the file *siglib_processors.h*. For the C6x DSPs, the following definition is used :

```
#define     SIGLIB_PTR_DECL         restrict
```

Please note that the `SIGLIB_PTR_DECL` macro is implemented as either `SIGLIB_INPUT_PTR_DECL` or `SIGLIB_OUTPUT_PTR_DECL` in *siglib_processors.h*. Please refer to the section entitled “The SigLib SWIG Interface” for further information.

This is a project wide optimisation and so care must be shown when using it however it can easily be enabled and disabled in the header file *siglib_processors.h*.

For all processors / compilers that do not support this option, the string `SIGLIB_PTR_DECL` must be defined to an empty string.

The performance of code generated by the C6000 compiler often benefits from the use of arrays and offsets, as opposed to pointers, for accessing data. For many functions, SigLib allows the choice of either. Please see the section on array / pointer accesses for further information.

In order to obtain optimum performance from the C6000 architecture it is often necessary to manage the CPU pipeline. This can often be achieved using #pragma statements but these are specific to a particular application requirement and often not

to a generic library such as SigLib. These kind of specific optimisations can easily be applied to SigLib on a function by function basis, as required. For further information about optimising C code on the C6000, please refer to the TMS320C6000 Programmer's Guide (SPRU198D), which is available for download from the TI web site.

For optimum run-time efficiency, the TMS320C67x code has been compiled using `float` and `long` data types. If you would like to use higher resolution for your code then these can be changed in the SigLib processor definition header file - *siglib_processors.h*.

Fixed-point Word Length Issues

The standard vector data types for SigLib are floating-point however SigLib has been successfully used in many fixed-point DSP applications by changing typedef'd data type "`SLData_t`" to an appropriate fixed-point type.

The main concerns for programmers of fixed-point devices are scaling issues which can lead to clipping and / or overflow in the numerical word. Scaling is an application dependent consideration and the effects and work-arounds are not possible to implement in a generic fashion. As an example scaling in a large FFT will depend on the data word lengths of the memory and ALU, the order of the FFT and the magnitude of the input data.

Further details on fixed point scaling issues are available at : https://www.numerix-dsp.com/appsnotes/FixedPointSigLib.pdf.

The section below gives a worked example of how to re-code a function to support a fixed-point device (the TMS320C62x).

TMS320C62X Word Length Issues

Several members of the TI TMS320C6000 DSP family can perform mathematical operations on 16 and 32 bit fixed-point data or on floating point data through the use of the run time support library. The data format to be processed is selected in the *siglib_processors.h* file, through the `#define`'d constant `SIGLIB_C62X_FLOATING_POINT_DATA`. When using SigLib with the TMS320C6000 fixed point data formats, it is necessary to set this to '0' and predefine the data word length (16 or 32 bits) via the `#define`'d constant `SIGLIB_C62X_SHORT_DATA`. If `SIGLIB_C62X_SHORT_DATA` is defined as TRUE (1) then 16 bit data is used, otherwise 32 bit data is used.

When using SigLib on a fixed point DSP it is necessary to consider the fixed point scaling issues, the following code section shows how to code a typical SigLib function for a fixed point device. The first process is to perform the scaling operations, (1.0 - α) in this case, at the start of the algorithm, to ensure that overflow possibilities are minimised. The results of the multiplication operations also need to be right shifted by 15 bits.

```
SLData_t SDA_OnePole (SLData_t Src, SLData_t Alpha, SLData_t *State)
{
    SLData_t OneMinusAlpha;

    OneMinusAlpha = ((DSP_DATA)0x7fff) - Alpha;
    *State = (((*State) * Alpha) >> 15) +
                  ((Src * OneMinusAlpha) >> 15);
    return (*State);
}                 /* End of SDA_OnePole () */
```

Compiling SigLib For Use On The StarCore DSP

The default functional mode for using SigLib on the StarCore DSP family is to utilise floating point data. This imposes a performance penalty but allows for quick and easy development.

Following initial algorithm development with floating point data, higher performance can be achieved by changing the `SLFixData_t` and `SLData_t` data types to `'long'`, in the file *siglib_processors.h*. This change will ultimately result in fixed point scaling issues. Moving to a fixed point number scheme will also allow further optimisations to be performed such as software pipelining.

Further details on fixed point scaling issues are available at : https://www.numerix-dsp.com/appsnotes/FixedPointSigLib.pdf.

