DSPLib For The Hitachi SH1 7000 Series Microcontroller

Application Note

19-031/1.0

June 1996

Preface

This manual is intended as a user guide for the Digital Signal Processing library DSPLib. DSPLib has been developed for use with the Hitachi 'Super H' range of RISC microcontrollers, and in particular the SH-1 device.

The organisation of this manual is as follows :

Overview:	Gives an overview of the library and its performance.
Function Descriptions:	Describes the use of each function, including background theory and software examples where necessary.
DSPLib Implementation:	Takes the reader through a typical DSPLib implementation. Includes notes on hardware interfacing to the SH-1 as well as examples of typical software routines.

Related Manuals

The reader should refer to the following Hitachi manuals when using DSPLib:

DSPLib For SH User Guide

SH7032/SH7034 Hardware Manual

SH7000 Series Programming Manual

SH Series C Compiler

SH Series Cross Assembler

H Series Linkage Editor

For software development tools, contact your Hitachi sales office.

TABLE OF CONTENTS

1.0 Overview

1.1 General Description

DSPLib represents a number of standard 'off the shelf' **fixed point** DSP operations which can be used either stand alone or in series to form an optimised DSP process which will run on a microcontroller. The functions offered by the library come in five distinct groups.

Fast Fourier Transforms

Window Functions

Filters

Convolution & Correlation

Miscellaneous

The library offers each function in both ANSI C source code and also optimised SH assembler. The C code is provided only for debugging and reference. It is not recommended that the C code be used in a finished system due to the inefficiencies introduced by the C compiler.

As well as the DSP functions, DSPLib offers a number of example programs to demonstrate to the user how easily DSPLib can be integrated into standard ANSI C code.

In order to reduce development times, this document has been written to take the reader through each function indicating typical pitfalls, as well as hints to improve execution speed. The document forms a type of user guide for the main part, and towards the end indicates some hardware design considerations as well when designing with the Hitachi SH series.

All the functions offered by DSPLib do not use any local static variables, thus enabling the functions to be called from interrupt routines without affecting the operation of the main program.

In order to use DSPLib with the Hitachi SH series of microcontrollers, the user will require the Hitachi tool chain to build programs. This tool chain consists of the following :-

SH Series Cross Assembler

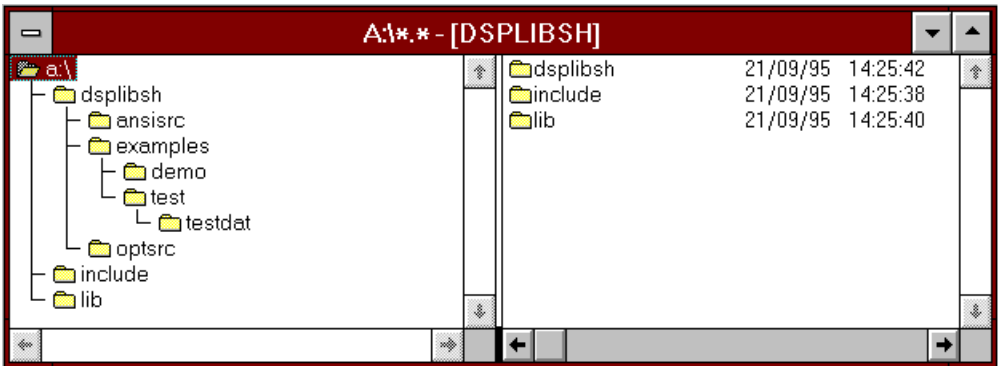
SH Series C Compiler

H Series Linkage Editor

Although a set of GNU tools are available for the Hitachi SH series, the optimised SH assembler in DSPLib is not compatible, and is therefore unsuitable. The ANSI C code can be compiled by the GNU tool, but will result in a very inefficient implementation.

DSPLib is supplied on a 3.5” disk in MS-DOS format. The disk consists of a file ensigma.lib which contains the entire library ready built in object form. Also on the disk is a header file called ensigdsp.h, as well as the ANSI C and optimised assembler functions. Some example code is also supplied on the disk, details of which may be found in the related Ensigma documentation. A number of test routines are included to allow the user to verify correct operation.

The directory structure of DSPLib is as follows :-



When installing DSPLib, create a directory for the above tree to sit in. The Hitachi tools should be installed in a separate directory, and the path then set such that the include and lib directories of DSPLib are visible when building programs. To copy the library onto a hard drive called c:\, place DSPLib into drive A:\ and type the following :-

```
XCOPY A:\* C:\shc\shc2.0 /s
```

Further details on installation and library building are included in the related Ensigma documentation.

1.1.2 Note About Data Types

DSPLib makes use of a number of ANSI C data types. Representation of these types on an SH-1 device are as follows :-

Byte 8 bits

Word 16 bits

Long 32 bits

Float 32 bits

Double 64 bits

1.2 Function Timings

The execution times are based on a 20Mhz SH1 device, and should be adjusted accordingly for other clock speeds.

1.2.1 FFT Timings

The timings are given below in **milliseconds** for the execution speed of the FFT functions.

Not In Place				
	Forward FFT		Inverse FFT	
Size	Complex	Real	Complex	Real
128	1.94	1.06	2.19	1.20
256	4.43	2.37	4.94	2.66
512	9.98	5.29	11.00	5.88
1024	22.23	11.70	24.28	12.88

In Place				
	Forward FFT		Inverse FFT	
Size	Complex	Real	Complex	Real
128	1.94	1.06	2.20	1.19
256	4.46	2.37	4.97	2.64
512	10.03	5.32	11.06	5.86
1024	22.39	11.75	24.44	12.83

1.2.2 Filter Timings

The timings are given below in **microseconds** for the execution speed of the filter functions. The multiple sample routines are quoted in a per sample format, computed by dividing the computational time for 100 samples by 100.

	Function			
No. Of Coeffs	FIR	LMS	FIR1	LMS1
5	8.6	24.9	17.2	34.7
10	10.3	40.6	18.9	50.4
100	46.1	328.4	54.7	338.2

	Function			
No. Of Biquads	IIR	DIIR	IIR1	DIIR1
1	10.5	40.1	18.1	48.0
2	19.7	77.6	28.8	86.4
20	185.5	752.8	221.4	784.3

1.2.3 Convolution & Correlation Timings

The timings are given below in **microseconds** for the execution speed of the various convolution and correlation functions offered by DSPLib. The timings are quoted per output sample, thus found by dividing the time to produce n output samples by n.

The table shows the computational time per output sample for a particular input array size iw. It should be noted that iw is always the smaller of the two input arrays.

	Routine			
Size Of iw	**ConvPartial**	**ConvCyclic**	**CorrCyclic**	**Correlate**
5	**9.7**	**9.7**	**9.0**	**8.1**
10	**10.6**	**11.2**	**10.2**	**9.0**
100	**54.6**	**55.4**	**47.7**	**46.4**

NOTE :- Special Case

In certain situations, computation takes place outside the input array boundaries. When this happens, the results formed from elements outside the boundary are not calculated. The effect of this is that the computational time decreases. An example of such a situation is when the two input arrays are the same size. When this happens, there is only one instance when the solution is formed entirely from elements inside the boundary. For all other instances, the number of computations per sample reduce.

Computation outside the array boundaries can take place in the ConvComplete and Correlate functions only. Taking the extreme case, when both the input arrays are the same size, the timings per sample are given below. Once again, the timings were found by taking the time required to produce n output samples, and dividing by n.

All timings quoted below are given in **microseconds**.

		Routine	
Size Of iw	**No. Of Output Samples**	**ConvComplete**	**Correlate**
5	**9**	**6.5**	**8.4**
10	**19**	**7.1**	**8.2**
100	**199**	**29.0**	**26.1**

2.0 Fast Fourier Transforms (FFT)

Introduction

The Fourier Transform provides a method of performing frequency analysis on a given time domain signal. The FFT of a continuous time domain signal returns the corresponding frequency domain equivalent. The use of a ready designed software routine releases the applications engineer from the burden of complicated mathematics involved with spectrum analysis. As a result of this, the designer is able to concentrate directly on the development of the specific application. Real world applications of the Fourier Transform are common place, and include such examples as noise analysis (e.g. analysing E.M.C. compliance), spectrum estimation as well as numerous telecommunication processes.

DSPLib provides routines which calculate the classical forward and inverse FFTs in Radix 2 form. Each routine is described in detail below, with examples of operation given where necessary.

2.1 FFT Background Theory

The purpose of this section is to reiterate some basic FFT theory such that the reader will gain an appreciation of the function of each of the software routines described in this chapter.

DSPLib provides basic building blocks that allow the calculation of Radix two decimation in time transforms. Radix two implies that the time domain input data is required in blocks that are a power of two. This means that an FFT cannot be performed on a sample by sample process, but instead must be presented in blocks of samples. The choice of size of the block of time domain samples presented to the FFT is determined by a number of factors such as:-

Frequency Content Of Signal

Required Resolution In Frequency Domain

Processor Time Available To Perform FFT

Must Be Power Of Two (Radix Two)

It is necessary for the designer to examine each of the above considerations and reach a favourable compromise in order to implement an efficient FFT for a specific application.

Although in-depth mathematical knowledge of the FFT is not required to use theDSPLib FFT functions, an appreciation of the structure is desirable, thus enabling the designer to understand the required formats for data input and output, as well as understanding the results generated by the FFT.

The forward Fourier transform is defined by the following equation:-

$$y(n) = 2^{-s} \sum_{n=0}^{N} e^{-2j\pi n/N} . x_n$$

Conversely the inverse transform may be written as :-

$$y(n) = 2^{-s} \sum_{n=0}^{N} e^{2j\pi n/N} . x_n$$

The exponential powers shown are known as the twiddle factors and are used to combine two data points within the FFT. To improve on computational time the twiddle factors are computed first and stored in a look-up table. It is desirable to ensure that the look-up table is stored in internal RAM in order to minimise memory access times and hence improve FFT performance. In the case of DSPLib the look-up tables are generated by executing the InitFft routine first.
DSPLib offers two distinct structures of FFT. Both will return bit identical results, and both have near identical execution speeds, however the use of memory differs for the storage of results.

2.1.1 Not In Place FFT

This version of the FFT takes the block of time domain samples from RAM, performs the FFT and returns the result to a new location in RAM. The structure of a not-in-place FFT can be seen in diagram 2.1 below.

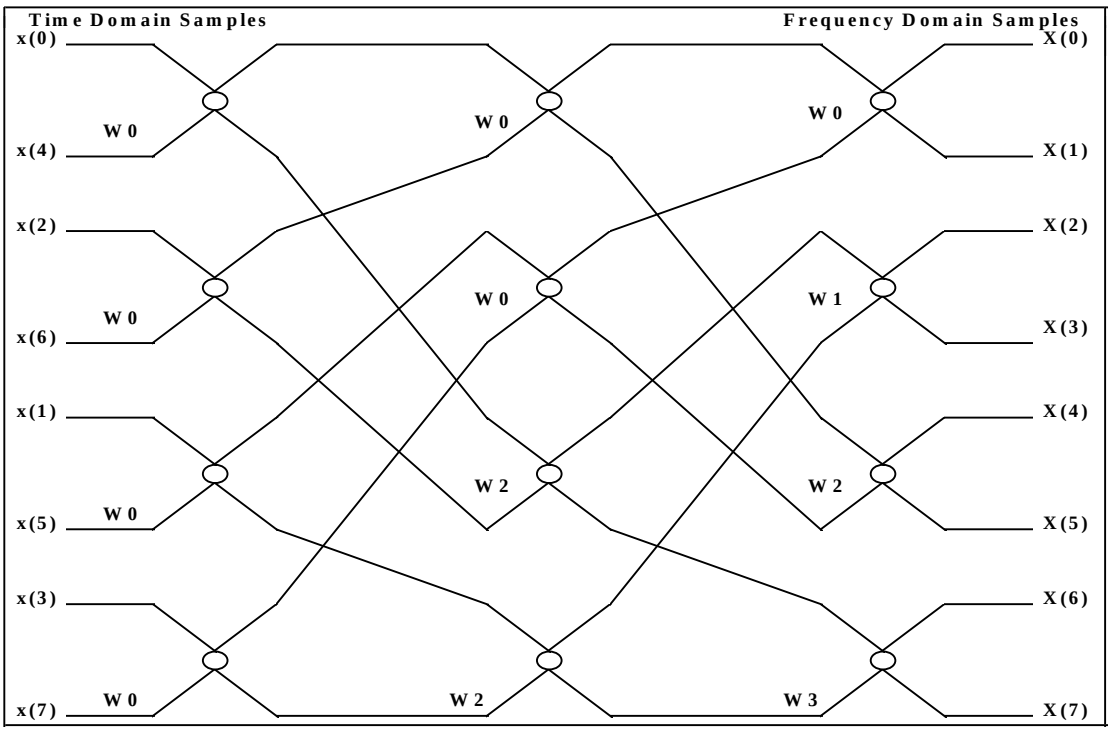


Diagram 2.1 **: Not In-Place FFT Flowgraph**

It should be noted from Diagram 2.1 that the inputs to the FFT are not in order. In order for the FFT to be performed, the data order must be re-arranged by way of bit reversal. Since this operation is performed by the SH, the bit reversal technique is implemented in software (unlike some DSP cores which have bit reversed addressing to enhance execution speed) by the library code, and is of no concern to the user. The output of the FFT is in order, and so no further modifying of the addresses of the samples is necessary.

The weightings applied at each of the butterflies on the Flowgraph are the twiddle factors which were described earlier. The twiddle factors are stored in a look-up table in order to enhance speed.

2.1.2 In-Place FFT

This version takes the block of time domain samples in memory, performs the FFT operation and deposits the results into the same memory locations in RAM. This method has the advantage of using less memory space (50% less), at the expense of execution time. The hardware arrangements for in-place FFTs require careful planning, since for each sample that is processed there are two memory accesses, a read and a write to the same location. To maximise speed, on chip RAM should be used.

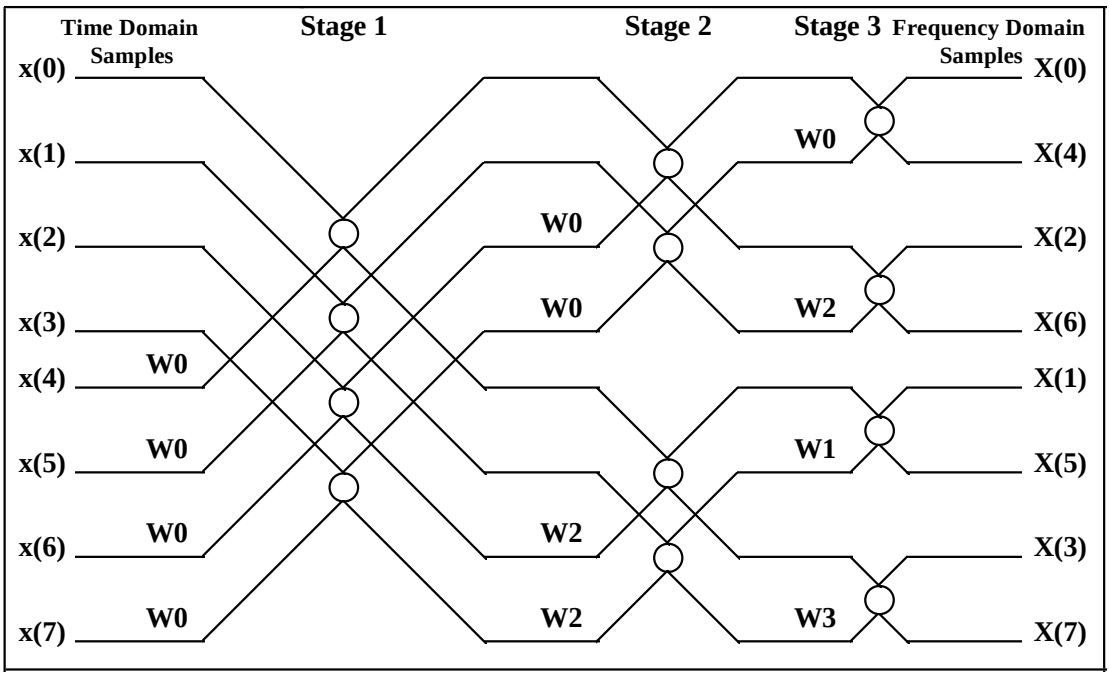


Diagram 2.2 **: In-Place FFT Flowgraph**

As can be seen in the above flow diagram, bit reversal is still necessary with in-place FFTs. The above diagram shows bit reversal is necessary on the output data. In reality, the difference in execution time between the two functions is negligible. Therefore, the only real consideration when deciding on the implementation is the amount of on chip RAM that is likely to be needed. For maximum conservation of memory space, use the in-place implementation.

As with any process where time is important, it is recommended that the whole of DSPLib is executed from on chip RAM.

2.1.3 FFT Scaling

When implementing an FFT, it is important to have an understanding of the signal being presented to it such that overflow does not occur at any stage. There are a number of different methods of preventing overflow, all based around the scaling of the input data. When deciding on the scaling applied, it is important to consider the quantisation introduced against the likelihood of overflow. The related Ensigma text describes in detail the method of scaling employed by DSPLib, and should be carefully considered before implementing a transform. For most applications the scaling EFFTALLSCALE (0xFFFFFFFF) will be suitable. This level of scaling simply performs halfing on every stage, with a guarentee that no overflow will occur.

To summarise FFT scaling, here are the three levels offered by DSPLib.

EFFTALLSCALE :- Halving performed on every stage, guaranteeing no overflow if the input samples are all less than 2^{30} .

EFFTMIDSCALE :- Halving performed on alternate stages, hence giving an output with the same power as the input.

EFFTNOSCALE :- No scaling performed. Signal properties must be well understood when using this.

2.1.4 Input / Output Array Format

DSPLib offers both real and complex FFT computation, and so the format and ordering of data arrays presented to the transform must be understood. Firstly, we will look at purely real arrays, as used in all the real forward and inverse transforms. The order of the array is as follows:-

x[2n] = Re{C(n)}	i.e. all elements are real on input
x[2n+1] = 0	all imaginary components are zero (real signal)

For a real transform, the input array contains real time domain samples and so contains no phase information. The output from the transform is the frequency domain representation of the signal. The output of the FFT is in complex array format (see below), however, in the case of a real signal all the complex samples are zero. Each real sample represents the signals frequency content at a certain point. The highest sample, i.e. the last sample represents Fs/2. I.e. the FFT only returns frequency data for half the sampling frequency.

The complex transforms require data to be presented in complex array format, and similarly the results are returned in the same format. The ordering of a complex array is as follows:-

x[2n] = Re{c(n)}	Real elements
x[2n+1] = Im{c(n)}	Imaginary elements

I.e. the input and output elements are ordered :- {real, imaginary, real, imaginary .....}

2.2 Routine Descriptions

2.2.1 FftReal

This function performs a purely real forward fast fourier transform. For a full description of the definition of the function, the Ensigma documentation should be consulted.

The routine is defined as follows :-

```
int FftReal( output, input, size, scale)
```

Where,

short	output[]	positive frequency output data
short	input[]	real input samples
long	size	size of FFT
long	scale	scaling applied to FFT

Using FftReal

The use of FftReal, as with any other function in DSPLib is relatively straight forward. The user must define the scaling and size of the function, as well as present input data in a suitable format.

This example, and others in this section use the LogMagnitude function from DSPLib to present the output of the FFT in log magnitude format. Although the use of this function is included in the text, further details of this function are included in the Ensigma documentation.

Example

The following example outlines a simple application of the FftReal function, intended as a starting point for the reader when developing more specific applications. In this example, the signal applied to the FFT contains two frequency components with different amplitudes. The FFT is used to determine the frequency and amplitude of the components of the signal.

Data presented to the FFT must be 16 bits or less(including the sign bit), and in blocks that are a power of two (radix two operation).

In our case, the input consists of 128 samples. The output will consist of 64 samples interlaced with zeros, confirming that the input is real, since all complex samples are null.

It is necessary for the user to determine the size of the FFT, and also the scaling applied during calculation. Since there are 128 input samples a 128 point FFT is required for a single block computation. Referring to the FFT structure in diagram 2.1, the required size of FFT becomes apparent.

In the diagram, an eight point FFT is shown, and is capable of processing data in blocks of eight. Therefore, in our case, the 128 input samples can be processed in one block using a 128 point FFT.