Using SigLib on the ADSP-2116x Architecture

Many SigLib functions can benefit from the use of the SIMD mode. This can be enabled at the C source level, using the following code section :

```
#ifdef __ADSP21160__
#pragma SIMD_FOR
#endif
```

Note : This code must be placed between the start of the function and any 'for' loops within the C code.

Using SigLib With Java™

For details on how to use SigLib with Java under Linux, please refer to the "Using SigLib With Java" sub-section of "Using SigLib Under Windows". The procedure is identical except that the SigLib library should be rebuilt as a shared object file (.so) rather than a Windows Dynamic Linked Library (.dll). Please refer to the quick start section for full details on using SigLib as a shared object library under Linux.

Using SigLib Under Windows™

The SigLib functionality can be accessed under Windows through either a statically or dynamically linked library. The SigLib DLL (*siglib.dll*) allows full access to all the DSP and mathematics functionality of the SigLib library from virtually any 64 bit Windows™ program. Examples and header files are included for C/C++ (*siglib.h*), Visual Basic™ (*global.bas*) Excel™, Agilent VEE™, National Instrument's LabVIEW™ etc. The SigLib DLL does not however provide access to the host dependent graphics functionality.

The functionality of these libraries is identical to the standard library and is documented in the library reference manual *ref.pdf*. The SigLib DLL uses two main data types for all information; floating point data is of type `double` and fixed point data is of type `long`. Examples are included for the DLL library, these are located in

the “DLLExamples” directory. Examples are supplied for Visual C++, Visual Basic (includes an example application and complete example globals.bas file for all the SigLib functions), Excel and HP VEE.

In the DLL, the data types `SLData_t` is a C typedef to type double (a 64 bit double precision floating point number) and `SLFixData_t` is a C typedef to type long (a signed 32 bit long integer).

Installing the Windows DLL

Prior to using the SigLib DLL (`siglib.dll`) it is necessary to copy the desired version (debug or release) from `c:\siglib\lib\Microsoft\dynamic_library` to the appropriate Windows folder.

Copy `siglib.dll` to :
`%windir%\SysWOW64\`

Microsoft Visual C++ (including Express Edition™)

The easiest way to build all of the Microsoft library build options (static/dynamic : debug/release) can be built using the batch file : *remakeMSVC.bat*.

Configuration batch files are provided (*setenv.bat*) for configuring the Microsoft compiler environment variables.

Alternatively, to build SigLib with Microsoft Visual C++, use the siglib.sln project file provided in the src folder. This allows SigLib to be built as either a static or a dynamic linked library. The libraries are located in either `c:\siglib/lib\Microsoft\dynamic_library` or `c:\siglib\lib\Microsoft\static_library\` folders.

Using SigLib With Java™

SigLib functions can be accessed via the Java Native Interface (JNI) or the Simplified Wrapper and Interface Generator (SWIG). This section describes the JNI interface. Please also refer to the section entitled "The SigLib SWIG Interface" for further information on the SWIG API.

Java does not support the ability to call C API functions directly so a Java Native Interface (JNI) wrapper layer has to be written, in C, to allow this. The wrapper supports the Java class based API and allows SigLib functions to be accessed from a Dynamic Linked Library (DLL) using the JNI.

The following discussion describes how to access a simple SigLib function (`SDS_Max`) via the JNI. This process can be expanded to access any SigLib function.

Step 1 - Create a file called `SigLibDSP.java` containing a java class called `SigLibDSP` which declares the required SigLib functions and includes a test function.

```
package SigLibDSP.Test;
/* Declare the SigLib class */
public class SigLibDSP
{
    // Declare the SigLib functions required in the application
  public static native double SDS_Max(double x, double y);

  static    // Static initializer is executed when class is
loaded
  {
    System.loadLibrary("SigLib");
  }

  public static void main(String[] args)
  {
    System.out.println("Calling SigLib function");

    double a, b, c;

    a = 1.0;
    b = 2.0;
    c = SigLibDSP.SDS_Max (a,b);

    System.out.println("SDS_Max " + a + " or " + b + " = " + c);
  }
}
```

Step 2 - Compile the class :

```
> javac SigLibDSP.java
```

Step 3 - Create the header (*.h*) file :

```
> javah -jni -o SigLibDSP.h -classpath . SigLibDSP
```

The resulting header file is shown below :

```
/* DO NOT EDIT THIS FILE - it is machine generated
*/
#include <jni.h>
/* Header for class SigLibDSP */

#ifndef _Included_SigLibDSP
#define _Included_SigLibDSP
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     SigLibDSP
 * Method:    SDS_Max
 * Signature: (DD)D
 */
JNIEXPORT jdouble JNICALL Java_SigLibDSP_SDS_1Max
  (JNIEnv *, jclass, jdouble, jdouble);

#ifdef __cplusplus
}
#endif
#endif
```

Please note : The function name has been changed by `javah` from `SDS_Max` to `Java_SigLibDSP_SDS_1Max` (note the additional "1").