The C code implementation is shown below, and shows how DSPLib is combined in a standard ANSI C program. The array INPUT has been created from an earlier source, either an ADC or generated by the SH in a routine. The samples in INPUT are transformed and written to the array OUTPUT.

```
#include <stdio.h>
#include <math.h>          /* Typical Libraries To Be Included */
#include <ensigdsp.h>

#define NSAM  128          /* Number Of Input Samples*/
#define NSAMS  NSAM/2      /* Number Of Output Samples From LogMagnitude */

void main(void)
{
       short OUTPUT[NSAM], LOGOUT[NSAMS];

       float  fscale;

       fscale =      1;

/* Enter Code Here To Generate INPUT Data */

/* Generate Twiddle Factor Look-Up Tables */

       if(InitFft (NSAM) != EDSP_OK)
       printf("Problem With Generating Look-Up Table");

/* Perform FFT Operation */

       if(FftReal (OUTPUT, INPUT, NSAM, EFFTALLSCALE) != EDSP_OK)
       printf("Problem With Performing FFT Operation");

/* Log Magnitude Results Returned By FFT */

       if(LogMagnitude( LOGOUT, OUTPUT, NSAM, fscale) != EDSP_OK)
       printf("Problem With LogMagnitude Routine");
}
```

The above example uses a total of three functions from DSPLib, and is centred around the FftReal function described above.

InitFft calculates the twiddle factors and stores them in a look-up table which is used by FftReal during its execution. More details of InitFft can be found in the Ensigma documentation.

LogMagnitude takes the complex output data from FftReal, and returns a purely real logarithmic result. The scale in LogMagnitude may be set to suit the application, and in this case is set to one. LogMagnitude can be found in the Ensigma text. Diagram 2.3 below shows a plot of the 128 samples contained in the array INPUT. The FFT processed these samples and returned the results to OUTPUT. Following the completion of FftReal, LogMagnitude calculated the log format of the results, and this can be seen in diagram 2.4 below.

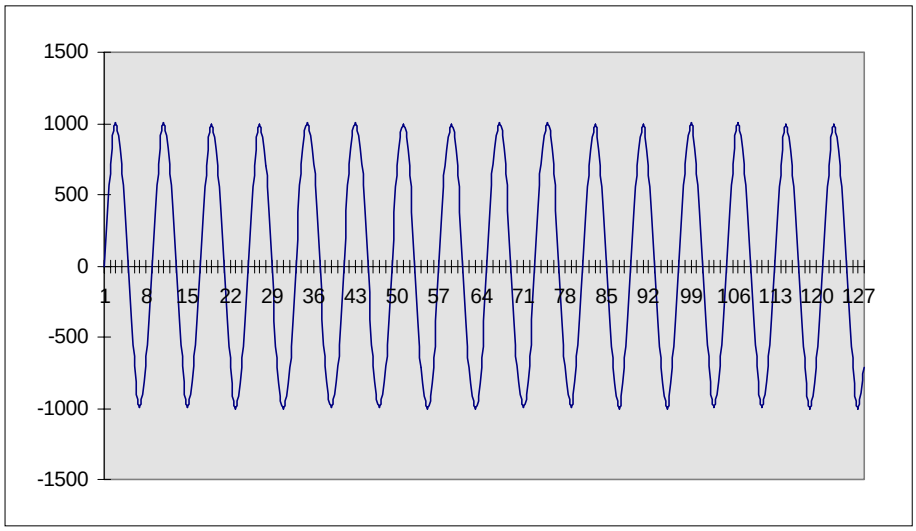


***Diagram 2.3* :- 128 Time Domain Input Samples**

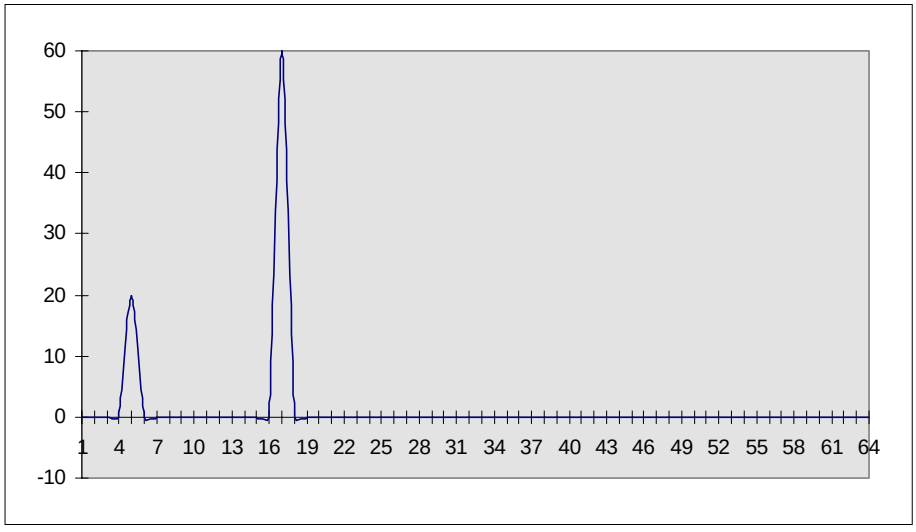


***Diagram 2.4* :- Result Of FFT (64 Frequency Domain Samples)**

The result of the FFT is shown in diagram 2.4 above. There are only two samples which are not zero, and these represent the two frequency components of the signal shown in diagram 2.3.

It should be noted that the result of LogMagnitude is half the length (64 samples) of the original FFT result (128 samples). This is due to the fact that the real and imaginary samples are squared, added together and then logged.

Analysis of the above plot in diagram 2.4 will give us detailed information on the spectrum of the signal in diagram 2.3. The plot is analysed as follows :-

The two peaks appear at samples 4 and 16. Since sample 64 is known to correspond to Fs/2 (half the sampling frequency), other samples may be calculated as follows to form a table :-

Frequency		Sample
Fs/2	64/1	64
Fs/4	64/2	32
Fs/8	64/4	16
Fs/16	64/16	4

Hence, from the above table we know that our signal contains two components, one at Fs/4 and the other at Fs/16, which correspond to samples 16 and 4. Next it is necessary to determine the amplitude of those components. Fs/16 and Fs/8 have amplitudes of 20dB and 40dB respectively. Since the log of a signal is 20 logVpk, where Vpk is the peak amplitude of the signal, the original signal amplitudes may be found as follows:-

Amplitude of Fs/16 = anti-log (20/20) = 10

Also

Amplitude of Fs/8 = anti-log (60/20) = 1000

So the result of our FFT process is the complete frequency content of the original signal up to and including Fs/2. Clearly higher in the spectrum is of no concern to us due to the aliasing effects of digitisation.

The method for determining the amplitude of the components shown above is only applicable if EFFTALLSCALE is used, and if the scale in LogMagnetude is set to one. For most applications, this arrangement will be satisfactory and provide the simplest implementation.

This example has omitted the use of windows when taking FFTs. Windows are used to compensate for the ‘end effect’ encountered when taking the transform of a finite data set. The sudden end of the data generates many harmonics which strictly speaking are not present in the signal. The result of this is that the FFT represents frequency components which are of no interest to the user.

In order to counter effect this, the data is applied through a window, which has the effect of 'fading' out the start and finish, thus reducing harmonics caused by straight edges. There are a number of different window shapes available, and the omittance of a window is referred to as rectangular windowing, and applies to the above example.

DSPLib provides four windows, being Blackman, Hamming, Hanning and Triangle. These functions and their use are described in section 1.3.2 .

2.2.2 FftComplex

This routine allows the user to present both real and complex data to the FFT for processing. The result of the FFT is in the same format as that in FftReal, only the complex values will be non-zero(assuming the complex input data is non-zero). The use of the routine is very similar to FftReal, and follows on directly from its description. It is recommended that the reader looks at the FftReal text in the previous section before continuing here.

The complex FFT is essentially two separate FFTs, one for the complex data and one for the real. These FFTs are completely separate, and do not interact in anyway. FftComplex is not-in-place, and hence returns its solution to a different memory location to that of the input data. Diagram 2.1 shows the structure of a not-in-place FFT.

The related Ensigma documentation gives detailed information regarding the definition of FftComplex, however there now follows a brief summary of the function for your reference.

The routine is defined as follows :-

```
int FftComplex( output, input, size, scale)
```

Where,

short	output[]	Complex Output Data
short	input[]	Complex Input Data
long	size	Size Of FFT
long	scale	Scaling Applied To FFT

Use Of FftComplex

FftComplex is essentially the same to use as FftReal, and so the previous example is applicable here. The difference between the two functions is that FftComplex allows complex data to be processed. Therefore, the input and output arrays are both in complex array format, and so the user should ensure familiarity with this.

When specifying the size of the FFT, it is important to remember that half the inputs are complex. Therefore, if the user has 128 real samples and 128 complex samples, and it is necessary to process the samples in one block, a 256 point FFT will be required.

A block diagram representation of the above instance is shown in diagram 2.5. The diagram shows how two N point FFTs are required to process the real and complex data samples. Typically the complex samples are representations of the signals phase characteristics, but as explained below, this doesn't have to be the case.

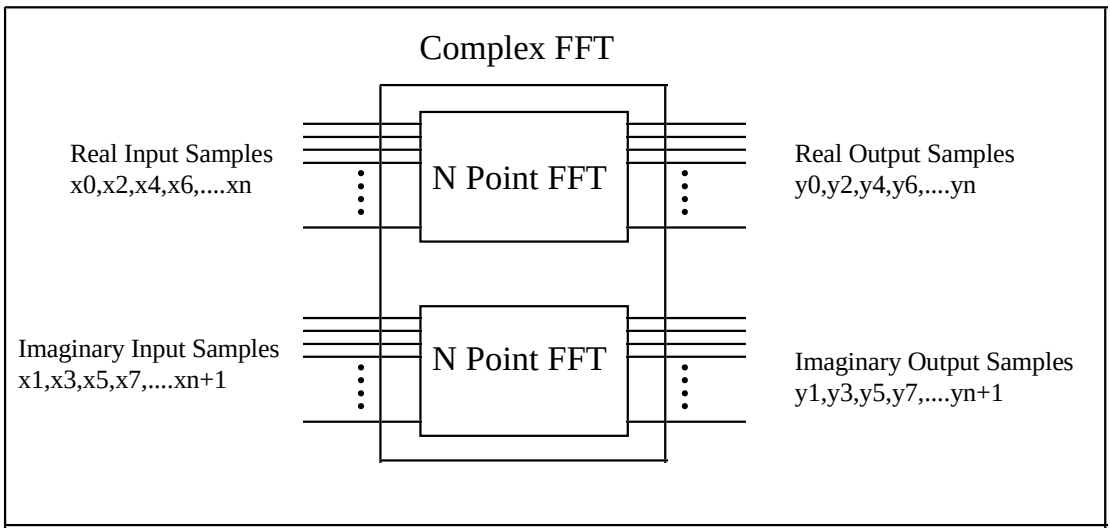


***Diagram 2.5* :- Representation Of FftComplex Showing Two Separate FFTs**

As the above diagram shows, the FftComplex routine is actually two separate interlaced FFTs. This fact can be exploited by way of performing two real FFTs simultaneously. Real data can be applied to both the real and complex inputs, with the corresponding results available in the odd and even samples at the output. In certain applications, this method can offer time savings over performing two separate FftReal routines.

2.2.3 IfftComplex

This function offers the exact opposite to FftComplex. IfftComplex calculates the inverse FFT and returns a not-in-place solution. The user presents the routine with a set of frequency domain samples in complex array format. The highest sample in the array must correspond to the frequency Fs/2s' imaginary component. If the signal is real, all the imaginary values (odd addresses) will be zero. The transform will return the equivalent time domain signal for the given frequency data. The time domain data will be in complex array format, and will require separating into real and imaginary if necessary. When applying data to IfftComplex, it is important that the samples are not in logmagnitude form, since this will generate an invalid return on the output.

As with all the FFT routines in DSPLib, the user must specify scale and size. Full details of the routines description are in the related Ensigma documentation.

The routine definition is summarised below for your reference:-

```
int  IfftComplex( output, input, size, scale)
```

Where,

short	output[]	complex output data
short	input[]	complex input data
long	size	size of inverse FFT
long	scale	scaling applied during IFFT execution

Use Of IfftComplex

The routine is very simple to use, and as described above is the exact opposite to the FftComplex routine described. One method of establishing correct operation is to perform both the forward and inverse transforms on a data set, then comparing the results to the original data input. If the results are different, there is a problem with one or both of the functions, such as overflow, excessive quantisation or simply scaling. An example of such a program is given below.

```
#include <stdio.h>
#include <math.h>          /* Typical Libraries To Be Included */
#include <ensigdsp.h>

#define NSAM  128          /* Number Of Input Samples*/
#define NSAMS  NSAM/2      /* Number Of Output Samples From LogMagnitude */

void main(void)

{
       short OUTPUT[NSAM], OUTPUTA[NSAM], LOGOUT[NSAMS];
       float  fscale;
       int    I;

       fscale =      1;

/* Enter Code Here To Generate INPUT Data */

       if(InitFft (NSAM) != EDSP_OK)
       printf("Problem With Generating Look-Up Table");


       if(FftComplex (OUTPUT, INPUT, NSAM, EFFTALLSCALE) != EDSP_OK)
       printf("Problem With Performing FFT Operation");

       if(IfftComplex (INPUTA, OUTPUT, NSAM, EFFTALLSCALE) != EDSP_OK)
       printf("Problem With Performing IFFT Operation");
```

```
for(i=0; I < NSAM; I++)
       {
              if((INPUT[i] - INPUTA[i]) != 0)
              printf("Problem With Format Of Results");
       }
 }
```

The above C code shows how both FftComplex and IfftComplex are used as functions in a program. The code performs a complex forward not-in-place FFT on the data stored in INPUT. The resultant frequency domain data then has an inverse FFT applied and the result stored in INPUTA. If both routines are operating correctly, the values stored in INPUTA should be identical to INPUT. This is tested by subtracting the two and comparing to zero. The result is compared to zero, and if different, an error message is displayed to indicate to the user that the functions are not operating correctly. The reader should note the use of the same scaling applied to both forward and inverse transforms. As discussed above, a number of factors can lead to the data being different, and most can be 'ironed' out by the user during development.

2.2.4 IfftReal

IfftReal presents a function which calculates the inverse not-in-place FFT. This function follows on from the above description of IfftComplex, with the exception of the format of the returned data. The user presents the routine with complex frequency domain samples, with the highest sample corresponding to Fs/2. The inverse transform then returns a purley real time domain representation of the input data. Further details regarding the definition of the function may be found in the related Ensigma documentation. For your reference the definition of the function is as follows:-

```
int IfftReal( output, input, size, scale)
```

Where,

short	output[]	real output data
short	input[]	complex input data
long	size	inverse FFT size
long	scale	scaling applied

Using IfftReal

When using IfftReal, the input must be in complex array format. However, the signal returned from the function must be real, and therefore the input must represent a real signal. Therefore, the complex input array must represent a positive frequency, and so all complex array elements must be zero. Before calling this routine the twiddle factors should be generated by calling InitFft.

2.2.5 FftInComplex

This routine performs the same function as FftComplex which was described earlier in this section. The user should choose this routine when memory space is of the essence. FftInComplex calculates the complex forward FFT in-place. The flowgraph for an in-place FFT can be seen in diagram 2.2. The function is described in detail in the related Ensigma documentation. For your reference, the definition of the function is as follows:-

```
int FftInComplex( data, size, scale)
```

Where,

short	data[]	complex input and output data
long	size	FFT size
long	scale	FFT scaling applied

Using FftInComplex

As described this function is identical in operation to FftComplex except for the use of memory. The function takes the input data from memory (stack) performs the calculations, then returns the results to the SAME memory location. This enables the user to maximise the use of memory space, but at the expense of overwriting the input data.

2.2.6 FftInReal

Not surprisingly this follows the same format as FftReal, except that the FFT is performed in-place. The user should refer to the section on FftReal for details of operation when using this routine, and also reference to the related Ensigma documentation would be useful. For your reference the definition of FftReal is as follows:-

```
int FftReal( data, size, scale)
```

Where,

short	data[]	real data input, complex data output
long	size	FFT size
long	FFT scaling applied	

Using FftInReal

The input to the FFT is real data only, where as the output is in complex array format with the complex array elements equal to zero. To allow the FFT to be calculated in-place, the Fs/2 frequency output is stored in the second element of the array.

This slightly complicates the output, and should be remembered when using. For more details consult the Ensigma documentation.

2.2.7 IfftInComplex

This routine represents an in-place implementation of the complex inverse FFT. Its operation is essentially the same as IfftComplex, only with an in-place computation.
As with all of the in-place routines, IfftInComplex should be chosen when memory usage is critical. The routine is described in the related Ensigma documentation on. However, for your reference, here is the definition of IfftInComplex:-

```
int  IfftInComplex( data, size, scale)
```

Where,

short	data[]	complex input/output data
long	size	FFT size
long	scale	FFT scaling applied

Using IfftInComplex

Operation is identical to the IfftComplex routine described only the results of the inverse FFT are written to the same memory location as the input. Hence all input data is overwritten, but memory usage is optimised. As with any of the DSPLib routines, it is desirable to locate data in on chip memory.

2.2.8 IfftInReal

This final routine in the FFT section implements a real inverse in-place FFT. Operation is identical to IfftReal described earlier in this section, with the exception of the in-place computation returning the results to the same memory location as the inputs. Full details of IfftInReal can be found in the related Ensigma documentation. For your reference the function is defined as follows:-

```
int  IfftInReal( data, size, scale)
```

Where,

short	data[]	Complex positive frequency data input, real data output
long	size	Inverse FFT size
long	scale	FFT scale applied

Using IfftInReal

As described, this method offers in-place inverse real FFT calculation. For details on the use of this function refer to the section detailing IfftReal. Real output data is in-place and so overwrites the original input data. Use of this function should be limited to applications where memory usage must be optimised.

When applying complex data to IfftInReal, the Fs/2 value should be placed in the second element of the array where the imaginary component of zero would normally be stored. Although this might complicate matters some what, it enables the function to operate in-place.

Summary

This section is intended to give the reader an insight into the operation of each of the FFT functions. While keeping the descriptions as general as possible to suit most applications, the descriptions should enable the user to choose between functions when designing a particular application, as well as pointing out certain design considerations.

The detail and theory in this document is not intended to form a reference for FFT design, but should instead provide a starting point from where further DSP research may begin.

NOTE

It is important to remember to initiate the twiddle factor tables before using any of the FFT routines. These tables are initialised by using the function InitFft in the DSPLib. Details regarding both this function and the logmagnitude function can be found in the related Ensigma documentation.

3.0 Window Functions

Introduction

This section of DSPLib provides four window generators which may be used for a number of applications, but essentially are used for counter-effecting the 'end-effect' experienced when taking the FFT of a finite data length.

Each of the four routines provide efficient methods of calculating the window, and deposit the results into a user specified memory location. The particular window is then recalled when required and applied accordingly to the desired data. The four windows are described below, along with plots of their associated shapes.

Since a window is a continuous signal between limits, it is necessary to divide the window into equally spaced samples. The number of samples used to represent the window is specified by the user when calling the function. Clearly the more samples used, the greater the accuracy of representation.

3.1 Function Descriptions

The window generators in DSPLib are defined as follows :-

```
int GenBlackman( output, win_size)

int GenHamming(output, win_size)

int GenHanning(output, win_size)

int GenTriangle(output, win_size)
```

Where,

short	output[]	array to containing window coefficients
long	win_size	size of window N and length of output[]

Each function will return EDSP_BAD_ARG if the user specifies a window size that is smaller than one. If the function executes correctly, the flag EDSP_OK is returned. These flags are intended for debug use, and will help the user identify problems in code development.

The functions generate a window to a size specified by the user. The shape of the generated window can be seen in respective diagrams below. Each diagram shows the window when size was selected as 128. Therefore, 128 samples (or window coefficients) are used to represent the windows.