Step 4 - Create a file called `SigLibDSP.c` to implement the JNI wrapper C code functions that are compatible with the Java API.

```
#include <jni.h>
#include "SigLibDSP.h"
#include <SigLib.h>
#include <stdio.h>

JNIEXPORT jdouble JNICALL Java_SigLib_SDS_1Max(JNIEnv *env,
jobject obj, jdouble x, jdouble y)
{
    return(SDS_Max(x, y));
}
```

Please note :

- There are two additional mandatory parameters : The JNI interface pointer (JNIEnv) and a reference to the calling object (jobject).
- The data types are java data types (jdouble)

Step 5 - Compile the C code into a Dynamic Link Library (DLL)
Free command line compilers are available from Microsoft for this task. The Microsoft command is :

```
> cl -I. -LD SigLibDSP.c -FeJavaSigLib.dll
```

Step 6 - Copy `JavaSigLib.dll` to your Java application directory

Step 7 - Execute the test application

SigLib DLL Example Source files

The Microsoft Visual C/C++ projects in the DLLExamples directory have been written so that they will work with either the static or the dynamic link libraries. It is possible to change from DLL to SLL linkage using the following steps :

In the Project | Settings | C/C++ | Preprocessor dialog add the following preprocessor definition : `SIGLIB_STATIC_LIB=1`

Rebuild

How To Use The DOS Examples With The SigLib DLL And Microsoft Visual C/C++

The SigLib DLLs can be accessed directly and there are several examples in the DLLExamples directory to show how this is done. An easier way to access the DLLs is to use the supplied translation library (`siglib.lib`), as used in the instructions below.

Here are the steps used with the BenchIIR example.

Ensure the DLL is installed correctly according to the section of this manual entitled "Installing the Windows DLL".

Create a new project.

File | New
Windows Console Application
Located in c:\SigLib\DLLExamples\
Called BenchIIR
Click OK
Select A "Hello World" Application
Click OK twice

Select File View and open BenchIIR.cpp

Replace the main function with the entire code from the DOS example (BenchIIR.c)

Note – do not remove the following line of code from your C++ file :

```
#include "stdafx.h"
```

Keeping the source file as .cpp makes it so much easier to configure the compiler, even if you just write pure c code as in this case.

Go to Project | Settings

Select Link

Go to the Object/library modules: line and add

```
C:\siglib\lib\Microsoft\static_library\Release
```

Do a Build | Rebuild all

That is it. If you do an F5 it will run but you will miss the output so if you open up a Command Line window on the Debug directory you will be able to run the app and see the output.

Using The SigLib DLL With Microsoft Visual Basic

The DLL is supplied with a Visual Basic example program in the folder C:\siglib\Examples\DLLExamples\VBTest.This folder includes two global definitions files that declare the SigLib functions and other details :

SigLibGlobal.bas	Visual Basic V6.0
SigLibGlobal.NET.bas	Visual Basic .NET

In the DLL, the data types `SLData_t` is a C typedef to type double (a 64 bit VB Double type) and `SLFixData_t` is a C typedef to type long (a signed 32 bit VB Integer type).

When trying to use the Visual Basic example for the SigLib DLL package with VB6 the graphics functionality is not installed by default. The following text is taken from the readme.txt file in \Common\Tools\VB\controls on the VB6 CD.

> Graph32.ocx has been updated to work properly in Visual Basic 6.0 and it requires two additional support files: gsw32.exe and gswdll32.dll. You must place the three files together in the \Windows\System directory or the control will not function properly.
>
> If you do not have these controls and wish to use these in Visual Basic 6.0, you
> can install them by:
>
> 1. Copy all of the files in this directory to your \WINDOWS\SYSTEM directory.
>
> 2. Register the controls by either Browsing to them in Visual Basic itself, or

manually register them using RegSvr32.Exe. RegSvr32.EXE can be found in the
\Tools\RegistrationUtilities directory. The command line is:

regsvr32.exe graph32.ocx

3. Register the design time licenses for the controls. To do this, merge the vbctrls.reg file found in this directory into your registry. You can merge this file into your registry using RegEdit.Exe (Win95 or WinNT4) or RegEd32.Exe
(WinNT3.51):

regedit vbctrls.reg (or other reg files associated with the controls)

Once these procedures have been completed the graphing functionality can be accessed through the "Pinacle-BPS Graph Control".