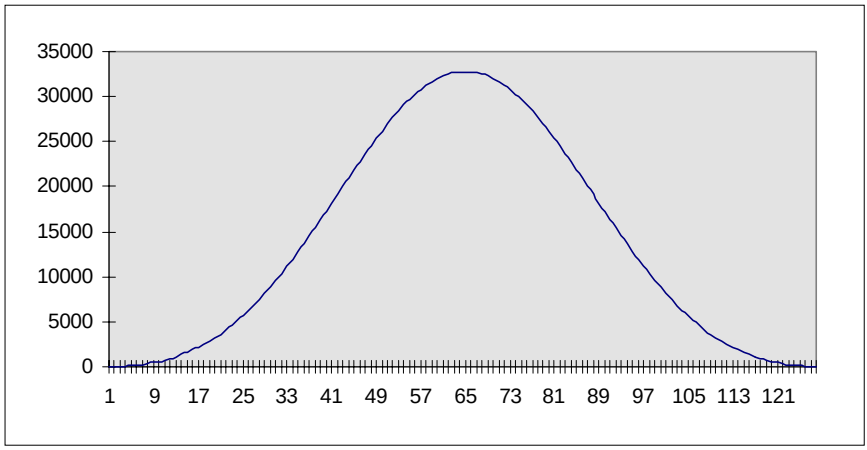


Diagram 3.1 :- **Blackman Window**

The Blackman window for 128 samples is shown above in Diagram 3.1. It can be seen that the side lobes actually extend right down to zero, as well as having a wide centre lobe. Mathematically, the Blackman window is defined as follows:-

$$w(n) = (2^{15} - 1)\left[0.42 - 0.5\cos\left(\frac{2\pi n}{N}\right) + 0.08\cos\left(\frac{4\pi n}{N}\right)\right] \quad 0 \le n < N$$

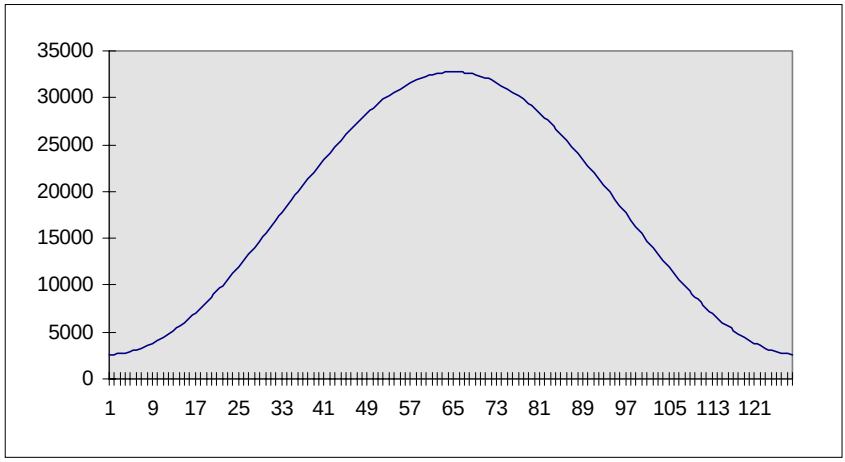


Diagram 3.2 :- **Hamming Window**

The Hamming window for 128 samples is shown above in Diagram 3.2. It can be seen that the side lobes do not extend right down to zero, but it does have a very wide centre lobe. Mathematically, the Hamming window is defined as follows:-

$$w(n) = (2^{15} - 1)\left[0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

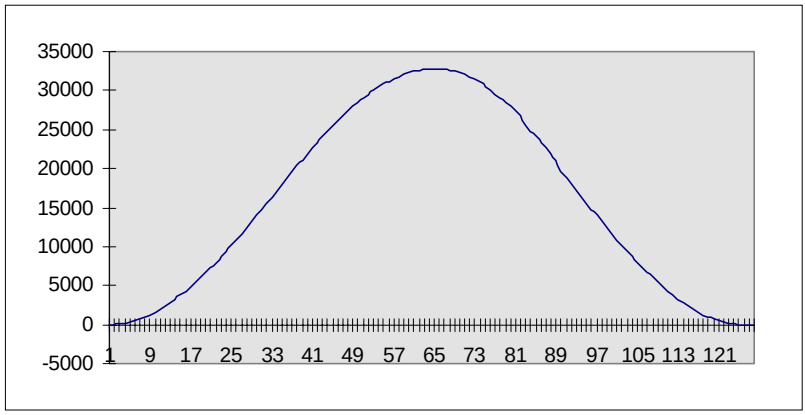


Diagram 3.3 :- **Hanning Window**

The Hanning window for 128 samples is shown above in Diagram 3.3. It can be seen that the side lobes extend right down to zero, and it has a very wide centre lobe. Mathematically, the Hanning window is defined as follows:-

$$w(n) = \frac{(2^{15}-1)}{2}\left[1-\cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

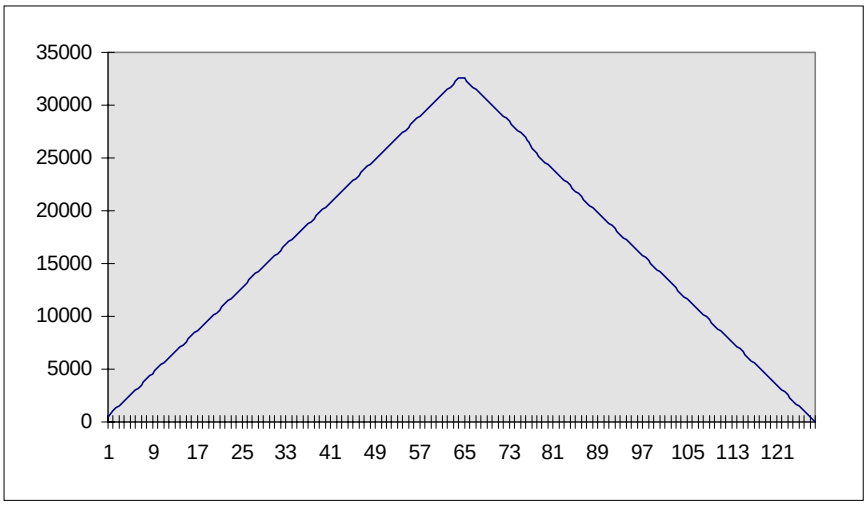


Diagram 3.4 :- **Triangular Window**

The Triangular window for 128 samples is shown above in Diagram 3.4. It can be seen that the side lobes extend right down to zero, and it has a very narrow centre lobe. Mathematically, the Triangular window is defined as follows:-

$$w(n) = (2^{15}-1)\left[1-\left|\frac{2n-N+1}{N+1}\right|\right] \quad 0 \le n < N$$

3.2 Using Window Functions

The use of window functions in DSPLib is very simple, and is demonstrated in the section of C code given below. It should be remembered that the execution of each of the window functions is relatively slow, and increases with the size of the window. In applications where time is an issue, such as real time processing, the window should be pre-calculated and stored in a look-up table. It is preferable to locate this table in on chip RAM inorder to minimise access times. The example below shows the Blackman window being generated. The values generated are stored on the stack in an array defined by the user. Because, ideally the stack is located in on chip RAM, access to the generated values should be fast. The location of the stack is defined by the user in the linkage editor.

```
#include <stdio.h>
#include <math.h>                /* Typical libraries */
#include <ensigdsp.h>            /* MUST be included */
#define      WINLEN 128    /* Define length of window */

void main(void)
{
      short output[WINLEN];      /* Define array to contain window */
      if(GenBlackman( output, WINLEN) != EDSP_OK)    /* Generate window */
      printf(" Problem Generating Blackman Window"); /* Report if error has
                                                          occurred */
}
```

When the above code is executed, the window values are stored in an array called output[] of length 128. Although there is not an upper limit to the size of window generated, care should be taken to ensure that the stack is sufficintly large to accomodate all the generated values, as well as other values being used in the programs execution.

The execution speed of the routines is of less importance than other routines described in DSPLib. The window function is typically used to generate values which are stored in a look up table for use in a real time loop. Therefore an optimised version is not available in DSPLib. The running time of the routine can be improved by the removal of the argument checking section of the code. This is only recommended once the specified arguments are known to be correct and generate the desired output.

3.3 FFT Application Of Window Functions

Before reading this section, the reader should ensure familiarity with the FFT functions offered by DSPLib. Section two of this document will provide the necessary background material relating to this example.

As described earlier, window functions have a number of different uses ranging from filter design to FFT pre-processing. It is the latter that is discussed here.

The FFT of a finite data set generates harmonics which are not related to the actual signal, and so leads to false solutions. This effect is caused by the sudden end of the input data, resulting in straight edges at either end. As familiarity with Fourier series representation would confirm, a great number of harmonics (infinite for exactly vertical) are required to represent a straight edge. It is these harmonics which are quite correctly represented by the FFT.

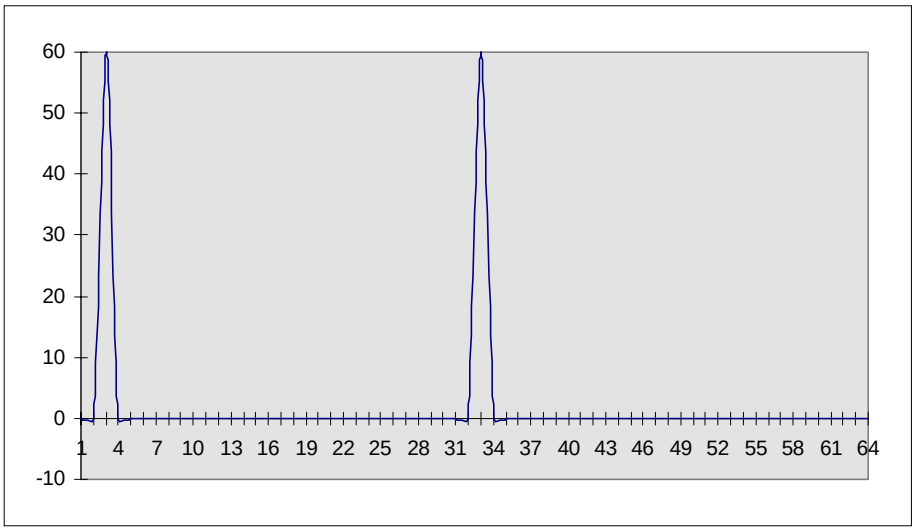


Diagram 3.5 :- **Desired FFT Result**

Diagram 3.5 shows the desired result from the application of an FFT to a particular signal. The signal in question has two frequency components, one at Fs/4 and the other at Fs/32. These two frequencies correspond to samples 32 and 4 respectively. With the introduction of a finite data set, with a sudden end to the samples corresponding to a straight edge, the resulting FFT is as shown in Diagram 3.6 below. Clearly a great number of harmonics have been generated which spread right across the spectrum of interest. The two frequency components are visible, but are completely encompassed by harmonics.

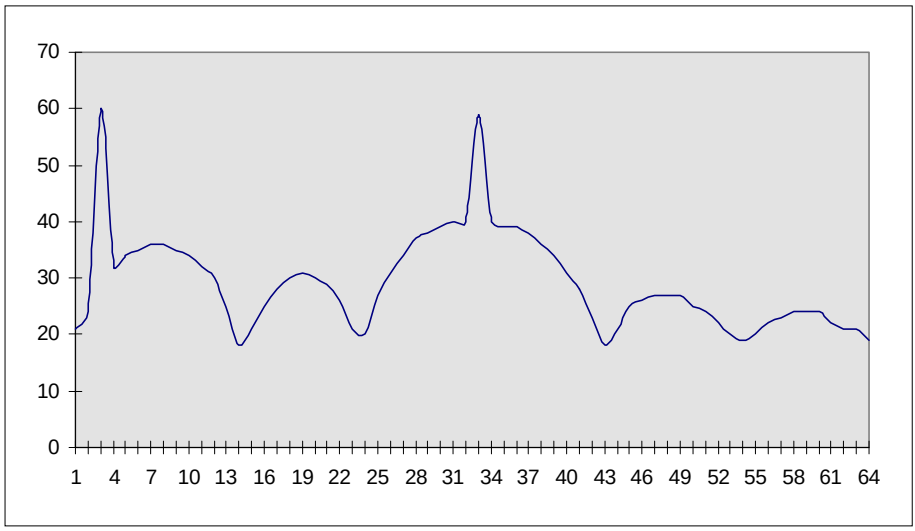


Diagram 3.6 :- **Result Of FFT On Finite Data Length**

In order to counter effect the harmonics, the finite time domain signal is multiplied by a window of the same length as the data set. The result of this operation is shown below. The effects of different windows are shown, with varying degrees of success. The C code implementation of this example is given at the end of this section.

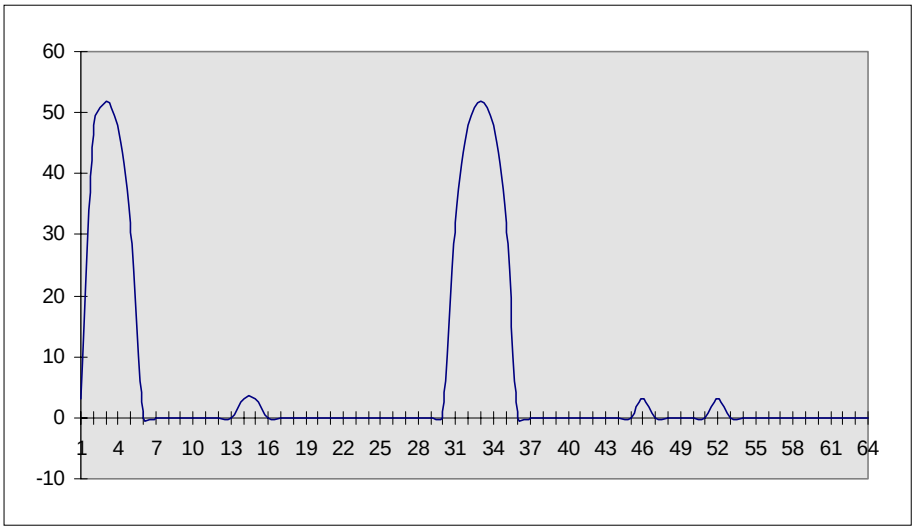


Diagram 3.7 :- **Result Of Blackman Windowing**

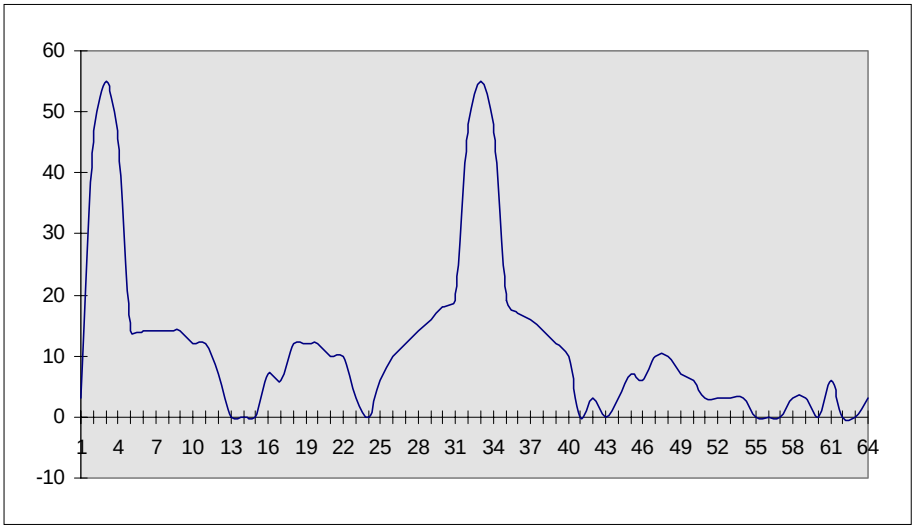


Diagram 3.8 :- **Result Of Hamming Windowing**

(Continued Over)

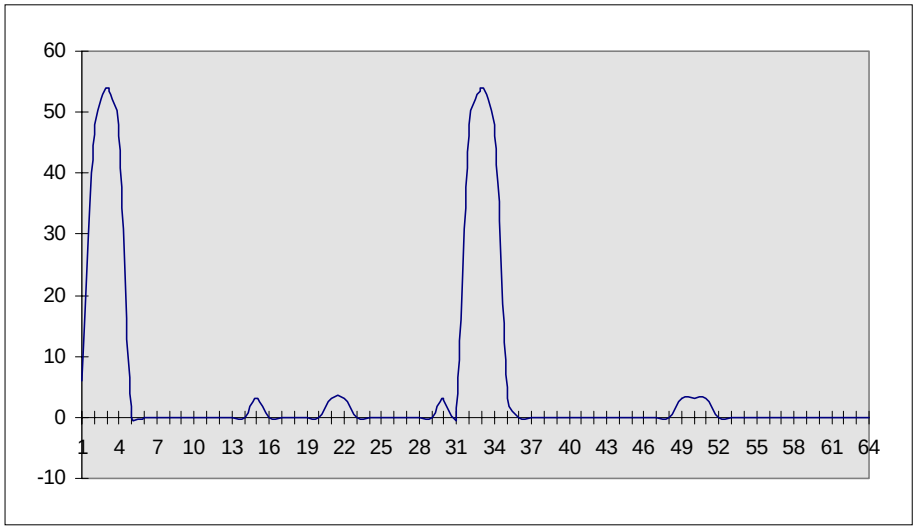


Diagram 3.9 :- **Result Of Hanning Windowing**

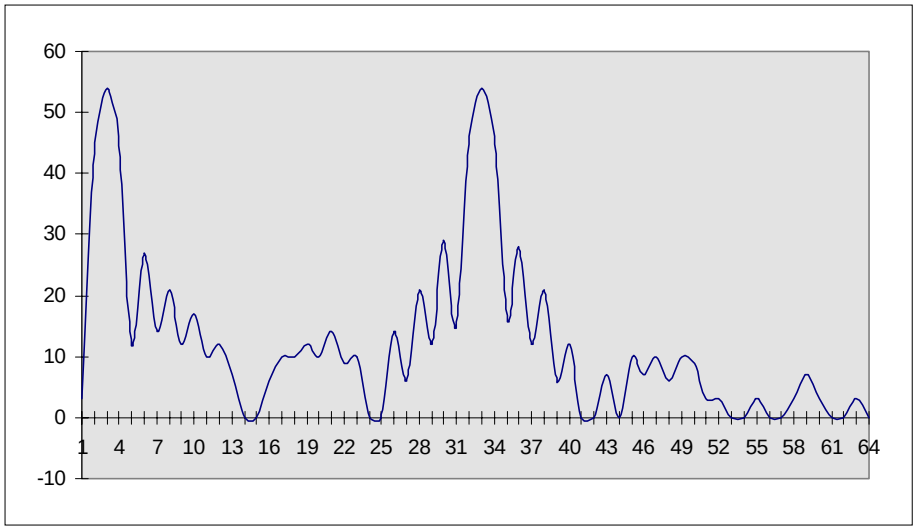


Diagram 3.10 :- **Result Of Triangular Windowing**

As can be seen in Diagram 3.9, the Hanning window generates the best result. The window has removed most of the harmonics, leaving a clear frequency representation of the finite data. In most applications, this type of 'trial and error' approach will suffice.

The C code implementation is as follows :-

```
/**
/** DESCRIPTION: This file Generates a sine wave, applies a Hamming window
/**              applies an FFT and then returns the solution in log format

#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#include "testhead.h"
```

```
#define NSAM     128              /* number of samples */
#define NSAMS    NSAM/2
#define TWOPI     6.283185307

/***************** Exercise To Take The FFT Of Data ******************/

void main(void)
{

/** Declare Local Variables **/

        short       sinea1[NSAM], window[NSAM], signal[NSAM], sinea[NSAM],
                    sineb[NSAM],fftouta[NSAM],outputa[NSAMS],outputb[NSAMS];

        int     n, k, len, res_shift;
        float   scale, scalefft;

        scale           =       1;
        res_shift       =       15;
        len             =       NSAM;
        scalefft        =       EFFTALLSCALE

/*** Initiate Twiddle Values ***/

        if(InitFft (MAXSIZE) != EDSP_OK)
        printf("Problem With Initialisation Of FFT");

/*** Generate Sine Wave With Sudden End***/

        if(sine == 0)
        {
             printf("Generating Sine Wave");
             k = len / 64;
             for(n = 6; n < 122; n++)
             {
             sinea1[n] = floor(1000 * sin(TWOPI * k * n / len) + 0.5);
             }

             k = len / 4;
             for(n = 6; n < 122; n++)
             {
             sineb[n] = floor(1000 * sin(TWOPI * k * n / len) + 0.5);
             }

        printf("Done");
        }

/*** Add Sine waves ***/

        for(n = 6; n < 122; n++)
        {
             sinea[n] = sinea1[n] + sineb[n];
        }

        for(n = 0; n < 6; n++)
        {
             sinea[n] = 0;
        }

        for(n = 122; n < 128; n++)
        {
             sinea[n] = 0;
        }

/*** Generate Windows ***/

        if(GenTriangle( window, NSAM) != EDSP_OK)
        printf(" Problem With Window Generator ");

/** Multiply Window With Signal **/

        if(VectorMult( signal, window, sinea, NSAM, res_shift) != EDSP_OK)
        printf("Problem With Vector Mult");

/*** Take FFTs Of Sine Waves ***/

       if(FftReal( fftouta, signal, NSAM, scalefft) != EDSP_OK)
       printf("Problem With FFT");
```

```
/** Covert FFT result to log magnitude format **/

        if(LogMagnitude( outputa, fftouta, NSAMS, scale) != EDSP_OK)
        printf("problem With LogMagnitude");

/** Release FFT RAM **/

        FreeFft();

}
```

The above C code listing brings together a number of different DSPLib functions, and provides a good example of DSP processing on the SH.

The reader should be familiar with the FFT content of the program, having read section 2.0 of this document.

Following the flow of the program, the first task was to generate the signal. Two sine waves, one of Fs/4 and one of Fs/64 are generated and then added together. The floor function used in the generation of the sine waves is contained in the math.h library. This function takes the solution of the equation to its' right, and rounds it to the nearest integer that is not greater than it. Hence the function rounds down.

Next, six zeros are deliberately inserted into the start and finish of the data set inorder to create a sudden end to the signal, and hence generate many harmonics, as shown in Diagram 3.6. Following that, a window is generated using one of the functions discussed in this section. The particular window function can simply be changed to test for best results, as was the case in this example.

A window is applied to the signal on a sample by sample bases. I.e. sample zero of the signal is multiplied by sample zero of the window, and so on until each element has been processed. This task is performed by the function VectorMult which is contained in DSPLib under the miscellaneous section. For more details regarding this function, the reader should consult section 6.0 of this document, as well as the related Ensigma documentation.

Once the signal has been windowed, the resultant waveform is applied to the FFT function. This function is described in detail in section two of this document, and is of no concern to us here.

The solution from the FFT is converted to log format, and stored in an array for future use. The FFT chosen was ‘not-in-place’, however an ‘in-place’ implementation would also be suitable and save memory usage.

The results shown in Diagrams 3.5 through 3.10 were generated by the given C code running on an SH-1 processor. Clearly for a real application, the signal would not be generated on chip, but would instead be read in from an external source such as an ADC. Notes on processing real time signals are included in section 7.0 of this document.

4.0 Filters

Introduction

This section of DSPLib provides a number of functions which allow the user to perform digital filtering on discrete time samples. Essentially the library offers three different filter networks, even though it contains eight filter functions. The three filters are as follows :-

Finite Impulse Response (FIR)

Infinite Impulse Response (IIR)

Least Mean Squares Adaptive Algorithm (LMS)

DSPLib allows the user to implement the above functions on both blocks of data, or alternatively on a single sample process which is intended for real time applications. The accuracy of the LMS algorithm can be further increased by the use of DIIR which is a double precision IIR implementation allowing the user to specify 32-bit coefficients.

This document is intended to provide the reader with an introduction to the implementation of digital filters on the SH processor. This chapter details some background theory on digital filters, but is not intended to form a comprehensive reference text. A certain level of DSP knowledge is assumed, and if the reader requires further information, a relevant DSP text should be consulted.

4.1 Filter Background Theory

In order to ensure a full appreciation of the function descriptions later in this section, some basic filter theory is outlined here. The theory describes the difference between the filter types, as well as outlining the filter structures used in DSPLib. It is recommended that the reader follows this text before moving onto the particular function description of interest.

The digital filter lies at the heart of DSP processing. Digital filters have found applications in many different areas, and their everyday use is common place. Digital filters can be found in equipment ranging from compact disc players to mobile telephones.

Digital filters are implemented with many different structures, and numerous hardware platforms. Hardware filters are available which allow the coefficients to be loaded in from ROM, and clearly dedicated DSP cores allow efficient implementation. DSPLib allows the user to take advantage of all the features you would expect when using a RISC core, as well as utilising the SH MAC (Multiply and ACcumulate) register, DMA (Direct Memory Access) controller and on chip ADC capabilities which together make real time micrcontroller DSP a reality.

4.1.1 Finite Impulse Response Filters (FIR)

DSPLib offers two implementations of FIR filters. The basic operation of these functions is described later in the routine descriptions.

The FIR filter has a number of distinct advantages, in particular :-

i) FIR filters implemented in direct form (functions of past and present samples), they are always stable.

ii) The phase response of an FIR filter is exactly linear. A filter that doesn't have linear phase will cause a phase distortion on the signal passed through it. Such distortion is unacceptable in certain applications such as audio, video and various types of data transmission.

iii) As indicated in the following text, the implementation of an FIR filter is very simple. The SH series of microcontrollers have an architecture that is particularly suited to the implementation of such filters. The heart of the FIR filter is the MAC operation, which the SH series can perform in three CPU clock cycles.

The FIR filter is described in the time domain as follows :-

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

Taking the Z transform of the above expression enables us to represent the FIR filter in the digital domain as follows :-

$$H(Z) = \sum_{k=0}^{N-1} h(k)Z^{-k}$$

As can be clearly seen, the entire FIR routine consists of multiplying past and present samples with the filter coefficients h(k), and summing the solutions. Hence the implementation of the filter is entirely MAC operations and delays.

DSPLib implements the FIR filter by way of a transversal structure. This structure is shown below in diagram 4.1.

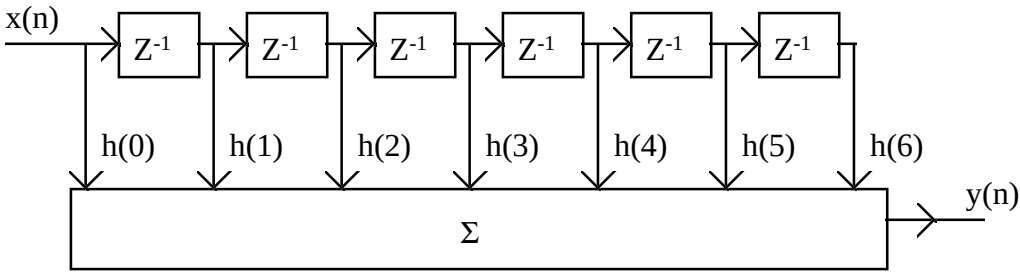


Diagram 4.1 :- **FIR Transversal Structure**

Diagram 4.1 shows the FIR filter structure as implemented by DSPLib. x(n) represents the input samples, and y(n) the resulting filtered output. The h(n) elements represent the filter coefficients, and it is these that determine the filters frequency response. For the DSPLib implementation, the coefficients should be 16-bit signed values. It is important to remember this when designing the filter as the quantisation will have a large influence on the overall response of the filter.

DSPLib doesn't support any filter design methods. It is required that the user determine their own coefficients either by traditional hand calculation methods, or alternatively the use of one of many commercially available software design packages. The later is recommended since different design methods can be tried, and also the analysis of finite word length effects is often possible by the use of simulation before the actual filter is implemented on the SH-1.

The implementation of an FIR filter, or indeed any type of digital filter depends upon the intended use. DSPLib provides two FIR routines, both implemented using a transversal structure. The difference in the routines is the execution. One routine takes a finite block of data and performs the filter operation upon it, whilst the second routine filters the data on a sample by sample process. Clearly the latter is more suited to real time applications, in particular situations where the sampling rate introduces harsh time constraints. More detail regarding the two functions is included in the routine description section of this chapter.

Because the FIR filter has no feedback, the filter is inherently stable. However, a drawback to this advantage is that to gain a particular performance criteria, an FIR will be of a higher order to that of the IIR. Therefore, in general unless any of the properties described at the start of this section are to be exploited, the IIR filter is used. The IIR filter is described below in 4.1.2.

4.1.2 Infinite Impulse Response Filters (IIR)

DSPLib offers four implementations of the IIR, all of which are described later in the routine description section of this chapter.

The IIR filter has feedback, and it is this feedback which leads to the filters strengths and weaknesses. The IIR requires fewer coefficients than FIR for the same set of specifications. IIR applications are normally in areas where sharp cut-off is required, along with high speed and in turn a high throughput of data. Of course this type of performance doesn't come free, and the price we pay for the IIR performance is decreased stability. Care must be exercised when designing the filter to ensure that the filter doesn't become unstable and result in an oscillating output.

The Z plane transfer function for the IIR is as follows :-

$$H(z) = \frac{Y(Z)}{X(Z)} = \frac{b_0 + b_1 Z^{-1} + b_2 Z^{-2}}{1 - a_1 Z^{-1} + a_2 Z^{-2}}$$

If we take the inverse Z transform of the above function and then separate out into difference equations, we arrive at the following :-

$$w(n) = (a_1 w(n-1) + a_2 w(n-2) + x(n))$$

$$y(n) = (b_0 w(n) + b_1 w(n-1) + b_2 w(n-2))$$

Where w(n) represents a centre tap in the filter, x(n) is the input data and y(n) is the resulting filtered data. Coefficients for the filter are represented as a and b and are ordered accordingly when being presented to the filter. The coefficient ordering and filter scaling is discussed in the routine description section.

All IIR filters in DSPLib are implemented using Direct Form II structure. This structure corresponds directly to the difference equations given above. The structure of this type of IIR filter is given below in Diagram 4.2

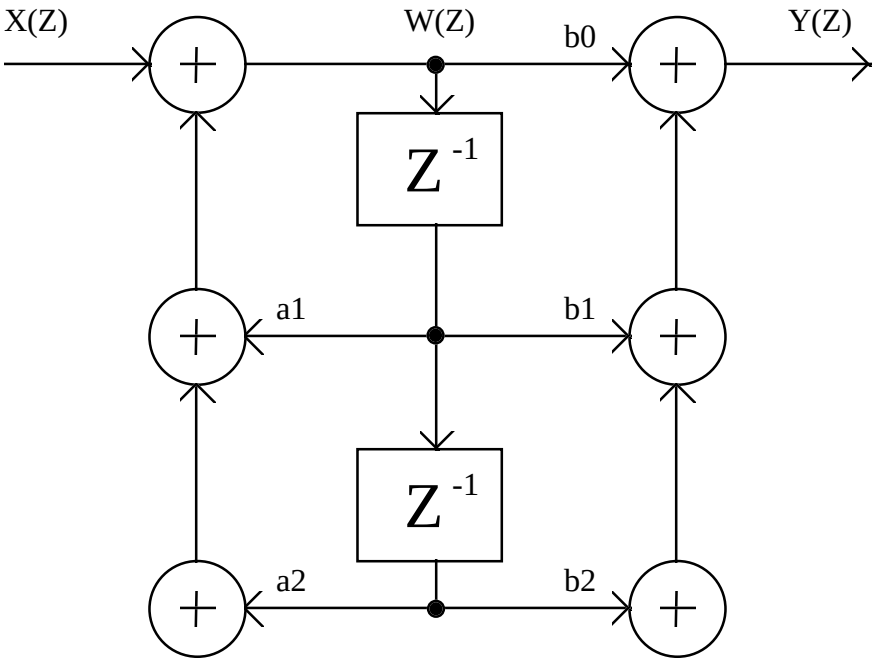


Diagram 4.2 :- Direct Form II Second Order Biquad

The above diagram shows a Direct Form II second order IIR biquad, which is the implementation used in DSPLib. If, as in most cases, a higher order filter is required, the biquads are cascaded. Hence when specifying your IIR filter order, it should be a factor of two.

The IIR routine descriptions detail the implementation of the filters. Care must be exercised when specifying shifts applied to the data, but again the routine description section discusses this issue in detail.

4.1.3 Filter Specification

Although actual filter design methods are not discussed in this document, it is however important that the reader has an understanding of methods for specifying filters. There are a number of excellent software filter design packages available, all of which require the user to specify the required response.

Taking a low pass filter as an example, a filter is typically specified by the following parameters :-

Pass Band Ripple
Stop Band Attenuation
Transition Width
Filter Order

The frequency response of an ideal low pass filter is shown below in Diagram 4.3. The frequency response cuts of immediately when we reach the cut off frequency Fc. Of course in reality, this kind of filter is not realisable, and so certain compromises must be reached wen arriving at an acceptable specification.

Diagram 4.3 :- **Ideal Low Pass Filter**

Because a finite number of filter coefficients are used in practice to specify a filter, and also the degree of quantisation introduced to both the coefficients and the data, a number of errors are introduced. Diagram 4.4 shows a practical filter response, and enables the user to gain an understanding of which parameters need to be specified when designing a filter.

The order of the filter has a great bearing on the response of the filter, with small order filters having a poor response.

If a sharp cut off is required, i.e. a small transition width, then the filter order must be very high. Clearly this introduces a number of factors which are limiting.

Firstly, a high order filter requires a large amount of computation due to its long length, this in turn introduces two problems. The length of the filter means that a long delay is introduced as the data ripples through the filter. Secondly, for each sample, a very large number of computations are required, and so the processing time is increased, hence reducing the real time capability of the system.

As well as computational problems, large filters introduce another problem. If a large number of coefficients are used, a large amount of memory is required to store both the coefficients and previous samples. Memory is expensive, and also introduces a limitation in speed due to the large number of memory accesses required.

When designing a filter it is necessary for the engineer to reach a number of compromises. The ideal response is played off against speed and cost, with a suitable system eventually being derived.

As well as the transition width expanding with a decrease in filter order, several other effects take place. The pass band region develops ripple, and the stop band attenuation becomes less. Clearly the designer should have a feel for the acceptable levels of ripple and attenuation such that an optimised filter may be designed. A practical filter response showing these effects is shown below in Diagram 4.4.

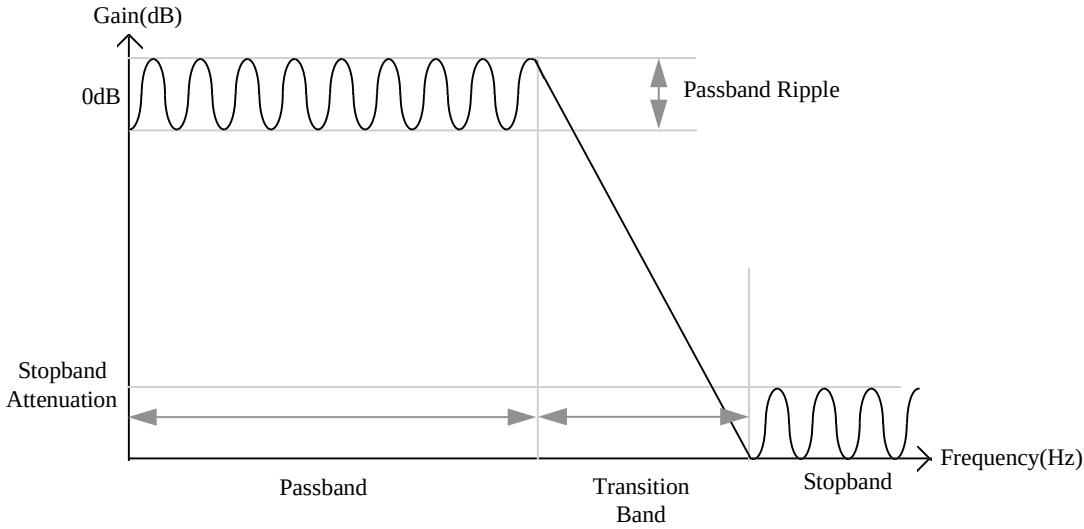


Diagram 4.4 :- **Actual Low Pass Filter Response**

As can be seen above, the actual response is very different to that desired. Typically when specifying a filter, the user indicates the acceptable level of ripple in the passband, the transition width and the required stopband attenuation.

The software design package will then normally suggest a filter order which would enable the stated criteria to be satisfied. Clearly if the user requires a filter with low pass band ripple, a small transition width and high stop band attenuation, a high order filter is required.

4.2 Routine Descriptions

4.2.1 Fir

The function FIR implements a Finite Impulse Response filter. The FIR filter has a transversal structure which is described in section 4.1.1 of this document. The related Ensigma documentation takes the reader through the definition of the function, but for the readers reference, the function is defined as follows :-

```
int Fir(output,input,no_samples,coeff,no_coeffs,res_shift,workspace)
```

Where,

short	output[]	output samples y
short	input[]	input samples x
long	no_samples	number of samples N to be filtered
short	coeff[]	array containing filter coefficients h
long	no_coeffs	length of filter, thus number of coefficients K
int	res_shift	right shift applied to each output
short	*workspace	pointer to filter memory

Using Fir

the use of the filter routines is relatively straight forward. It is important to initialise the filter memory (on chip), before calling the routine. The memory is initialised by calling the routine InitFir. This routine sets aside a section of on chip RAM for the storage of coefficients and also past samples which are required for FIR computation.

Coefficient Storage

Coefficients should be calculated as 16 bit fixed point. The calculated coefficients should then be stored in an array which can be seen by the compiler. Assuming that the array is called coeff, a typical file would follow the following format :-

short coeff[10] = {0,0,12,15,16,16,15,12,0,0};

Clearly when a large number of coefficients are used, it is good practise to store the coefficients in their own file, and include the array file name in the top of your main program.

Example

The following example outlines the use of the Fir routine. The example is written in C and is intended to provide the reader with a typical code layout.

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#include "testhead.h"

/*** Include File Containing data to be filtered ***/

#define NSAM     128
/* number of samples */
#define no_coeff  32
#define MAXSUM  67108864

void main(void)
{

/*** Declare Local Variables ***/

       short  coeff[no_coeff], filout[NSAM];
       short  *work;
       int    n, res_shift;
       long   sum;

/*** Array Containing Coefficients ***/

       coeff[32]={0, 0, 9, -1, 62, -253, 99, -168, 1283, -1017, 75, -3560,
       5196, -237, 7506,-26122, 17126, 17126, -26122, 7506, -237, 5196,-3560,
       75, -1017, 1283, -168, 99,-253, 62, -1, 9};

       res_shift = 16;            /*** Shift Applied To Output Data ***/

/*** Check Scaling Of Coefficients ***/

        for(n = 0; n < no_coeff; n++)
        {
              sum += fabs(coeff[n]);
        }

        if(sum >= MAXSUM)
        printf("Bad Coefficients");

/*** Initiate Filter Workspace ***/

        if(InitFir(&work, no_coeff) != EDSP_OK)
        printf("Problem With InitFir");


/*** Fir Filter The Signal ***/

       if(Fir( filout, signal, NSAM, coeff, no_coeff, res_shift, work) ==
       EDSP_BAD_ARG)
       printf("Filter Problem");

}
```

The above C code example shows a typical filter routine for processing blocks of samples.

In this case, the routine takes a block of 128 samples, and performs a filter operation on them accordingly. The filter is of length 32, and the pre-calculated samples are stored in the array coeff[] at the start of the program.

Before the filter is initialised, the coefficients are checked for scaling. It is suggested as a general rule of thumb that the absolute sum of the filter coefficients is less than 2^{26}. Checking this should help to avoid saturation, as long as the result shift is set accordingly. Obviously, in an environment where time is an issue, it is desirable to perform such checks during development, and remove the code for the final version when the routine is known to be working.

Next, the filter workspace has to be initialised. The routine InitFir() sets aside an area of on chip RAM for the filter routine to use as a scratch pad. The actual area is undefined, and will vary on each execution. It is therefore recommended that the user does not attempt to access this area at any time. When calling this routine, a pointer must be specified to the area, in our case the pointer 'work' is used. This pointer must be specified when calling the actual filter routine.

Finally in our C code example, the FIR filter routine is executed. The function Fir is called, with the following arguments:-

filout() = Samples from filter
signal() = Input samples for filter
NSAM = Number of input/output samples
coeff() = Filter coefficients
no_coeff = Number of filter coefficients (length of filter)
res_shift = Shift applied to output
work = Pointer to filter workspace in RAM

These arguments tie in with the Fir definition at the start of this section. If the arguments are outside of a particular range, the filter will not operate, and will instead return a bad argument flag. This flag, called EDSP_BAD_ARG is tested in the above example, and its' generation in turn is used to generate an error message to the user.

The flag EDSP_BAD_ARG is generated if the following occurs :-

no_samples	<	1
no_coeffs	<=	2
res_shift	<	0
res_shift	>	27

4.2.2 Fir1

This routine follows on directly from the above description of Fir, and it is therefore recommended that section 4.2.1 is read before moving on to this section.

Fir1 is identical to Fir except for the fact that it processes one sample at a time.

This routine is optimised for signal sample computation, and should be used instead of the above routine when 'no_samples' is set to one. The routines definition is outlined below :-

```
Fir1(output,input,coeff,no_coeffs,res_shift,workspace)
```

Where,

short	*output	Pointer to single output sample y(n)
short	input	single input sample x(n)
short	coeff[]	array containing filter coefficients h
long	no_coeffs	number of coefficients K (length of filter)
int	res_shift	right shift applied to output
short	*workspace	pointer to filter memory

Fir1 will give bit identical results to Fir, and uses the filter workspace to hold previous samples that are necessary for computation. The routine finds applications in areas where real time processing is required, and as a result the filter operation is applied on a sample by sample bases.

Since this function is essentially identical to the Fir routine, coefficients are stated in an identical manner to that shown in 4.2.1. Also, for the purpose of a software example, the code shown in 4.2.1 is relevant. For an identical filter as that shown, the above code may be re-written as follows :-

```
/*** Fir Filter The Signal ***/

if(InitFir(&work, no_coeff) != EDSP_OK)
printf("Problem With InitFir");

for(n=0; n < NSAM; n++)
{
       if(Fir1( filout, signal[n], coeff, no_coeff, res_shift,
       work)==EDSP_BAD_ARG)
       printf("Problem With Filter")
}
```

The above example is a 'snippet' of code which could replace the filter section of the example code shown in 4.2.1. The code uses a for loop to filter a number of samples one at a time. Clearly, in a real time environment, the loop would be omitted, and the filter operation would be initiated by the end of the ADC conversion, as an example.

It should be noted that before using Fir1, the filter workspace should be initialised by calling InitFir. This is the same initialisation routine as that used for Fir.

In order to minimise quantisation noise and saturation, the same scaling rules apply as that discussed earlier, being to keep the absolute sum of the coefficients to a level lower than 2^{26}. The resultant shift should also be set to 26 if overflow is considered to be a problem, however in most cases this simply isn't an issue.

4.2.3 Iir

Now we move onto IIR filters. The reader should be familiar with the basic concepts of IIR filters before reading the following routine descriptions. Section 4.1.2 of this document takes the reader through the basic ideas behind IIR filters, as well as explaining the filter structure for implementing IIR, and in particular the Direct Form II, as used in the DSPLib routines.

As described above, Iir is implemented using Direct Form II biquads. A biquad is a single second order filter stage which may be cascaded in order to form higher order filters. Now because second order filters are cascaded to implement the higher order designs, it is important for the reader to remember that the stated filter order must be a factor of two.

The Iir function is defined as follows :-