Newer versions of VB do not include the same OCXs as VB6 so the following notes are useful :

A quick search on the Microsoft web site shows that GRAPH32 and GRID32 have been removed from the latest VB but the functionality is available in other OCXs. The following URL has the details : http://support.microsoft.com/default.aspx?scid=kb;en-us;172193.

The DBGRID32.OCX is still provided and you will need to load it according to the instructions on this page :
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon98/html/vbconcontrolfilenames.asp.

Using The SigLib DLL With Microsoft Excel

Microsoft Excel does not require any additional files to access the DLL, please refer to the example provided for further information.

In the DLL, the data types `SLData_t` is a C typedef to type double (a 64 bit double precision floating point number) and `SLFixData_t` is a C typedef to type long (a signed 32 bit long integer).

Using The SigLib DLL With Agilent VEE

When using the DLL with Agilent (formerly Hewlett-Packard) VEE, the header *siglibv.h* should be used within the VEE application. The VEE header file (*siglibv.h*) can be recreated at any time from the standard SigLib header file (*siglib.h*) using the batch file *veegen.bat* in the include subdirectory. This batch file requires Microsoft Visual C/C++ to be installed to operate correctly although any C pre-processor could be used. SigLib is supplied with two VEE examples that use many of the techniques,

listed below, that are necessary for interfacing to DLLs. A free evaluation version of VEE can be obtained from the Agilent web site.

In the DLL, the data types `SLData_t` is a C typedef to type double (a 64 bit double precision floating point number) and `SLFixData_t` is a C typedef to type long (a signed 32 bit long integer).

Using The SigLib DLL With National Instruments' LabVIEW

To call SigLib functions from LabVIEW a "Call Library Function" icon should be placed on the block diagram and configured to call the SigLib function, as follows :

Library Name : *Siglib.dll*
Function Name : SigLib function
Select the appropriate parameter
Insert appropriate Function Prototype - This can usually just be copied from *siglibv.h*

In the DLL, the data types `SLData_t` is a C typedef to type double (a 64 bit double precision floating point number) and `SLFixData_t` is a C typedef to type long (a signed 32 bit long integer).

Using SigLib On PocketPCs Under Windows Mobile 2003™

SigLib can be compiled using the siglibce.vcw workspace using Microsoft Embedded Visual C++ to generate a dynamically linked library. The binary version of the library is provided with the library pre-compiled for the following configuration :

"Release model"

```
SLData_t = float
SLFixData_t = long
```

The choice of float and int for the data types has been chosen because the PocketPC does not include a math coprocessor but uses software instead so these types provide better performance than double and long. This can be changed when the full source code version of SigLib is used and the library is recompiled.

How to compile and use library on Windows Mobile 2003

In order to use SigLib under Windows Mobile 2003 and compile the example programs you will need to install the following software on your development computer :
eMbedded Visual C++ 4.0 : http://go.microsoft.com/?linkid=1396859.
eMbedded Visual C++ 4.0 SP4 : http://www.microsoft.com/downloads/details.aspx?FamilyID=4a4ed1f4-91d3-4dbe-986e-a812984318e5&displaylang=en.
Windows Mobile Developer Power Toys : http://www.microsoft.com/downloads/details.aspx?familyid=74473fd6-1dcc-47aa-ab28-6a2b006edfe9&displaylang=en.
This provides a PPC Command Shell, which is quite useful for development.
PocketPC GNUPlot : http://www.rainer-keuchel.de/wince/gnuplot-ce.html.
Numerix CE Host Library for GNUPlot : https://www.numerix-dsp.com/files. The

version supporting ‘float’ and ‘short’ data types should be installed into C:\siglib\nhl_gp_ce.

The SigLib DLL can be rebuilt using the project file siglibce.vcw. The pre-built libraries (Debug and Release) are located in : C:\siglib\lib\ARMV4WinCEdll. Ensure that device is attached and siglib.dll has downloaded correctly after it has been built. If you have the binary version of SigLib then you will need to download and register the dll manually. To do this, copy siglib.dll to a suitable folder e.g. \Windows and then use regsvr from the following web page to register it : http://www.pocketpcdn.com/articles/regsvr.html.