```
int  Iir(output,input,no_samples,coeff,no_sections,
         workspace )
```

where,

short	output[]	output samples
short	input[]	input samples
long	no_samples	number of samples to be filtered
short	coeff[]	filter coefficients
long	no_sections	number of second order sections
short	*workspace	pointer to workspace

Coefficient Storage

The definition is similar to that shown in the Fir descriptions earlier. However, due to the differences in structure, the Iir coefficients are specified in a very different manner.

The filter coefficients are stored in a single array. By looking at the transfer function of a single second order biquad, we can see that there are two groups of coefficients for each stage, being a_n and b_n.

$$H(z) = \frac{Y(Z)}{X(Z)} = \frac{b_0 + b_1 Z^{-1} + b_2 Z^{-2}}{1 - a_1 Z^{-1} + a_2 Z^{-2}}$$

Hence, as can be seen from the transfer function, for each second order biquad, the user must specify five coefficients. However, another factor is thrown into the equation which further complicates things. The Iir functions in DSPLib can have a shift applied to them at each stage, and the shift applied is stated in the same file as the coefficients. The Iir difference equations for Iir should make things clearer.

The equations are as follows :-

$$w(n) = (a_1 w(n-1) + a_2 w(n-2) + x(n)).2^{-15}$$

$$y(n) = (b_0 w(n) + b_1 w(n-1) + b_2 w(n-2)).2^{-a_{0k}}$$

If you are not familiar with these equations for the second order biquads, you should read section 4.1.2 before preceding here.

As can be seen above, each of the two ‘a’ coefficients have a 15 bit right shift applied, which means that the ‘a’ coefficients MUST be in Q15 format in order to ensure accurate results. The ‘b’ coefficients are subject to a user specified shift, which as described is contained in the same array as the coefficients. I.e. in the above equation if the’b’ coefficients are in Q15 format, a_{0k} should be set to 15. Hopefully, it therefore becomes clear that the a_{0k} term describes a shift rather than a coefficient. The k part of the term describes which cascaded biquad the term belongs to.

Hence, if two biquads were used, the coefficients would be specified as follows :-

$$\{a_{00},a_{10},a_{20},b_{00},b_{10},b_{20},a_{01},a_{11},a_{21},b_{01},b_{11},b_{21}\}$$

It should now be clear that in the above array, the terms a_{00} and a_{01} are not coefficients, they represent a right shift which is applied to the ‘b’ coefficients.

Example

There now follows a basic example of how the Iir function is typically used. The coefficients have been designed on a separate package and have been quantised to 16 bit resolution. The ‘a’ coefficients are in Q15 format, and the ‘b’ coefficients are in Q15 format, with the shift set to 15 as well, thus returning the solution in the correct form.

The C code example is as follows :-

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#include "testhead.h"

/*** Include File Containing data to be filtered ***

#define NSAM        128             /* number of samples */
#define no_sections 2
#define no_coeff    12

void main(void)
{
```

```
/*** Declare Local Variables ***/

       short  coeff[no_coeff], output[NSAM];
       short  *work;

/*** Array Containing Coefficients ***/

       coeff[no_coeff]={15,-29186,-11663,9008,18009,9008,15,-27423,
       28914,14004,27934,14004};

/*** Initiate Filter Workspace ***/

        if(InitIir(&work, no_sections) != EDSP_OK)
        printf("Problem With InitFir");

/*** IIR Filter The Signal ***/

       if(Iir( output, signal, NSAM, coeff, no_sections, work) ==
       EDSP_BAD_ARG)
       printf("Filter Problem");

}
```

The above C code takes the reader through a very simple example of how to integrate the Iir function within a program. The coefficients are included in the program for clarity, but could just as easy be in a different file that is pointed to by an include statement. The two a_0 values of 15 can be easily seen in amongst the coefficients, which constitute a shift right.

The filter workspace is initiated by calling InitIir. In a time critical system, this operation should be performed during the initial set-up, and not included in the main loop. The filter only has to be initialised once for a particular filter size, and is completely independent of the number of function calls made to Iir.

The final part of the program details the actual calling of the function. The data is taken from the array ‘signal’, which is omitted from the above code. In reality, ‘signal’ would be a series of time domain samples stored in its’ own file pointed to by an include statement. The function is tested for bad arguments, and returns an error message if there is a problem.

4.2.4 Iir1

This function follows directly on from the above description of Iir. Iir1 offers an IIR filter which performs a filter operation on a single sample, thus making it particularly suitable for real time operations. The function is defined as follows :-

```
int  Iir1( output, input, coeff, no_sections, workspace)
```

Where,

short	*output	pointer to filtered output sample
short	input	input sample
short	coeff[]	filter coefficients
long	no_sections	number of second order filter sections
short	*workspace	pointer to start of filter workspace

The function is used in exactly the same manner as Iir. Filter coefficients are specified in the same format as that described in 4.2.3, and the method of applying a shift to the output is also identical. When the user wishes to use Iir with no_samples set to one, this function should be used instead. Iir1 is optimised for single sample operation, and will offer some computational savings over Iir in this particular situation.

The software example given in the previous example is directly relevant here. For the same routine to be implemented using Iir1, the following change would be made.

```
/*** IIR Filter The Signal ***/

for( n=0; n <= NSAM; n++)
{
       if(Iir1(output,signal, coeff, no_sections, work) == EDSP_BAD_ARG)
       printf("Filter Problem");
}
```

I.e the filter section of the routine has been replaced by a for loop which repeatedly runs the Iir1 routine. Clearly in a real application it is unlikely that the function would be used in this manner, but it serves as an example of the function operation. This routine will return bit identical solutions to the example using Iir.

Before calling Iir1, it is important to initialise the filter workspace. Although the filter performs computation on one sample at a time, the filter must store previous input and output samples in order to perform the filter operation. Therefore, as with all the filter routines in DSPLib, the user must initialise the filter workspace. When using Iir or Iir1 the function InitIir must first be called. When calling this routine, the user must provide a pointer to the start of the workspace, and also indicate the size of the filter. The example in section 4.2.3 shows the use of InitIir, and is directly relevant to the use when calling Iir1.

4.2.5 DIir

This function implements an IIR filter with a direct form II structure. The filter is made up of a series of cascaded second order biquads. If the reader is not familiar with the structure of an IIR filter, they must consult section 4.1.2 before continuing here.

This routine offers a filter which is essentially the same as that offered by Iir, except that it allows double precision computation. This added precision is achieved by the inclusion of 32 bit coefficients. Input and output data is still 16 bit, but the added precision of the coefficients will increase the accuracy of the filter in situations where accuracy is an issue and could possibly affect filter stability.

The related Ensigma documentation explains how the double precision is realised, and it is the purpose of this document to demonstrate how to use the routine. DIir is defined as follows :-

```
int  Diir( output, input, no_samples, coeff, no_sections,
          workspace)
```

Where,

short	output[]	filter output samples
short	input[]	filter input samples
long	no_samples	number of samples to be filtered
long	coeff[]	32 bit filter coefficients
long	no_sections	number of second order sections (biquads)
long	*workspace	pointer to filter workspace

Using DIir

The use of DIir is essentially the same as using Iir, only the coefficients are 32 bit. Before calling DIir it is important to initialise the filter workspace. This is achieved by calling the function InitDIir. When calling this function, the user must specify the number of second order sections in the filter, and also provide a pointer to the area in RAM which this function sets aside for the filters use. The reader can see this function in the C code example below. It is important that the user doesn't try to access the filter workspace at any point for two reasons. First of all, the area is undefined, and so the user has no idea of the location or size of the area before or during execution.
Secondly, the filter will access the RAM to store past samples. The user does not have access to the method of storage, and so intervention could result in the failure of the filter.

Coefficient Storage

Since this function implements an IIR filter, the method of storing coefficients is similar to that shown in section 4.2.3. However, due to the double precision of the coefficients, the method of applying shifts to the results is different. In order to understand the filters operation, we must first look at the double precision filter difference equations.

$$w(n) = [a_{1k} d_k(n-1) + a_{2k} d_k(n-2) + 2^{31} x(n)].2^{-31}$$

$$y(n) = [b_{0k} d_k(n) + b_{1k} d_k(n-1) + b_{2k} d_k(n-2)].2^{-a_{0k}}$$

The above equations are essentially the same as those given for the Iir function. The coefficients are stored in a single array in the same format, which is outlined below.

$$\{a_{00},a_{10},a_{20},b_{00},b_{10},b_{20},a_{01},a_{11},a_{21},b_{01},b_{11},b_{21}.....\}$$

The area where this function differs is the specification of shifts applied to outputs of biquads. As the difference equations show, the 'a' coefficients must be specified in Q31 format. The output of each biquad can be shifted by the use of a_{0k} and it is important that the reader fully understands how the shifts work.

When the 16 bit input x(n) is applied to the filter, it is firstly multiplied by 2^{16} to put it into Q31 format which is suitable for the filter arithmetic. This shift must be compensated by the user at the output in order to gain accurate results. The following two block diagrams should help to clarify matters.

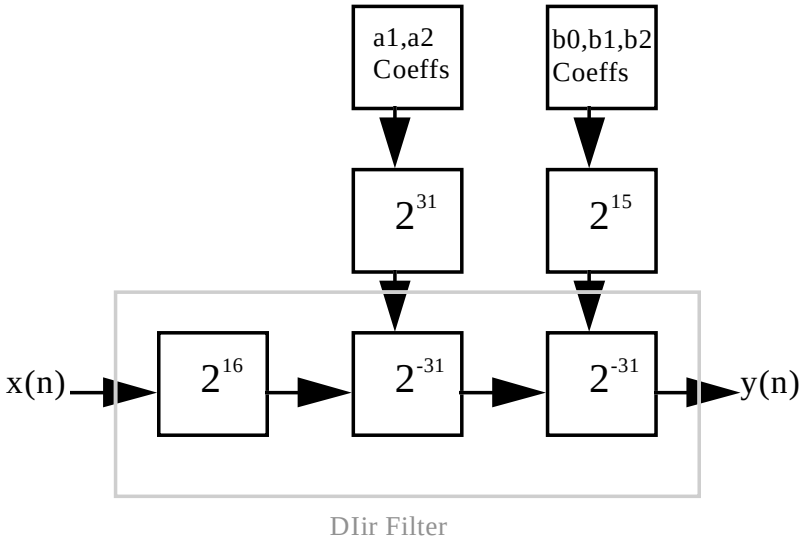


Diagram 4.5 :- **Shifts Applied In A Single Second Order Biquad**

Referring to the above diagram, the method of applying shifts to a single second order biquad can now be explained.

The input samples, x(n) are immediately multiplied by 2^{16} by the function, the user has no control over this. Next, all the 'a' coefficients must be specified in Q31 format, as they are multiplied by 2^{-31} by the filter function, and again the user has no control over this. Finally we introduce the 'b' coefficients, which are multiplied by a user specified shift. This shift, a_{0k} must not only compensate for the format of the 'b' coefficients, but also compensate for the original shift applied to the input data.

In the diagram above (4.5), the coefficients are stored in Q15 format. The final result shift applied is therefore 2^{-31}. The reason for this is as follows : The coefficients are in Q15 format, which means that they have been multiplied by 2^{15} to put them in correct form. Also the shift has to compensate for the original 2^{16} shift applied at the start of the process. Therefore we arrive at the required shift of 2^{-31} which will compensate for both shifts and return the solution in the correct format, whilst allowing the processor to perform double precision computation.

If the filter order is higher than two, i.e. more than one biquad is used, the compensation doesn't have to be applied until the last stage, thus allowing the intermediate stages to be expressed in Q31 format.

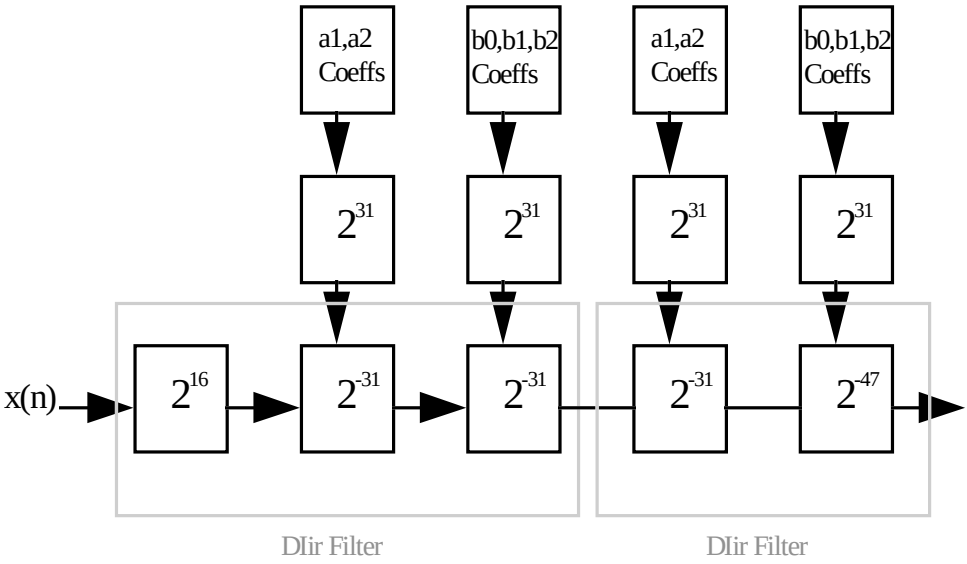


Diagram 4.6 :- **Shifts Applied In A Fourth Order Filter**

The function allows the final output stage shift to be expressed right up to and including 48. This means that on a fourth order system and higher, the filter can be a true dual precision. In the example above, it is clear to see that the final shift applied is 47, which compensates for the Q31 format of the 'b' coefficients, and also compensates for the original 2^{16} shift applied at the start. It is very important to remember that such a large shift is only possible on fourth order filters and higher, and as a result, it is not possible with this function to implement a true second order double precision filter.

Example

The best way to reiterate the above theory is by way of an example. This example is written in C as are all the examples in this text. The example shows the array containing the filter coefficients, as well as the function calls to implement the filter. The filter coefficients were calculated on a design package, and implement a fourth order low pass filter. The values obtained from the package were as follows :-

Design Package Solution	Q31 Format
a_{10} = 0.6149292	1320550402
a_{20} = 0.4001465	859308065
b_{00} = 0.1794281	385318910
b_{10} = -0.3097076	-665092006
b_{20} = 0.1794281	385318910

a_{11} = -0.2411804	-517930965
a_{21} = 0.8682861	1864630202
b_{01} = 0.7138977	1533083637
b_{11} = -0.7098694	-1524432929
b_{21} = 0.7138977	1533083637

Hence the result of the filter design operation is shown on the left, and the Q31 format value is shown on the right. These values are then used in the double precision filter example outlined below. The reader should take particular notice of the coefficient array, and especially the a_0 values which describe the shift applied during the filter operation.

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#include "testhead.h"

/*** Include File Containing data to be filtered ***/

#define NSAM        128             /* number of samples */
#define no_sections 2
#define no_coeff    12

void main(void)
{

/*** Declare Local Variables ***/

       short  coeff[no_coeff], output[NSAM];
       short  *work;

/*** Array Containing Coefficients ***/

       coeff[no_coeff]={31, 1320550402, 859308065, 385318910, -665092006,
       385318910, 47, -517930965, 1864630202, 1533083637, -1524432929,
       1533083637};



/*** Initiate Filter Workspace ***/

        if(InitDIir(&work, no_sections) != EDSP_OK)
        printf("Problem With InitDIir");

/*** DIIR Filter The Signal ***/

       if(DIir( output, signal, NSAM, coeff, no_sections, work) ==
       EDSP_BAD_ARG)
       printf("Filter Problem");

}
```

The above example filters a set of data samples 128 long using the double precision IIR filter. Before calling the filter routine, the filter workspace is initialised by calling InitDIir. This function requires two arguments, one being a pointer to the workspace, and the second being the number of second order sections the filter contains, thus indicating to the function the size of RAM required.

The filter coefficients are stored in an array in the main program, and this should clearly demonstrate the shifts which are required. All the coefficients are in Q31 format (the 'a' coefficients must be in Q31 format), and the two shifts applied are 31, and 47. The shift of 47 compensates for the 16 place shift applied at the input as described earlier.

The actual filter routine is called at the end of the program, and will filter the block of 128 samples, and then finish. The actual calling of the routine and arguments used is identical to Iir described in 4.2.3, except that the solution although still 16 bit is of a higher precision.

4.2.6 DIir1

This function offers exactly the same filter operation as that described in 4.2.5. The function implements a double precision IIR filter, but only processes a single sample at a time. This makes DIir1 particularly suitable for real time operation where the user wishes to perform computation on a sample by sample bases. The function is defined as follows :-

```
int  DIir1( output, input, coeff, no_sections, workspace)
```

Where

short	*output	pointer to output sample
short	input	input sample
long	coeff[]	array containing 32 bit coefficients
long	no_sections	number of second order sections
long	*workspace	pointer to start of filter workspace

The filter is used in exactly the same manner as DIir, and the reader should consult section 4.2.5 for full details of the operation.

The method for specifying the filter coefficients is also exactly the same, i.e. they are all 32 bit and require the same methods of applying shifts as that described earlier. If the user wishes to use DIir with 'no_samples' set to one, this function should be used instead. DIir1 has been optimised for single sample computation and will provide an increase in speed compared to DIir.

To modify the C code example given in the previous section such that DIir1 can be used, the following change would be made.

```
/*** DIIR Filter The Signal ***/

for( n=0; n <= NSAM; n++)
{
      if(DIir1(output,signal, coeff, no_sections, work) == EDSP_BAD_ARG)
      printf("Filter Problem");
}
```

As should be clear from the example code above DIir1 is the same as DIir only the user does not have to specify the number of samples to be processed since it only processes one at a time.

Before calling DIir1, it is essential that the filter workspace is initialised.

This is achieved in exactly the same manner as that for DIir, that is by calling InitDIir and specifying the number of second order sections the filter contains, and also providing a pointer to the assigned memory location. The use of this initialisation function can be seen in the software example in section 4.2.5.

The code above is intended only to indicate the operation of the function. Imbedding the function in a For loop will actually increase the run time compared to the use of DIir, and so is not recommended. However, for evaluation purposes, the user will find that both DIir and DIir1 will return bit identical results.

Even though the filter only processes one sample at a time, it is necessary to store past samples to feedback into the filter. These samples are stored in the filter workspace, and the very oldest samples are discarded as the number of single sample computations increase.

As a final note in this section it should be pointed out to the user that using the double precision routines will most probably not be necessary, and will impose a time constraint. In most applications, the accuracy of the IIR functions will be suitable, and due to the simpler nature of the implementation, the IIR functions run an order of magnitude faster on the SH1. The use of the double precision filters should be contained to areas where the user feels that single precision will adversely affect the filter operation and lead to problems such as instability. As an example of the speed difference, on a 10MIPS SH-1, Iir1 takes 18.1uS/sample where as DIir1 takes 48.0uS/sample.

4.2.7 Lms

This is the final filter structure offered by DSPLib, and comes in two forms LMS and LMS1. This section describes the background behind the LMS filter, and also includes an example of the routine performing a filter operation.

LMS stands for Least Mean Squares, which in turn describes the algorithm which is used in this adaptive filter. The LMS adaptive filter is based around an FIR filter, and it is assumed that the reader is familiar with its' basic operation. If this is not the case, the reader should consult section 4.1.1 of this document which explains the basic concepts behind the FIR, as well as showing the transversal structure which is incorporated into this adaptive algorithm.

The function Lms is defined as follows :-

```
int  Lms( output, input, ref_output, no_samples, coeff,
          no_coeffs, res_shift,  conv_fact, workspace)
```

Where,

short	output	output samples y(n)
short	input	input samples x(n)
short	ref_output	desired output values d(n)
short	no_samples	number of samples to be filtered
short	coeff[]	filter coefficients h(n)
long	no_coeffs	number of coefficients
int	res_shift	right shift applied to output
short	conv_fact	convergence factor 2μ
short	workspace	pointer to filter workspace

The LMS routine provides a filter that will adapt to a reference signal given to it. This means that no filter design is necessary, the filter will adjust its own coefficients using the LMS algorithm such that the reference signal provided to it is included from the main filter signal.

As an example, if a filter is supplied with a reference sine wave of frequency Fs/4, and an input signal containing a number of sine waves such as Fs/2,Fs/4,Fs/8 and Fs/16, the output of the filter would be Fs/4. Hence the filter has adapted its coefficients such that the Fs/4 component of the signal is included.

The LMS filter is mathematically described as follows :-

FIR Filter :-

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k) x(n-k) \right]$$

LMS Adaptive Algorithm :-

$$h_{n+1}(k) = h_n(k) + 2\mu e(n) x(n-k)$$

Where h(k) are the filter coefficients which are adapted. μ determines the convergence rate, i.e. the rate at which the coefficients converge and provide the required output. Clearly the faster the convergence rate, the less accurate the output. It is up to the user to arbitrarily test different values until a suitable response is achieved.
The same scaling conventions apply to this filter as with the Fir routine. However, it is not guaranteed that the adaptive algorithm will keep to the convention, and saturation may occur. If this is the case, it is most likely that the filter is having trouble adapting, and the filter order should be increased to rectify this.

Diagram 4.7 shows a block diagram showing the implementation of the filter.

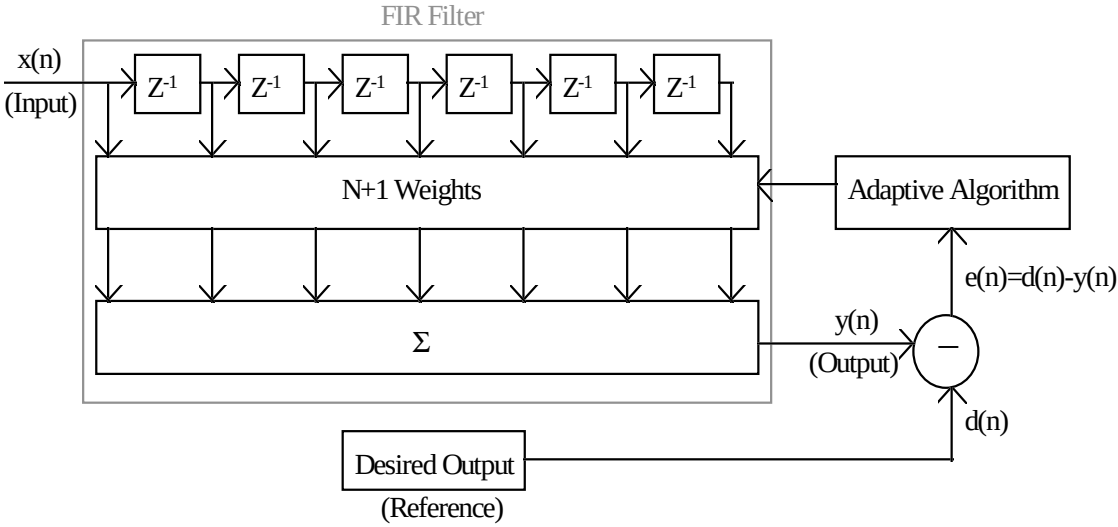


Diagram 4.7 :- **LMS Filter Block Diagram**

With reference to the above diagram, the area within the grey block is the standard FIR filter implemented in a transversal structure. The output of the filter is subtracted from the desired signal output (reference) which in turn generates an error signal e(n). This error signal is used by the adaptive algorithm to update the filter weights (coefficients) in order to reduce the error signal, and thus eventually arrive at the desired filter response. Initially, when the filter hasn't started to adapt, the filter coefficients are set to an arbitrary value. The value chosen should not make a difference to the final value reached, however, it may have some affect on the rate at which it meets the ideal weights.

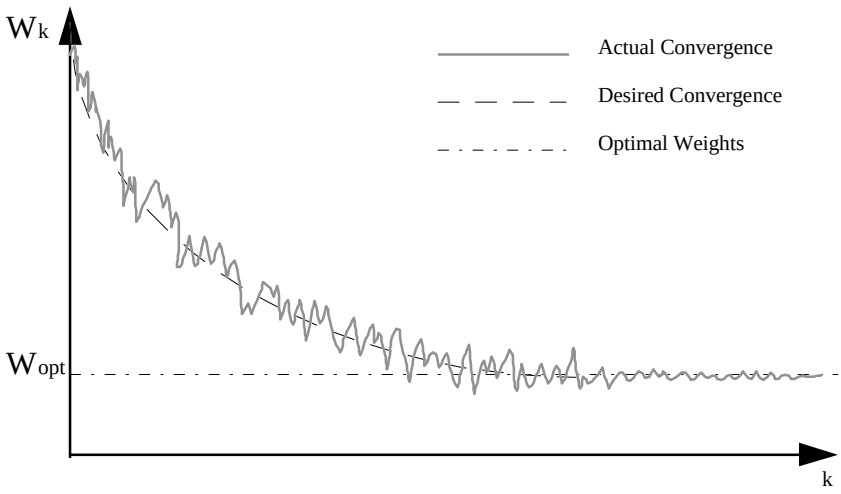


Diagram 4.8 :- **Convergence Of Filter Weights**

The above diagram illustrates the variations in the filter weights as the feedback adjusts them accordingly. Although the final weights are very close to the desired values, they never actually settle there, but will instead fluctuate around the optimum value.

Example

Now that the basic theory has been covered, an example should help to reiterate any problem areas. The example takes an input signal which consists of five sine waves added together, and adaptivly filters all but one of the components. The reference input is the frequency component that is left at the end. The C code is given first, followed by an in depth explanation of the process, including the analysis of various filter lengths.

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>

#define NSAM      128              /* number of samples */
#define TWOPI     6.283185307
#define no_stage  1
#define MAXSIZE   NSAM
#define NSAMS    (NSAM/2)
#define TWOMU    32767
#define ncoeff   8

/*****************   Exercise   To   Implement   An   LMS   Adaptive   Filter
******************/

void main(void)
{
       short   coeff[ncoeff], ref[NSAM], noise[NSAM], noisea[NSAM],
       fftout1[NSAM], sine1[NSAM], sine2[NSAM], sine3[NSAM], sine4[NSAM],
       sine5[NSAM], signal[NSAM], output1[NSAMS], filout[NSAM];

       int     n, k, loop, res_shift;
       long    sum, len;
       short   *work;

       len     =   NSAM;
       res_shift = 16;
       loop = 100;

/** Coefficients For Adaption **/

       coeff[ncoeff]={1,1,1,1,1,1,1,1};

/*** Generate Sine Waves ***/

       printf("Generating Sine Wave");
       k = len / 8;
       for(n = 0; n < len; n++)
       {
             sine1[n] = floor(3162 * sin(TWOPI * k * n / len) + 0.5);
       }

       k = len / 16;
       for(n = 0; n < len; n++)
       {
             sine2[n] = floor(6553 * sin(TWOPI * k * n / len) + 0.5);
       }

       k = len / 4;
       for(n = 0; n < len; n++)
```

```
        {
              sine3[n] = floor(3162 * sin(TWOPI * k * n / len) + 0.5);
        }

        k = len / 32;
        for(n = 0; n < len; n++)
        {
              sine4[n] = floor(3162 * sin(TWOPI * k * n / len) + 0.5);
        }

        k = len / 64;
        for(n = 0; n < len; n++)
        {
              sine5[n] = floor(3162 * sin(TWOPI * k * n / len) + 0.5);
        }

        printf("Done");


/*** Create Signal To Be Filtered By Adding Sine waves ***/

         for(n = 0; n < len; n++)
         {
              signal[n] = (sine1[n] + sine2[n] + sine3[n] + sine4[n] +
              sine5[n]);
              ref[n] = (sine3[n] );
         }

         if(InitLms(&work, ncoeff) != EDSP_OK)
         printf("Problem With InitFir");


/*** Perform Adaptive Filter Operation ***/
/*** Adapt coeffs 100 times ***/


for(n=0; n < loop; n++)
         {
              if(Lms(filout, signal, sine3, NSAM, coeff1, ncoeff, res_shift,
              TWOMU, work) == EDSP_BAD_ARG)
              printf("Problem With LMS Operation");

         }

         free(work);

}
```

The above code performs two tasks. First of all the code generates the signals which are to be filtered. Clearly this is for test purposes only, and in a real system, the data would come from a device such as an ADC. The signal to be filtered consists of five sine waves all added together. The corresponding frequencies of these signals are Fs/4, Fs/8, Fs/16, Fs/32 & Fs/64. The sum of these waves forms the data set to be filtered, consisting of 128 samples. Next, the reference signal is generated, which is chosen to be Fs/4. This means that the coefficients should adapt and remove all bar the Fs/4 component from the signal.

Next step in the code is to initialise the filter workspace by calling the function InitLms. This function is used in exactly the same manner as the previously described initialisation routines. So finally in the code, we arrive at the actual filter operation. The function is called 100 times in order to demonstrate how well the coefficients adapt. The calling of the function is fairly self explanatory, and should not cause any problems. The function Lms modifies the filter coefficients stored in coeff[], and as a result they should be stored in on chip RAM, thus allowing fast access. If the coefficients are stored off chip, it is a good idea to copy them into on chip RAM. The spectrums of the signals involved in the filter operation are displayed below.

Diagram 4.9 shows the five sine waves added together which forms the input to the adaptive filter. This signal was then adaptivly filtered as described above with eight coefficients. The value of 2μ was chosen to be 32767, and wasn't changed during the whole of the testing period. The user may wish to start off with a similar value in order to establish operation, then change accordingly in order to achieve faster convergence.

The result of filtering with eight coefficients or weights is displayed in figure 4.10. As can be seen, the unwanted elements of the signal have been attenuated, but not removed. The reason for this is that the small filter order physically cannot apply the attenuation required to remove the unwanted elements.

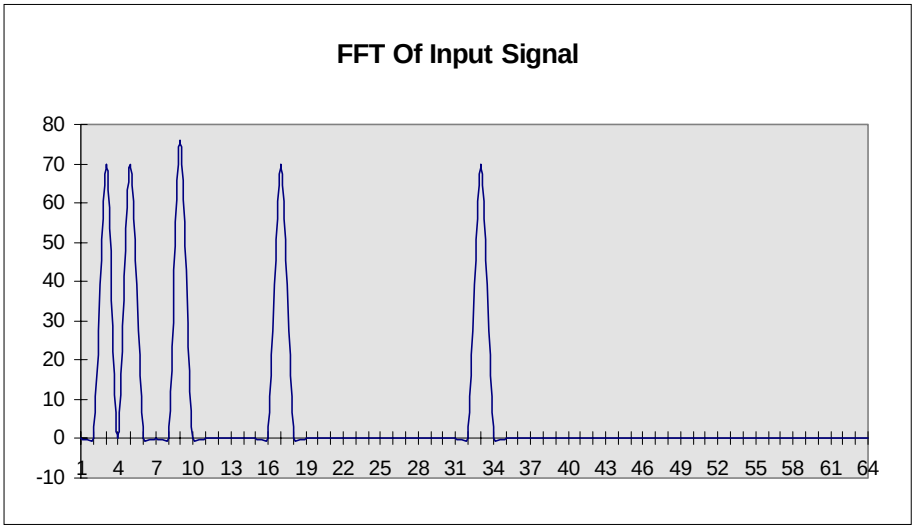


Diagram 4.9 :- **Spectrum Of Input Signal To Adaptive Filter**

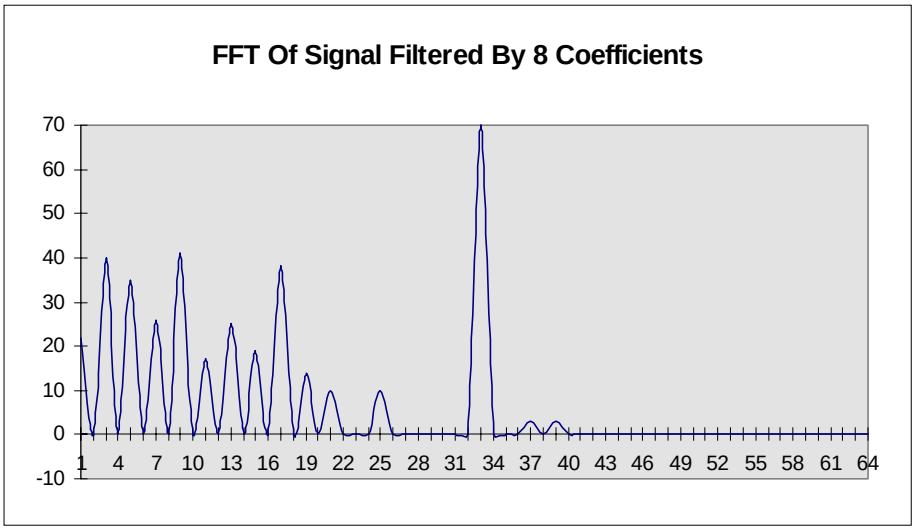


Diagram 4.10 :- **Spectrum Of Filtered Signal Using 8 Coefficients**

As can be seen in Diagram 4.10, the filter has adapted to the reference signal, but has failed to filter the unwanted components, even after one hundred operations. Therefore, for the purposes of demonstration, the number of coefficients, i.e. the FIR filter length is increased to thirty two. This means that in the software example above, ncoeff = 32, and the array coeff consists of thirty two elements each containing one.

With the increased coefficients, the program was ran, and the results shown in Diagram 4.11 below. It is clear to see that the increase in coefficients increased the selectivity of the filter, and was more capable to reject the unwanted components from the signal.

In a real application, it is good practice to try different filter lengths on a trial and error basis, thus enabling the user to arrive at the most efficient implementation. Clearly large filters are bad news since they require more memory, and as a result run slower.

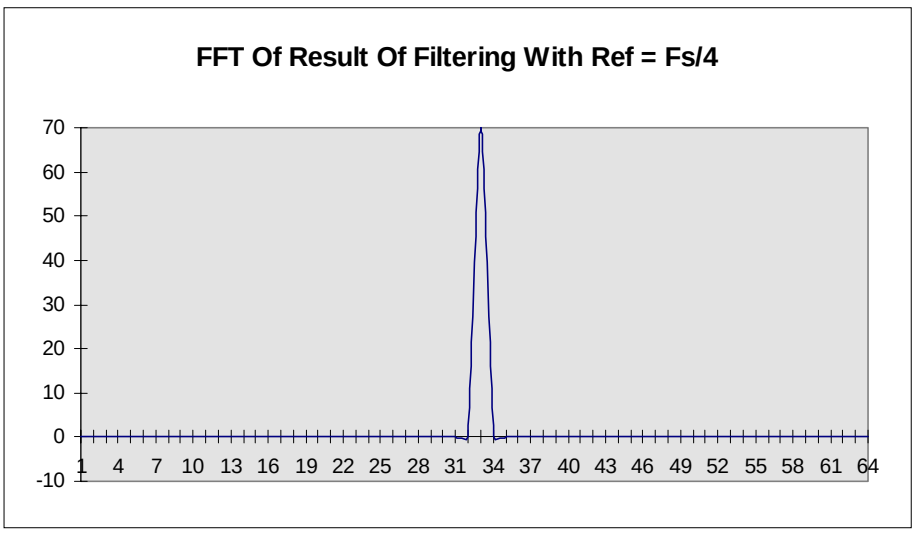


Diagram 4.11 :- **Result Of Adaptive Filtering With 32 Tap Filter**

As can be clearly seen above, with a filter length of 32, the unwanted signal components have been completely removed. It is therefore unnecessary to use a filter of a higher order for this particular application.

4.2.8 Lms1

This function is the final one to be discussed from the filter section of DSPLib. The function is identical in every way to the function Lms, and the in depth discussion in the previous section is directly relevant here. The reader should consult both section 4.1.1 on FIR theory, and also section 4.2.7 detailing adaptive filters.

Lms1 provides an algorithm which implements the Least Mean Squares adaptive algorithm on a single sample. Hence the function finds particular application in real time processing.

The function Lms1 is defined as follows :-

```
int  Lms1( output, input, ref_output, coeff, no_coeffs,
          res_shift, conv_fact, workspace)
```

Where,

short	*output	output sample y(n)
short	input	input sample x(n)
short	ref_output	desired output value d(n)
short	coeff[]	filter coefficients h(n)
long	no_coeffs	number of coefficients
int	res_shift	right shift applied to output
short	conv_fact	convergence factor 2μ
short	workspace	pointer to filter workspace

By examining the filter definition it should be clear to the reader that the only difference between the two functions is that no_samples has been omitted. In applications where Lms is used with no_samples set to one, Lms1 should be used instead. Lms1 has been optimised for single sample operation, and will provide time savings over the use of Lms. If more than one sample at a time requires sampling, Lms should be used since it is quicker than using Lms1 in a for loop.

Before calling Lms1, the filter workspace should be initialised. This is achieved by calling the function InitLms. This is the same routine as that used to initialise the Lms filter workspace. The workspace is used by the filter to store past samples which are required for computation. It is important that the user makes no attempt to access this RAM at any time during filter execution in order to ensure filter accuracy and speed.

Because of the adaptive algorithm, Lms is a degree slower than Fir to operate. It is therefore a possibility that the Lms routine can be used to generate filter coefficients during development, and once finished can be inserted into a fixed FIR in order to speed up implementation. Clearly if constant adaptive filtering is required this method is not suitable.

5.0 Convolution and Correlation

Introduction

DSPLib offers five functions in this section, two related to correlation and three related to convolution.

This text offers the reader a brief introduction to the theory behind convolution and correlation, before describing each function individually. The background theory is intended only as an introduction, and a relevant DSP text should be consulted for any additional information required.

The five functions offered by DSPLib are as follows :-

ConvComplete
ConvCyclic
ConvPartial
Correlate
CorrCyclic

The differences between each function are described in the routine descriptions section of this report (5.2).

Applications of these functions include areas such as image processing where two images are compared for similarity, and also range and distance finding in areas such as sonar and radar.

5.1 Background Theory

5.1.1 Convolution

The simplest way to describe convolution is to state that convolution in the frequency domain is equivalent to multiplication in the time domain. The equation for convolution is given below :-

$$y(m) = \left[\sum_{i=0}^{W-1} w(i)x(m-i) \right].2^{-res\_shift}$$

It should be noticed that this identical to the expression used to describe an FIR filter. The operation of such a filter is that the input samples are convolved with the filter coefficients and result in a filtered output. The convcomplete function will return bit identical results to fir, when used in this manner.

When not used for filtering, the convolution function convolves the two input data sets and returns a solution in y(m). The solution returned describes how an input signal interacts with the system in order to generate a particular output.

Relating correlation and convolution together is fairly simple. The convolution of two data sets is identical to the cross-correlation of one data set with the time reversal of the second data set relative to that used in convolution. Correlation is described below, and is important in the understanding of convolution.

5.1.2 Correlation

Correlation essentially highlights similarities in data sets, and as a result finds applications in areas such as robotic vision and range finders. The equation for correlation is given below :-

$$y(m) = \left[\sum_{i=0}^{W-1} w(i)x(i+m) \right].2^{-res\_shift}$$

The two data sets w(i) and x(i) are correlated, and the output y(m) will increase in amplitude when the two inputs exhibit similarities. Hence, in range finder applications, the transmitted pulse is correlated with the received pulse. A peak will occur when the transmitted data is received, and thus enables the distance to be computed. Alternatively, the output can decrease in amplitude, indicating a negative correlation, i.e. similar signals that are out of phase.

Correlation can be used to examine the properties of a given data set. If a set of data is correlated against its self, the process is referred to as cross correlation. This method can be used to remove noise from a periodic signal, and is shown by way of an example later in this text.

The library offers a number of different variations of the above two functions. Essentially, the same task is performed, and its only the range over which the computation is computed that differs. The actual differences are described below in the routine descriptions section (5.2), and also includes a number of examples written in C.

The theory behind convolution and correlation is beyond the scope of this text. The descriptions and examples that follow should provide the reader with an outline of the subject, and should help to allow the user to access the functions operation without having to become involved with complex general mathematical descriptions often found in text books. It should also help to highlight typical application areas, thus presenting ideas for its use.

Although the example programs are shown written in C, it is recommended that the optimised source code be used for the actual DSPLib function calls. The use of C compiled code for DSP functions is not suitable, and will introduce a large time overhead. All the optimised DSPLib functions are included in the file ensigma.lib. This file should be included in the link command file, when linking the programs together.

5.2 Routine Descriptions

5.2.1 ConvComplete

This function offers complete convolution between two data sets. The function is essentially the same as Fir, and will return identical results given the same data. The function is defined as follows :-

```
int ConvComplete(output, iw, ix, iw_size, ix_size,
     res_shift)
```

where,

short	output[]	output data (size W+X-1)
short	iw[]	input w
short	ix[]	input x
long	iw_size	size of w (size W)
long	ix_size	size of x (size X)
int	res_shift	right shift applied to output

This function completely convolves the input data sets, and inserts zeros at the start and end of the two arrays to enable this to take place.