Build test_fft.vcw example from C:\siglib\CEExamples\test_fft
Ensure that device is attached and test_fft.exe is downloaded correctly

The test_fft application was generated using the wizard in eVC++ to generate a simple program. The code was then inserted from one of the standard examples. The only modifications that are required to the standard project are to add the directories for the include files and to add the appropriate siglib.lib file to the linker section.

Using SigLib On Android™

SigLib can be compiled using Android Studio. Android Studio support is provided for both native libraries and SWIG wrapper.

To build the native library : cd siglib/src; ./remakeandroid.sh. This script has been developed and tested under Cygwin.

To build the SWIG library : cd siglib/src ./swigjavaAndroid.sh. This script has been developed and tested under Cygwin.

An Android Studio example is provided in siglib/Examples/Android/SigLib_Graph

Using SigLib With Ch™

Ch is an embeddable C/C++ interpreter for cross-platform scripting and shell programming application. SigLib DLL and UNIX shared object functions can be called from Ch.

There are two methods for accessing SigLib from Ch. The first is to call the functions directly and the second is to call the functions in the SigLib .DLL file, which must be located in the `Windows\System32` directory. The first method allows the use of the standard SigLib API and the functions are parsed as the program executes. The second allows the pre-compiled SigLib functions to execute more efficiently.

Method 1
In order to access the SigLib functions directly from Ch you will need to add the following lines to `_chrc` in your home directory :

```
_ipath = stradd(_ipath, "C:/siglib/include;c:/siglib/src;");
_fpath = stradd(_fpath, "c:/siglib/src;");
```

To access the SigLib functionality from Ch you should include the header file : *siglibch.h*.

Method 2
Please add the following lines to `_chrc` in your home directory :

```
_ipath = stradd(_ipath, "C:/siglib/include;");
_lpath = stradd(_lpath, "C:/siglib/lib/ch;");
```

Note
Please ensure that the appropriate section in the `_chrc` file in your home directory is configured correctly for your C compiler tools.

The SigLib SWIG Interface

SWIG (Simplified Wrapper and Interface Generator) - www.swig.org – is an API that allows DLLs written in C/C++ to be accessed from a variety of high-level (Java, C# etc.) and scripting (Perl, PHP, Python, Tcl, Ruby etc.) languages.

The SigLib SWIG API allows the SigLib functions to be accessed from any language supported by SWIG and examples are provided for C#, Java and Perl.

In order to use the SigLib SWIG API it is necessary to understand how SigLib accesses and processes data. SigLib supports the standard C API for functions and the data is either passed in one of two ways :

- As directly accessed parameters – e.g. a constant value that is used by the function
- As an indirectly accessed array of data values, accessed through a pointer

When supplying an array of data to the SigLib function it is necessary to provide the following information :

- A pointer to the start of the data array
- The length of the array

The array for the SigLib SWIG data must be declared as type : `SWIGTYPE_p_double`.

The SWIG API build files are supplied in the /SigLib/SWIG folder, which contains the following batch files for rebuilding the SigLib SWIG API :

cleanup.bat - Deletes all files that are not required to build the SWIG APIs
swigc#.bat - To build the SWIG API for C#
swigjava.bat - To build the SWIG API for Java
swigperl.bat - To build the SWIG API for Perl
swigpython.bat - To build the SWIG API for Python

In order to build the SWIG APIs you will need to use a C compiler. The batch files provided work with the free Microsoft Visual C++™.
When using these batch files it may be necessary to edit them to reflect the installation paths and version numbers for your chosen language.

Examples for using the SigLib SWIG API are provided in the following directories :

/examples/c#examples
/examples/javaexamples
/examples/perlexamples

Please also refer to the section entitled "Using The SigLib DLL With Applications That Do Not Support C/C++ Structures, Enumerated Types And #define Statement" for further information.

Within *siglib.h* the input and output characteristics of the pointers are declared using the #defines `SIGLIB_INPUT_PTR_DECL` and `SIGLIB_OUTPUT_PTR_DECL` as shown in the following table :

SigLib Type	SWIG Type
`SIGLIB_INPUT_PTR_DECL`	`INPUT`

SIGLIB_OUTPUT_PTR_DECL	OUTPUT

The SWIG API is not safe from array overflow so SigLib includes a file that can be used to replace the standard SWIG `carryas.i` file. This file is located in the folder :

`/siglib/SWIG/carrays.`

Using The SigLib DLL With Applications That Do Not Support C/C++ Structures, Enumerated Types And #define Statements

When calling the SigLib DLL from applications that do not support C/C++ structures it is possible to replicate the functionality by passing all the members of the structure to the function as separate parameters of the appropriate type.

SigLib uses the following data structures to represent complex data types :
Complex Cartesian (Rectangular) numbers :

```
typedef struct
{
      SLData_t    real;
      SLData_t    imag;
} SLComplexRect_s;
```

Complex Polar numbers :

```
typedef struct
{
      SLData_t    magn;
      SLData_t    angle;
} SLComplexPolar_s;
```

Enumerated types in C/C++ replace the definition with an integer number that defaults to a linear sequence starting at 0. SigLib uses the following enumerated types and the list includes the value associated with the following definitions :

Enumerated type : `SLWindow_t`, used for window types :

```
SIGLIB_HANNING                                0
SIGLIB_HAMMING                                1
SIGLIB_GENERALIZED_COSINE                     2
SIGLIB_BLACKMAN                               3
SIGLIB_BARTLETT_TRIANGLE_ZERO_END_POINTS      4
SIGLIB_BARTLETT_TRIANGLE_NON_ZERO_END_POINTS  5
SIGLIB_KAISER                                 6
SIGLIB_BLACKMAN_HARRIS                        7
SIGLIB_RECTANGLE                              8
```

Enumerated type : `SLSignal_t`, used for signal generation types :

```
SIGLIB_SINE_WAVE                  0
SIGLIB_COSINE_WAVE                1
SIGLIB_WHITE_NOISE                2
SIGLIB_GAUSSIAN_NOISE             3
SIGLIB_CHIRP_LIN                  4
SIGLIB_CHIRP_NL                   5
SIGLIB_SQUARE_WAVE                6
SIGLIB_TRIANGLE_WAVE              7
SIGLIB_IMPULSE                    8
SIGLIB_IMPULSE_STREAM             9
SIGLIB_STEP                       10
SIGLIB_PN_SEQUENCE                11
```

Enumerated type : SLSignalFillMode_t, used for signal buffer fill modes :

```
SIGLIB_FILL                     0
SIGLIB_ADD                      1
```

Enumerated type : SLSignalSign_t, used for signal data types :

```
SIGLIB_SIGNED_DATA              0
SIGLIB_UNSIGNED_DATA            1
```

Enumerated type : SLEcho_t, used for echo types - feedbackward / feedforward :

```
SIGLIB_ECHO                     0
SIGLIB_REVERB                   1
```

Enumerated type : SLRoundingMode_t, used for rounding of data mode :

```
SIGLIB_ROUND_UP                 0
SIGLIB_ROUND_TO_NEAREST         1
SIGLIB_ROUND_DOWN               2
SIGLIB_ROUND_TO_ZERO            3
SIGLIB_ROUND_AWAY_FROM_ZERO     4
```

Enumerated type : SLModuloMode_t, used for data modulo types :

```
SIGLIB_SINGLE_SIDED_MODULO      0
SIGLIB_DOUBLE_SIDED_MODULO      1
```

Enumerated type : SLClip_t, used for clipping the data values :

```
SIGLIB_CLIP_ABOVE               1
SIGLIB_CLIP_BOTH                0
SIGLIB_CLIP_BELOW               -1
```

Enumerated type : SLThresholdMode_t, used for data threshold and clamping types :

```
SIGLIB_SINGLE_SIDED_THRESHOLD   0
SIGLIB_DOUBLE_SIDED_THRESHOLD   1
```

Enumerated type : SLLevelCrossingMode_t, used for zero crossing types :

```
SIGLIB_POSITIVE_LEVEL_CROSS     0
SIGLIB_NEGATIVE_LEVEL_CROSS     1
SIGLIB_ALL_LEVEL_CROSS          2
```

Enumerated type : SLArbitraryFFT_t, used for arbitrary length FFT types :

```
SIGLIB_ARB_FFT_DO_CZT           0
SIGLIB_ARB_FFT_DO_FFT           1
```

Enumerated type : SLParity_t, used for parity checking types :

```
SIGLIB_NO_PARITY                0
SIGLIB_EVEN_PARITY              1
SIGLIB_ODD_PARITY               2
```

Enumerated type : SLELGTriggerTiming_t, used for early-late gate trigger types :

```
SIGLIB_ELG_TRIGGER_START        0
SIGLIB_ELG_TRIGGER_MIDDLE       1
```

Enumerated type : SLCostasLoopFeedbackMode_t, used for Costas loop feedback modes :

```
SIGLIB_COSTAS_LOOP_MULTIPLY_LOOP        0
SIGLIB_COSTAS_LOOP_POLARITY_LOOP        1
SIGLIB_COSTAS_LOOP_HARD_LIMITED_LOOP    2
```