The function is mathematically described as follows :-

$$y(m) = \left[\sum_{i=0}^{W-1} w(i)x(m-i) \right] . 2^{-res\_shift} \qquad 0 \le m < W + X - 1$$

In order to ensure correct operation, a number of argument checks are performed, and are as follows :-

iw_size < 1
ix_size < 1
res_shift < 0
res_shift > 27

The flag EDSP_BAD_ARG is returned, and the function aborted if a bad argument is detected.

As is the case with most of the DSPLib examples, optimum performance will be obtained if the program and data are located in on chip memory. Clearly the ability to achieve this depends upon the Microcontroller used, and program size. It is recommended that at the very least one of the input arrays be located in on chip RAM, which will help to avoid bus conflicts.

By way of an example of complete convolution, if the data set ‘1234’ is to be completely convolved with the set ‘5678’, the following shifts would be performed :-

00001234000	⇐ Held static
00000005678	⇐ 1st convolution
00000056780	⇐ 2nd convolution
00000567800	⇐ 3rd convolution
00005678000	⇐ 4th convolution
00056780000	⇐ 5th convolution
00567800000	⇐ 6th convolution
05678000000	⇐ 7th convolution

Note how the insertion of zeros enable the two data sets to be completely convolved. One data set is shifted relative to the other, and the convolution is taken.

This example also indicates how the size of the output array is larger than the input arrays. In the case of ConvComplete, the output array size should be W+X-1. It is essential that this is ensured since failure to do so will result in the corruption of the stack.

5.2.2 ConvCyclic

ConvCyclic offers a function which cyclically convolves two data sets and returns the result in a third array. The function is defined as follows :-

```
int ConvCyclic( output, iw, ix, size, res_shift)
```

Where,

short	output[]	output data
short	iw[]	input w
short	ix[]	input x
long	size	size of both input arrays (N)
int	res_shift	right shift applied to output

Because the two input arrays are cyclically convolved, they must be the same size. It is up to the user to zero pad the smaller of the two arrays if their sizes are different. Cyclic convolution is mathematically described as follows :-

$$y(m) = \left[\sum_{i=0}^{N-1} w(i)x(|m-i+N|_N) \right].2^{-res\_shift} \qquad 0 \le m < N$$

In order to ensure correct operation, a number of argument checks are performed, and are as follows :-

size < 1
res_shift < 0
res_shift > 27

If any of the above are true, the function aborts and returns the flag EDSP_BAD_ARG.

As is the case with most of the DSPLib examples, optimum performance will be obtained if the program and data are located in on chip memory. Clearly the ability to achieve this depends upon the Microcontroller used, and program size. It is recommended that at the very least one of the input arrays be located in on chip RAM, which will help to avoid bus conflicts.

Clearly the three convolution functions offered by DSPLib perform the same mathematical operation between two sets of data. The way in which they differ is in the manipulation of arrays, and in the results that are included in the output. In the previous example, the two arrays were completely convolved, and zero padded to ensue that all results were obtained. Cyclic convolution is different to that, and is now shown in an example.

Assume that the two arrays ‘1234’ and ‘5678’ are two be cyclically convolved. As before, one of the arrays is held static, and the other shifted around it. After each shift, the two arrays are convolved. This is performed until every element is convolved with every other element. E.g.

1234 ⇐ Held Static
5678 ⇐ 1st convolution
8567 ⇐ 2nd convolution
7856 ⇐ 3rd convolution
6785 ⇐ 4th convolution

Hence it should become clear that the output array is the same size as the input arrays. In the above example, 5678 was rotated four times in order to perform cyclic convolution. This is shown in the diagram below, where the outer circle is rotated.

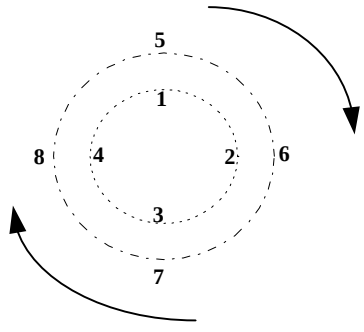


Diagram 5.1 :- **Cyclic Rotation**

5.2.3 ConvPartial

ConvPartial offers partial convolution between two data sets. The function is essentially the same as ConvComplete, but will not include results that are obtained outside the data sets. The function is defined as follows :-

```
int ConvPartial( output, iw, ix, iw_size, ix_size,
     res_shift)
```

Where,

short	output[]	output data (size X-W+1)
short	iw[]	input w
short	ix[]	input x
long	iw_size	size of w (size W)
long	ix_size	size of x (size X)
int	res_shift	right shift applied to output

The input arrays can be different sizes, however, the solution does not include elements derived from outside the array boundaries.

The function is mathematically described as follows :-

$$y(m) = \left[\sum_{i=0}^{W-1} w(i)x(m+W-1-i) \right].2^{-res\_shift} \qquad 0 \le m \le X - W$$

In order to ensure correct operation, a number of argument checks are performed, and are as follows :-

iw_size < 1
ix_size < 1
ix_size < iw_size
res_shift < 0
res_shift > 27

If any of the above cases are true, the function will abort and return the flag EDSP_BAD_ARG.

As is the case with most of the DSPLib examples, optimum performance will be obtained if the program and data are located in on chip memory. Clearly the ability to achieve this depends upon the Microcontroller used, and program size. It is recommended that at the very least one of the input arrays be located in on chip RAM, which will help to avoid bus conflicts.

When using the function it is important to pay attention to the array sizes in order to avoid stack corruption.

The inputs can be different sizes, but input w must be larger than input x. The size of the output array is clearly essential, and is defined as X-W+1.
By way of an example, the partial convolution method is now explained. Following on from previous examples in this section, we will convolve two data sets, one containing 1234 and the other 56789. Clearly, one array must be larger than the other in order to satisfy the argument checking described earlier.

```
    1234        ⇐ Held static
   xxx5678
   xx56789
   x56789
   56789        ⇐ 1st convolution
  56789         ⇐ 2nd convolution
 56789x
56789xx
56789xxx
```

As can be seen, the larger array is shifted across the smaller one. There are only two instances when the convolutional result is made up from results obtained within both arrays. The x's represent areas where results would be calculated from outside array boundaries, such as in ConvComplete, but in this case are rejected. Hence, in the above example, the output array would contain two elements, which holds true to the equation X-W+1 = 5-4+1 = 2.

5.2.4 Correlate

Correlate offers complete correlation between two arrays, and deposits the solution in a third array. Convolution is the same as correlation, only one of the input arrays is in reverse order, i.e. time reversed. The function is described as follows :-

```
int Correlate(output,iw,ix,iw_size,ix_size,no_corr,
              res_shift)
```

Where,

short	output[]	output data (size X-W+1)
short	iw[]	input w
short	ix[]	input x
long	iw_size	size of w (size W)
long	ix_size	size of x (size X)
long	no_corr	number of correlations M to compute
int	res_shift	right shift applied to output

The user can control the number of correlations made. Correlation outside the array can be performed, and zeros will automatically be inserted as required by the function.

The function is mathematically described as follows :-

$$y(m) = \left[\sum_{i=0}^{W-1} w(i)x(i+m) \right].2^{-res\_shift} \qquad 0 \le m < M$$

In order to ensure correct function operation, a number of argument checks are performed. The criteria for a bad argument is as follows :-

iw_size < 1
ix_size < 1
no_corr < 1
ix_size < iw_size
res_shift < 0
res_shift > 27

If one of the above is true, the function will abort and return the flag EDSP_BAD_ARG.

As is the case with most of the DSPLib examples, optimum performance will be obtained if the program and data are located in on chip memory. Clearly the ability to achieve this depends upon the Microcontroller used, and program size. It is recommended that at the very least one of the input arrays be located in on chip RAM, which will help to avoid bus conflicts.

The actual operation of Correlate is shown in an example at the end of this section (5.3). However, in order to clarify the operation of shifting arrays relative to each other during correlation calculation an example follows.

Assume that the data set 1234 is to be correlated with 56789. Clearly, by looking at the argument checking, the second array must be larger than the first.

The process starts with both arrays in line, and then shifts one relative to the other the user specified number of times.

1234	⇐ Held static
56789	⇐ 1st Correlation
56789	⇐ 2nd Correlation
567890	⇐ 3rd Correlation
5678900	⇐ 4th Correlation
56789000	⇐ 5th Correlation

Clearly the definition of the size of the output array requires care. The array will contain one value for each correlation performed, and therefore should be no_corr elements long. Special care should be exercised here since failure to allocate sufficient elements in the output array will result in corruption of the stack due to the nature of the C programming language.

5.2.5 CorrCyclic

CorrCyclic offers the same functionality as Correlate, only the input arrays are shifted in a cyclic manner relative to each other. The function is defined as follows :-

```
int CorrCyclic( output, iw, ix, size, res_shift)
```

Where,

short	output[]	output data
short	iw[]	input w
short	ix[]	input x
long	size	size of input arrays N
int	res_shift	right shift applied to output

The function requires two input arrays of the same size. The two inputs are cyclically correlated and the solution returned in a third array called output. In certain circumstances, the result can be false, in which case Correlate must be used. It is important that the user has a good understanding of the correlation process before using this function. CorrCyclic can be mathematically described as follows :-

$$y(m) = \left[\sum_{i=0}^{N-1} w(i)x(|i+m|_N) \right].2^{-res\_shift} \qquad 0 \le m < N$$

In order to ensure that the function operates correctly, a number of argument checks are performed. The argument checking criteria is as follows :-

size < 1
res_shift < 0
res_shift > 27

If any of the above are true, the function will abort and return the flag EDSP_BAD_ARG. For the purpose of development, it is important to test for this flag. However, once a working system is achieved, the argument checking can be removed in order to speed up execution time.

As is the case with most of the DSPLib examples, optimum performance will be obtained if the program and data are located in on chip memory. Clearly the ability to achieve this depends upon the Microcontroller used, and program size. It is recommended that at the very least one of the input arrays be located in on chip RAM, which will help to avoid bus conflicts.

Since the two data sets are cyclically correlated, the output array should be the same size as the two input arrays. In the example below, the shifts involved in correlating two four element arrays are shown, and it is also shown that the output solution contains four elements.

The data set ‘1234’ is to be cyclically correlated with the data set ‘5678’. The shifts involved in this process are as follows :-

1234 ⇐ Held static
5678 ⇐ 1st correlation
8567 ⇐ 2nd correlation
7856 ⇐ 3rd correlation
6785 ⇐ 4th correlation

The example above should show that the array containing the results is the same size as the input arrays, in this case four elements. It is essential that the user defines an array of suitable size to contain the correlation solutions. Failure to do this will result in the corruption of the stack.

5.3 Correlation Example

By way of an example, this section shows how the correlation function can be used to analyse the random properties of the noise generated by GenGWnoise. Sample C code is given, as well as plots of output data.

Correlation can be used to determine how random a series of white noise samples are. This test uses the DSPLib function GenGWnoise which is explained in the miscellaneous section of this document. The function generates a series of white noise samples which are claimed to be random.

The randomness of the white noise samples can be ascertained by performing auto correlation on the array. This means that the same set of samples are loaded into both input arrays in the Correlate function. The function then performs correlation, and returns a solution in a third array. The plot of this solution can be seen in diagram 5.2 below.

The correlation process will highlight likeness between the input arrays. In our case, if the data is truly random, the level of correlation will be very low. The reason for this is that with random noise, each sample is has an equal probability of being positive or negative. Most samples will therefore cancel each other out, and will thus return a low correlation. The one exception to this is the first correlation, i.e. the first element of the output array. At this instance, each element of the input array is being correlated to itself since no shifting has yet taken place. This means therefore that a very high level of correlation will take place, and a large value returned to the first element. Once the data is shifted by one place, the level of correlation will become very low. This can be seen in Diagram 5.2 over the page, and serves to prove the random properties of the data.

The white noise was generated by GenGWnoise, and is discussed in the next chapter.

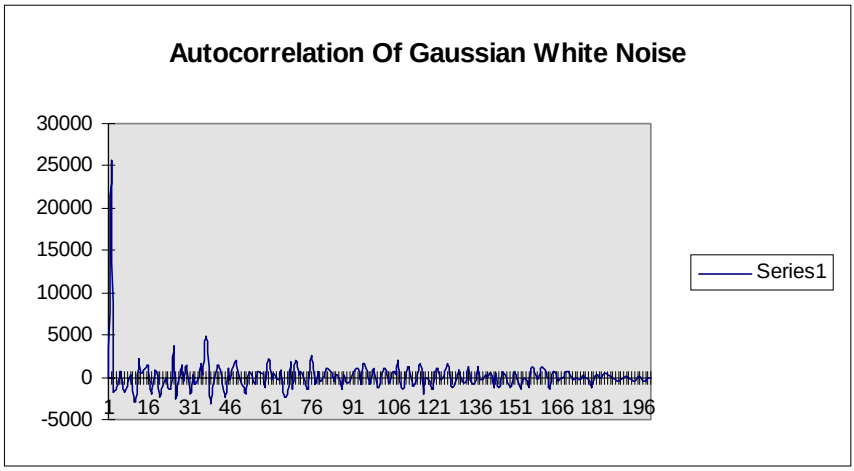


Diagram 5.2

The above plot confirms the random properties of the white noise generated by GenGWnoise. It is clear to see the large amount of correlation when the input arrays were aligned. Once the data was shifted, the correlation remained low indicating the lack of periodicity in the white noise.

C Code Implimentation

```
#include <stdio.h>
#include <math.h>
#include <ensigma.h>
#define NSAM 128

void main(void)
{
       /*** Define Local Variables ***/

       short noisea[NSAM], noiseb[NSAM], result[NSAM];
       int res_shift, a;
       float varience;
       varience = 32767;
       res_shift = 15;

/*** Generate White Noise ***/

       if(GenGWnoise(noisea, NSAM, varience) != EDSP_OK)
       printf("Cannot Generate Noise");

/*** Copy Arrays For Auto Correlation ***/

       for(a = 0; a < NSAM; a++)
       noiseb[a] = noisea[a];

/*** Perform Auto Correlation ***/

       if(Correlate(result, noisea, noiseb, NSAM, NSAM, NSAM, res_shift)!=
       EDSP_OK)
       printf("Problem With Correlate");

}
```

6.0 Miscellaneous Functions

Introduction

This section offers the user a number of useful functions which may be used to 'glue' certain DSP functions together. All the functions are unrelated, and perform small tasks which the user would be expected to require when implementing a number of DSP operations on an SH Microcontroller.

Because of the diverse nature of the functions, software examples have not been provided. It is however hoped that all the functions are reasonably self explanatory, and should cause no problems.

6.1.1 GenGWnoise

This function offers a method of generating true Guassian white noise, of a length specified by the user. The noise generated has zero mean, and a user specified variance. The method of generation is described in detail in the related Ensigma text, but essentially uses the rand operator in C, and arranges pairs of samples to a certain criteria.

The function is defined as follows :-

```
int GenGWnoise( output, no_samples, variance)
```

Where,

short	output[]	array containing white noise samples
long	no_samples	number of output samples required
float	variance	user defined variance of noise distribution σ^2

In order for the function to operate correctly, the user must define no_samples to be one or greater, and also set the variance to a value greater than 0.0. Failure to ensure this will result in the function aborting and returning the flag EDSP_BAD_ARG.

The equations to determine the white noise samples are as follows :-

$$o_1 = \sigma r_1 \sqrt{-2\ln(x)/x}$$
$$o_2 = \sigma r_2 \sqrt{-2\ln(x)/x}$$

The variables r_1 and r_2 represent two numbers generated by the random number generator **rand**, between -1 and 1. These two random numbers are then squared and added together. If the sum is less than one, then they will be used in the above equations. If the sum is greater than one, another pair of random numbers will be generated, and tested in the same manner. This process continues until the user specified number of noise samples have been generated.

Samples are generated in pairs by the equations shown above. If an odd number of samples are required, the second of the two solutions is discarded. It should also be noted that this function isn't suitable for real time operation due to the floating point arithmetic employed.

6.1.2 MatrixMult

MatrixMult offers an optimised method of performing matrix multiplication. Two matrices are multiplied together and the result deposited in a third. The output matrix should not be 'in-place', i.e. should not conflict with either of the input matrices.

The function is defined as follows :-

```
int MatrixMult(op_matrix, ip_matrix1, ip_matrix2,
          no_rows1, no_cols1, no_cols2, res_shift)
```

Where,

void	*op_matrix	pointer to first element of output matrix
void	*ip_matrix1	pointer to first element of input matrix1
void	*ip_matrix2	pointer to first element of input matrix2
long	no_rows1	row dimension of matrix1 (m)
long	no_cols1	column dimension of matrix1 & row dimension of matrix2 (n)
long	no_cols2	column dimension of matrix2 (p)
int	res_shift	right shift applied to each output

The function will fail if any of the dimensions m,n or p are less than one. Also the stated shift applied to the output matrix elements must be greater than zero, and less than twenty eight. If the function fails the flag EDSP_BAD_ARG is returned.

Care must be exercised when specifying array size to contain the array solution. It should be noted that void pointers are used to point to the first element of the matrices. When multiplying two matrices, it is essential that they are conformable. That is the column of the first matrix must be the same size as the row of the second. The dimension of the resulting matrix will be m x p.

Matrices are stored in the standard C manner, in one array. The elements are ordered as follows :-

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{pmatrix}$$

would be represented by the array :-

$$\{a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$$

In order to ensure maximum efficiency, the user should ensure that matrix1 is located in on chip RAM. Clearly the ability to do this depends upon the SH device being used, as well as other memory allocation considerations. Clearly the function will operate correctly with both matrices located off chip, but bus conflicts may arise which will slow down the execution time.

6.1.3 VectorMult

This function offers an optimised routine for multiplying two vectors together. The result of the multiplication is stored in a third array of the same size as the inputs. The output array should be located in a different memory location to both of the inputs, i.e. it should not be in-place. A typical application of this function is to apply a window to a set of data.

The function is defined as follows :-

```
int VectorMult(output,ip1,ip2,no_elements,res_shift)
```

Where,

short	output[]	array containing output
short	ip1[]	input array 1
short	ip2[]	input array 2
long	no_elements	number of elements in each array
int	res_shift	right shift applied to each output

The function will abort if the user specifies the number of elements to be less than one. It will also abort if the right hand shift applied is less than zero or greater than sixteen. If the function aborts, the flag EDSP_BAD_ARG will be returned.

It is important that the two input arrays used with this function are the same size, and that the no_elements argument is set to this size, or less. Clearly, if the user doesn't want to multiply all the array elements together, the no_elements argument is set to a value less than the input array size.

Vector mult can be represented by the following C code. Clearly, if the user wishes to use this type of process, VectorMult should be used since it offers an optimised version compared to that generated by most C compilers. This will be of particular importance in real time applications where execution time is of great importance.

```
short  output[NSAM], ip1[NSAM], ip2[NSAM];
long   NSAM = 128;
int    res_shift = 0, a;

for(a=0; a < NSAM; a++)
output[a] = (ip1[a] * ip2[a]);
```

As can be seen, the function performs element wise multiplication. I.e. it multiplies the first element of ip1 by the first element of ip2 and so on.

If the user requires the calculation of the dot product of two arrays (every element multiplied by every element), the matrix mult should be used with n set to one. For more details on MatrixMult the reader should consult section 6.1.2.

6.1.4 MsPower

MsPower offers a function which will simply calculate the mean square power of an input array presented to it. The result is returned as a long (32-bits) in a single element pointed to by a pointer defined by the user.

The function is defined as follows :-

```
int MsPower( output, input, no_elements)
```

Where,

long	*output	result
short	input[]	input array
long	no_elements	number of elements contained in input array

The function performs simple argument checking, and will return EDSP_BAD_ARG if no_elements is less than one.

The function is mathematically described as follows :-

$$\text{Mean Square Power } = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2$$

As can be seen the function simply returns the mean of the sum of the squares of the elements in the input array. The result of the division is rounded to the nearest integral value. There is a risk of overflow if the number of elements specified is more than 2^{32}-1.

6.1.5 Mean

This function follows directly from that described above. The function calculates the mean of a set of data presented to it in an array. The single element solution is a short and is pointed to by a pointer defined by the user.

The function is defined as follows :-

```
int Mean( mean, input, no_elements)
```

Where,

short	*mean	mean of input data $\bar{x}$
short	input[]	input array x
long	no_elements	number of elements N in x to process

The function performs basic argument checking, and will return the flag EDSP_BAD_ARG if the number of elements N specified is less than one.

The function is mathematically described as follows :-

$$\bar{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

The result of the division is rounded to the nearest integral. If the user specifies a number of elements that is larger than 2^{16}-1 then the possibility of overflow exists.

6.1.6 Variance

This function offers the user a routine to calculate the variance of a set of input data. The user presents the function with an array containing the data to be analysed, and the function will then return both the mean of that data and also the variance.

The function is defined as follows :-

```
int Variance( variance, mean, input, no_elements)
```

Where,

long	*variance	variance of the input array σ^2
short	*mean	mean of the input array $\bar{x}$
short	input[]	input data x
long	no_elements	number of elements N in input array

Basic argument checking is performed, and the function will abort and return the flag EDSP_BAD_ARG if the number of elements specified is less than one.