Enumerated type : SLIIRNormalizedCoeffs_t, used for normalized filter coefficient types :

```
SIGLIB_BUTTERWORTH_IIR_NORM_COEFFS   0
SIGLIB_BESSEL_IIR_NORM_COEFFS        1
```

Enumerated type : `SLFilterBandTypes_t`, used for filter band types :

```
SIGLIB_FILTER_LOW_PASS               0
SIGLIB_FILTER_HIGH_PASS              1
SIGLIB_FILTER_BAND_PASS              2
SIGLIB_FILTER_NOTCH                  3
```

Enumerated type : `SL3x3Coeffs_t`, used for zero crossing 3x3 filter coefficient types :

```
SIGLIB_EDGE_ENHANCEMENT              0
SIGLIB_HORIZONTAL_EDGE               1
SIGLIB_VERTICAL_EDGE                 2
```

Enumerated type : `SLSignalCoherenceType_t`, used for signal coherence types - used in order analysis :

```
SIGLIB_SIGNAL_COHERENT               0
SIGLIB_SIGNAL_INCOHERENT             1
```

Enumerated type : `SLFindType_t`, used for find types :

```
SIGLIB_FIND_GREATER_THAN_ZERO              0
SIGLIB_FIND_GREATER_THAN_OR_EQUAL_TO_ZERO  1
SIGLIB_FIND_EQUAL_TO_ZERO                  2
SIGLIB_FIND_LESS_THAN_ZERO                 3
SIGLIB_FIND_LESS_THAN_OR_EQUAL_TO_ZERO     4
SIGLIB_FIND_NOT_EQUAL_TO_ZERO              5
```

Enumerated type : `SLCompareType_t`, used for compare types :

```
SIGLIB_NOT_EQUAL                     0
SIGLIB_EQUAL                         1
```

C and C++ allows the use of a command (#define) to declare constant values, applications that do not support this command will need to use the correct constant value :

NULL pointer constants

SIGLIB_NULL_DATA_PTR	0	Null pointer to SLData_t
SIGLIB_NULL_FIX_DATA_PTR	0	Null pointer to SLFixData_t
SIGLIB_NULL_COMPLEX_RECT_PTR	0	Null pointer to SLComplexRect_s
SIGLIB_NULL_COMPLEX_POLAR_PTR	0	Null pointer to SLComplexPolar_s
SIGLIB_NULL_ARRAY_INDEX_PTR	0	Null pointer to SLArrayIndex_t

Array index constants

SIGLIB_INVALID_ARRAY_INDEX	-1	Invalid SLArrayIndex_t
SIGLIB_SIGNAL_NOT_PRESENT	-1	Signal not present in array
SIGLIB_SEQUENCE_NOT_DETECTED	-1	Sequence not detected
SIGLIB_LEVEL_CROSSING_NOT_DETECTED	-1	Level crossing not detected
SIGLIB_NO_PARITY_ERROR	-1	No parity error detected

SIGLIB_PI	π (3.141592653589793239….)
SIGLIB_TWO_PI	$2.0 * \pi$
SIGLIB_HALF_PI	$\pi / 2.0$
SIGLIB_QUARTER_PI	$\pi / 4.0$
SIGLIB_INV_TWO_PI	$1.0 / (2.0 * \pi)$
SIGLIB_INV_SIXTEEN	$1.0 / 16.0$
SIGLIB_SQRT_TWO	$\sqrt{2.0}$
SIGLIB_INV_SQRT_TWO	$1.0 / (\sqrt{2.0})$
SIGLIB_INV_SQRT_TWO_PI	$1.0 / (\sqrt{(2.0 * \pi)})$
SIGLIB_EXP_MINUS_ONE	$e^{-1.0}$

SIGLIB_ON	1	Switch is ON
SIGLIB_OFF	0	Switch is OFF
SIGLIB_TRUE	1	Switch is TRUE
SIGLIB_FALSE	0	Switch is FALSE
SIGLIB_NULL	0	Switch is ZERO
SIGLIB_ENABLE	1	Switch is ENABLED
SIGLIB_DISABLE	0	Switch is DISABLED
SIGLIB_POSITIVE	1	Number has positive sign
SIGLIB_NEGATIVE	0	Number has negative sign
SIGLIB_CLIP_ABOVE	1	Buffer data clip direction
SIGLIB_CLIP_BOTH	0	
SIGLIB_CLIP_BELOW	-1	
SIGLIB_FIRST_SAMPLE	0	State control used by
SIGLIB_NOT_FIRST_SAMPLE	1	SigLib functions

SIGLIB_GOERTZEL_DELAY_LENGTH	2	State array length for G. filter
SIGLIB_RESONATOR_DELAY_SIZE	3	Size of resonator state vector
SIGLIB_IIR_DELAY_SIZE	2	Size of IIR filter state vector
SIGLIB_IIR_COEFFS_PER_BIQUAD	5	Number of coefficients per biquad

Digital Communications Constants

```
SIGLIB_BPSK_BITS_PER_SYMBOL   1
SIGLIB_BPSK_NUMBER_OF_PHASES  2
SIGLIB_BPSK_BIT_MASK          0x1

SIGLIB_QPSK_BITS_PER_SYMBOL   2
SIGLIB_QPSK_NUMBER_OF_PHASES  4
SIGLIB_QPSK_BIT_MASK          0x3

SIGLIB_OPSK_BITS_PER_SYMBOL   3
SIGLIB_OPSK_NUMBER_OF_PHASES  8
SIGLIB_OPSK_BIT_MASK          0x7

SIGLIB_QAM16_BITS_PER_SYMBOL  4
SIGLIB_QAM16_NUMBER_OF_PHASES 16
SIGLIB_QAM16_BIT_MASK         0x0f
```

Compiler / Processor Constant Values

SIGLIB_FIX_MAX	The maximum fixed-point data value that can be handled
SIGLIB_FIX_WORD_LENGTH	The length of fixed-point data type
SIGLIB_EPSILON	The smallest value such that `(1.0 + SIGLIB_EPSILON) != 1.0`
SIGLIB_MIN_THRESHOLD	A sample value close to zero but above numerical error floor
SIGLIB_MIN	Minimum positive value
SIGLIB_DB_MIN	20*Log_{10} (minimum negative value)
SIGLIB_MAX	Minimum realistic positive value – a test for overflow in a real application.
SIGLIB_INV_MAX	1.0 / (SIGLIB_MAX)

SigLib also includes a number of "standard" word width definitions :

SIGLIB_NIBBLE_LENGTH	4	Standard 4 bit nibble
SIGLIB_BYTE_LENGTH	8	Standard 8 bit byte
SIGLIB_SHORT_WORD_LENGTH	16	Standard 16 bit short word
SIGLIB_LONG_WORD_LENGTH	32	Standard 32 bit long word
SIGLIB_LONG_LONG_WORD_LENGTH	64	Standard 64 bit long long word
SIGLIB_SHORT_WORD_MAX	32767	Maximum value in a short word
SIGLIB_LONG_WORD_MAX	2147483647	Maximum value in a long word

SigLib Utility Programs

These applications are located in the '/SigLib/utils' directory.

Polyphase Filter Coefficient Conversion Utilities

GENPP

This application converts converts a single FIR filter coefficient file called *filter.dat* to the appropriate number of poly-phase filter banks, in the file *ppfilt.dat*. *Filter.dat* should be a comma delimited ASCII file and can be generated from Digital Filter Plus. Genpp requires the user to enter the number of stages to be generated.

Usage :

```
> genpp number_of_stages
```

Credits

Our thanks go to the following, for their support, assistance, advice and/or code.

Lois Edwards

Dr. Ian Burnett	University of Wollongong, Australia.
Chris Butler	DSP Technology, UK
Alan Campbell	Texas Instruments Inc., USA, formerly with Brookhaven National Laboratories, USA and CERN Laboratories, Europe.
Albert Chau	Canada
Dwight Harm	Trax Softworks Inc., USA
Richard Hathway	Milspec Systems
Mika Kuoppala	Image Soft Oy, Finland.
John Macpherson	Baker Hughes INTEQ, USA.
Shawn McCaslin	Cicada Semiconductor, USA
Jim Schley-May	3rd Ear Audio, USA
Rob Whitton	Formerly with Loughborough Sound Images, UK.
David Troendle	Louisiana State University
Chad Cloman	Data Fusion Corp., USA
Dave Reynolds	USA
Freddy Ben-Zeev	Chief Software Architect, GuideTech, USA
Chris Tisone	SEI, Inc.
Pieter du Preez	Wavecom, Switzerland
Alan Cohen	Duke River Engineering
Andrew Biagioni	Duke River Engineering
Brian Zhang	ION Geophysical Inc
Alain Weiler	
Ali Rahman	MB Dynamics
Roman Ukhov	cmake build file