Mathematically, the function performs two tasks. Firstly it calculates the mean of the data, and makes this solution available to the user. Secondly, the variance is calculated using the solution to mean calculated previous. The equations for mean and variance are as follows :-

$$mean = \bar{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

$$\text{var}\,ience = \sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} (x(i) - \bar{x})^2$$

In each case, the results of the divisions are rounded to the nearest integral value. The result of mean is returned as a short which is 16-bit. Variance solution is returned as a long 32-bit value. There is a possibility of mean overflowing if the number of elements is greater than 2^{16}-1. The variance calculation is not checked for overflow.

6.1.7 MaxI

This function will locate the maximum value in an array presented to it, and will then return a pointer to that value. It does not return the value itself. Maxl is defined as follows :-

```
int Maxl( max_ptr, input, no_elements)
```

Where,

short	**max_ptr	address of pointer to max element
short	input[]	input array to be searched
long	no_elements	number of elements in input to be searched

The function will abort and return the flag EDSP_BAD_ARG if the user specifies less than one element in the input array. The address of the element with the maximum value is returned in max_ptr. If two elements or more have the same maximum value, the address of the element nearest to the start of the array will be returned.

6.1.8 MinI

Minl compliments the above function in that it returns the address of the lowest element in the input array presented to the function. The actual minimum value is not returned, but instead a pointer to it. The function is defined as follows :-

```
int Minl( min_ptr, input, no_elements)
```

Where,

short	**min_ptr	address of pointer to min value
short	input[]	input array to be searched
long	no_elements	number of elements to be processed

The function will return the flag EDSP_BAD_ARG if the user specifies a number of elements that is less than one.

As with the above function Maxl, if there are two or more min elements the function will return a pointer to the element that is closest to the start of the array.

Timings for these functions are impossible to quote since the total execution time depends upon the number of elements in the array being searched. One method of further improving the execution speed is to remove the argument checking, as long as the number of elements in the input array is greater than one, this will not cause and problem, nor compromise the reliability of the function.

It should be noted that removal of argument checking is only recommended on simple functions such as this one. For more complicated functions, the argument checking will provide an effective tool for debugging processes.

6.1.9 Peakl

This final function is simply a variation on Maxl. The difference with this function is that it will return the address of the maximum absolute value in the input array. I.e. it looks for the largest value irrespective of sign. As an example in an array of two elements {30,-50} Peakl would return a pointer to the -50 element.

The function is defined as follows :-

```
int Peakl( peak_ptr, input, no_elements)
```

Where,

short	**peak_ptr	address of pointer to the peak element
short	input[]	input array to be searched
long	no_elements	number of elements in input to be processed

The function performs basic argument checking and will return the flag EDSP_BAD_ARG if the user specifies the number of elements to be less than one. The process can be speeded up by removing the argument checking, but the reader should first read the note at the base of the previous page.

As with the previous two functions, if there are two elements or more that have the same absolute peak value, the function will return the address of the element that is closest to the start of the array.

7.0 DSPLib Implementation

Introduction

This final section of the application notes sets out to give a typical example of how a working system based around the SH1 processor can be implemented. By the very nature of DSP, it is very unlikely that this example will exactly suit the needs of the reader due to the great diversity of digital signal processing. However, the aim of this section is to give the reader some ideas about hardware set-up, as well as programming around the functions offered by DSPLib. Furthermore, it is hopped that consultation of this section will speed up the design process, and help to avert possible time consuming problems.

Implementation can be split into two distinct areas, hardware and software. The hardware design considerations are considered first, followed by some typical software examples to tie in. Clearly, any software examples given will be fairly hardware dependant, but should help the reader to understand the process of interfacing DSPLib with the real world.

7.1 Hardware

The hardware implementation of a DSP process is greatly dependant upon the process it is intended for. Having said that, every DSP system can be broken down and shown to contain at least two if not all of these three distinct sections :-

Analogue To Digital Conversion

Digital Processing

Digital To Analogue Conversion

A system containing two of the above sections is the compact disc player for example. The data is already in digital format, and so requires no analogue to digital conversion. Conversely, a mobile telephone contains all three, with analogue data coming from the microphone, and also being sent to the speaker.

The hardware example given in the following text contains all three sections, and is capable of performing digital processing on real analogue signals, and returning the solutions as real analogue signals.

It is important for the reader to remember that the data conversion processes are not simply digital to analogue or analogue to digital converters, but instead require the careful consideration of the use of sampling rates, and also filtering. In the data conversion process, the use of anti-aliasing and anti-imaging filters are required. Since it is the purpose of this document to show how the Hitachi SH-1 processor can be interfaced, notes on filter design are not included.

The filters used in a typical application take the form of a low pass network with a Butterworth response. The filter order chosen depends upon the acceptable level of aliasing/imaging but would typically be around 6th order implementations.

7.1.1 Analogue to Digital Conversion

This process consists of three distinct blocks, and is shown below. The real world analogue signal is firstly bandlimited by the anti-aliasing filter. The purpose of this filter is to prevent signals that are near or above the Nyquist frequency (twice the sampling frequency) passing into the sampling device. As discussed above, the filter would normally take the form of a low pass network with Butterworth response.

The band limited signal is passed into the analogue to digital converter. In the case of this example, the on chip ADC has been used, which produces a ten bit representation of the analogue data. The ADC is controlled by a sample rate generator. The sample rate dictates the ADC conversion process, and should be chosen to be at least twice the highest frequency component present in the input.

In this particular example, the bandwidth of the input signal is 3.4Khz, and the sampling rate was chosen to be 8Khz. This system should in part help to mimic telephone based systems which have similar specs.

There are many methods open to the user for sample rate generation. In order to gain maximum efficiency, it is suggested that the sample rate be synchronised to the CPU clock. For the purposes of this example, the CPU clock has been divided by external hardware dividers, eventually arriving at a frequency of 7.8Khz, which is an acceptable rate for the sampling of the given signal bandwidth.

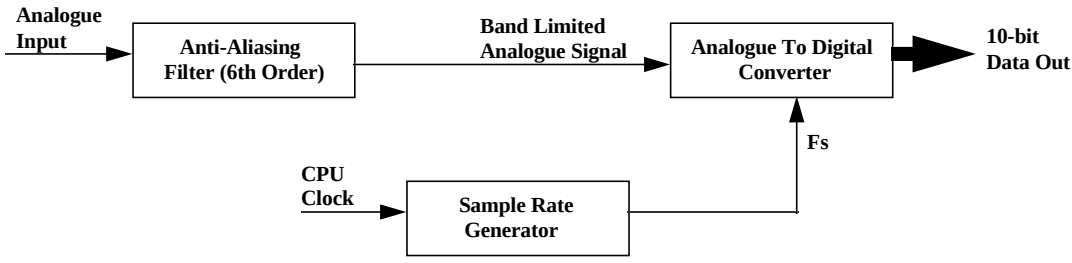


Diagram 7.1 :- **Analogue Interface**

As mentioned earlier, the Hitachi SH-1 processor contains an on chip ADC. This device gives ten bit resolution, and helps to simplify the implementation of a real time system. Clearly if the user requires a higher resolution, an external ADC should be used. The device could be memory mapped, and accessed in the same manner as RAM. Notes on using the internal ADC are included in the software section of this document. For further information on the SH-1 peripherals, the reader should consult the SH7032/SH7034 hardware manual.

Once the data is converted into digital form, it is suitable to present to the DSPLib routines. The actual interface software used to load data in, and also write data out at a rate of 8Khz is described in the software section.

7.1.2 Digital To Analogue Conversion

Once the digital processing is complete, it is necessary to convert the descrete time domain samples returned by DSPLib into a continuous analogue waveform. This process involves two stages, and is rather obviously the reverse to the analogue conversion process described above.

Because the SH-1 series of microcontrollers don't have on chip DACs, the user must interface a device to it. The actual specific details of how this can be achieved are included in the circuit description section of this report.

Keeping to a more general form, the best location for a DAC on the SH series of devices is on the data bus. By locating the DAC on the memory map, the device can be accessed exactly like a piece of RAM. The access time to the DAC can be controlled by the insertion of wait states, and these are discussed fully in the software description section at the end of this report.

The use of serial data is very popular in many DSP systems today. The SH-1 has two bi-directional serial ports which could be used to interface to serial DACs as well as ADCs. It should however be noted that the respective converters would most probably not interface directly to the serial port, and would infact require some 'glue' logic. The main reason for this is that the serial protocol used by the SH series would probably not be suitable for the converter being connected. The use of some 'glue' logic would enable the associated start and stop bits to be removed/modified inorder to satisfy the chosen serial device.

When converting data from digital to analogue, it is essential that the resulting waveform is filtered in order to remove most imaging frequencies. The imaging frequencies are generated as a result of the steps that are produced by the DAC. These steps are referred to as quantisation, and reflect the accuracy of the converter. As the converter can only output one of say 1024 levels (10-bit), the waveform is stepped in between. The sharp edges of the steps create many harmonics which roll off with a sinex/x function in the frequency domain.

The purpose of the anti-imaging filter is to allow frequencies only in the bandwidth of interest to be passed, thus stripping the signal of the unwanted quantisation steps. It is specific to the user to specify the response of the filter. The filter is normally low pass, and exhibits Butterworth characteristics. In the case of the example system described in this section, a 6th order filter was chosen, with a 3dB bandwidth of 3.4Khz.

For both the ADC and DAC process, use was made of a virtual ground. The purpose of this process is to make the signal presented to the ADC purely positive. The DC offset introduced was then removed by software.

Once the DSP operation is complete, the DC offset is re-introduced, and the purely positive output signal then sent to the DAC. The output of the DAC is de-coupled in order to remove the signals DC content. By using this method, a single quadrant converter can be used at both ends, thus helping to keep system cost to a minimum at the expense of only a few clock cycles.

7.1.3 Circuit Description

The reader should consult the enclosed schematic of the example system hardware when reading the following description, which can be found at the end of this section.

The input signal is applied on the ADC_In line on the left of the schematic. This signal has been band limited to 0 - 3.4Khz in order to prevent aliasing occurring.

Once applied, the signal is de-coupled by C1 which has the effect of removing any DC component present in the signal. Next the input is passed through a unity gain buffer. This stage is included to isolate the filter stage from the ADC process. It is important to ensure that the final stage of the anti-aliasing filter is not loaded in any way, which could in turn cause the filter response to be altered. By using a buffer, the final stage of the filter will 'see' a high impedance, thus removing any risk of loading. The output of the buffer is fed back to the inverting input, which forces the amplifier to act as a voltage follower. Once buffered, the signal is once again decoupled by C2, which removes any DC introduced by the amplifier.

The next stage concerns the introduction of a 2.5 volt DC offset or 'virtual ground'. The purpose of this process is to ensure that the signal applied to the ADC is purely positive. Up to this point, the input signal has been around five volts peak to peak, centred around zero. Now, a DC offset is added to the signal, which centres the signal around 2.5 volts, shifting the whole signal positive. This will now ensure that the whole signal is sampled with no loss of information. After sampling, the DC offset is removed using software techniques. This process is described in the software section of this document.

The DC offset is generated by a dedicated virtual ground generator device U8. There are a number of these devices on the market, and any three pin type should be suitable. It is important to remember that the reference should be as noise free as possible, as noise here is introduced into the signal. The use of a bandgap type generator should ensure low noise operation. It is important that a zener device is not used as these are very noisy and will degrade the systems signal to noise ratio.

Prior to application to the ADC, the signal is passed through an over voltage protection circuit provided by D1 and D2. If the input is between 0 and 5 volts, the diodes have no effect on the signal. However, if the voltage should rise above the supply rail, the diode will have the effect of clamping the signal to the supply. This is also true if the signal should fall lower than zero, diode D1 will clamp the input to ground. This arrangement is included to offer some protection over the SH-1 on chip ADC.

After the input protection, the signal is in suitable form to be applied to the SH-1 ADC. This ADC has eight multiplexed inputs, and in the case of this example, input AN0 has been chosen. For more information on the SH-1 hardware configuration, including the ADC, the reader should consult the SH7032/SH7034 Hardware Manual.

The SH-1 ADC can be configured such that it is triggered externally, as is the case in this example. The trigger is applied to the ADTRG line located on pin 63. The trigger is a square wave with 50% duty cycle, and conversion starts on each falling edge. The trigger signal is derived from the 16Mhz CPU clock which is available from the CK line on pin 71. This output is derived from the system crystal, and has been cleaned up and passed through a duty cycle correction circuit.

The 16Mhz signal is repeatedly divided by three four bit ripple counters U4, U5 and U6. The end effect is 16Mhz / 2048 = 7.8Khz. The resultant signal from U6 is used to start the ADC. An alternative method for sample rate generation is to program the SH-1 timing pattern generator. Details on this subject may be found in the SH-1 Hardware Manual.

The final section of hardware to explain is the digital to analogue conversion. This is achieved by way of a memory mapped 12-bit device U2. Although a twelve bit device has been chosen for this example, the output resolution is only really 10-bit due to the 10-bit ADC used. If the reader wanted a 16-bit system, the method of connection is identical, only the device would use the first sixteen data lines on the bus instead of the first twelve in this case.

Because the device is memory mapped, a small amount of address decoding logic is required. The SH-1 has most of the address decoding provided on chip, and the memory map for the device can be seen in its associated hardware manual. In the case of this example, the DAC is located in area 6 which starts at address E000000 and ends at EFFFFFF. When an address within that area is accessed, the line CS6 falls low. In the case of a write operation, which is what happens with the DAC, the WR pin falls low as well. U3 controls the chip select on the DAC, by ensuring that the CS pin on the DAC falls low when both the CS6 line and the WR line are low. By the nature of the real time operation of the system, the CS pin on the DAC should fall low at the same rate as the ADTRG pin, in this case 7.8Khz. The width of the write cycle can be controlled by the SH-1 wait state controller. Correct programming of this is important to ensure that the write cycle is long enough to be seen by the DAC. In the case of the AD668 device shown, a write cycle of length 125nS was used, which equates to two CPU clock cycles.

The DAC used in this example is an Analog Devices AD668. This device is suited to direct connection to CPU buses due to its data latch arrangement. It is important that the power supply arrangements are noted, a split rail supply of 16 volts is required. For more details on this device, the appropriate data sheet should be consulted.

The output of the DAC is 0 - 5volts and will contain a large amount of quantisation noise. The removal of the DC offset can be achieved by de-coupling the signal, before passing it to a suitable Butterworth low pass filter arrangement.

7.2 Software

This section describes the relatively straight forward software routines required to pass real time data through the system outlined in the earlier hardware description.

The routines are specific to the hardware configuration detailed, but are written such that modification for any application should be quite simple.

The main program is written in ANSI C, and in turn calls a number of functions. For optimum performance, these functions have been written in assembler. The assembler format shown in the examples below is SH series, and will work with the relevant Hitachi tools. It should be noted that the code will not assemble with the GNU tools that are available for the SH series. In order to achieve this, the format of the code will have to be changed as appropriate.

7.2.1 Main()

This is the main program which as described calls the other routines in turn. The code can be seen below.

```
#include <stdio.h>
#include <math.h>
#include "coeff.dat"
#include <ensigdsp.h>

#define NSAM    1
#define no_sections 1

void system(void);
void dac(short*);
void adc(short*);
void detect(short*);

void main(void)
{
       short *pointer;
       short sample;  /*  Data From ADC */
       short output;  /* Data For DAC */
       short *result; /* Start DSP operation Flag */
       short *work;
       int end = 0;

       pointer = &sample;
       result = &output;


       system(); /* Initialise System (Wait State = 1)*/
```

```
/** Filter workspace **/

       if(InitIir(&work, no_sections) != EDSP_OK)
       printf("Problems .....");

/** Set Up ADC And DMAC **/

       adc(pointer);

/** Real Time Loop **/

       do
       {

       detect(pointer);    /**  Wait For End Of DMA     **/

       Iir1(result,sample,coeff,no_sections,work)

       dac(result);  /*Output Data To Memory Mapped DAC */

       }
       while(end == 0);
}
```

Following the above C code, the operation is as follows :-

The code implements an IIR filter using the function Iir1 from DSPLib. The first part of the code is the include statements, and as well as the expected header files, the ensidsp.h is included. This header file contains all the function definitions for DSPLib, as well as some frequently used constants. At this point, the coefficients for the IIR filter are included as well in the file coeff.dat.

After some function prototypes and constant declarations, the code starts proper. The first function call is to system(). This function is not part of DSPLib, but is instead a small section of code which sets up the hardware on the SH-1. This function is described in detail in section 7.2.2.

Following on from system(), the IIR filter workspace is initialised. This function is described in section 4.2.3. In order for this function to operate correctly, it is essential that the memory sections are defined properly in the linker command file. For further information on the Hitachi linker, the reader should consult the H series linkage editor users manual.

Next in the code, the on-chip ADC is initialised. This function adc() is described in section 7.2.3, and requires a pointer which is used to point to the sampled value. The function makes use of the DMA (Direct Memory Access) controller. When the ADC has completed its conversion, the result is moved into a memory location pointed to by the pointer provided by the user in one clock cycle.

Now we enter the real time loop. Until now, the code has been set-up, and only executed once. Inside this loop, the processor performs sample by sample computation. The first function we meet is called detect(). This function forces the processor to wait for the end of the DMA operation before continuing the operation.

This ensures that the DSP operation is only performed on a new sample. The function is described in section 7.2.4.

Once the end of the DMA operation has been detected, the DMA controller is reset, and the DSP operation can take place on the single sample received. In order to achieve real time operation, it is essential that the DSP task ends before a new sample arrives.

When the DSP task is complete, the function dac() is used to output the result to the memory mapped DAC described in the hardware section of this document. The function dac() is described in detail in section 7.2.5.

7.2.2 system()

The H series assembler for this function is as follows :-

```
_system:

;Initalise Wait States For DAC
;Set Up Pin Function Controller

       MOV.L   PACR1,R3
       MOV.L   SETUP,R2  ;Enable ADTRG Pin
       RTS
       MOV.W   R2,@R3

       MOV.L   WCR3,R3
       MOV.L   WAIT,R2  ;One Wait State
       RTS
       MOV.W   R2,@R3

       .ALIGN 4

WCR3   .data.l     H'05FFFFA6
PACR1  .data.l     H'05FFFFC8
SETUP  .data.l     H'0000330A
WAIT   .data.l     H'00008800

       .END
```

This function performs two tasks. The first task is to set up the pin function controller on the SH-1 such that the ADTRG pin is selected. This will allow the on chip ADC to be externally triggered via that pin. This process is achieved by setting up the PACR1 register.

Finally, the function sets up the wait state controller such that the write cycle to the external DAC is of a sufficient period. In this particular example, the write cycle is two CPU clock cycles long. The register WCR3 was used to control this task. Details on both the registers mentioned here can be found in the SH-1 hardware manual.

7.2.3 adc(short*)

The SH assembler listing for this function is as follows :-

```
_adc:


;Set Up DMAC

       MOV.L   SAR0,R1
       MOV.L   ADDRA,R2
       MOV.L   R2,@R1
       MOV.L   DAR0,R1
       MOV     R4,R2
       MOV.L   R2,@R1
       MOV.L   TCR0,R1
       MOV     #H'01,R2
       MOV.W   R2,@R1
       MOV.L   CHCR0,R1
       MOV.L   DATA,R2
       MOV.w   R2,@R1

;Start DMAC

       MOV.L   DMAOR,R1
       MOV     #H'01,R2
       MOV.W   R2,@R1


;Set Up ADC

       MOV.L   ADCSR,R1
       MOV     #H'048,R2
       MOV.B   R2,@R1
       MOV.L   ADCR,R1
       MOV     #H'0FF,R2
       RTS
       MOV.B   R2,@R1

;Register Vectors

      .ALIGN 4
DATA  .data.l   H'0000D09
ADDRA .data.l   H'05FFFEE0
ADCSR .data.l   H'05FFFEE8
ADCR  .data.l   H'05FFFEE9
SAR0  .data.l   H'05FFFF40
DAR0  .data.l   H'05FFFF44
TCR0  .data.l   H'05FFFF4A
CHCR0 .data.l   H'05FFFF4E
DMAOR .data.l   H'05FFFF48

       .END
```

This function performs several tasks. When calling the function, the user provides a pointer to a variable. This pointer is used by the function to set up the DMA controller such that the ADC value is transferred to it in one clock cycle. The setting of the DMA is quite intricate.

It is therefore recommended that the reader consult the SH-1 hardware manual DMA section. The registers seen above can be followed, and set up to suit a particular application.

When the DMA set up is complete, the DMA controller is started. This now means that when the ADC generates an interrupt at the end of a conversion, the DMA will transfer the data to the destination address in one clock cycle.

The final part of this code sets up the internal ADC. The options chosen include an external trigger option and also an interrupt generation which in turn starts the DMA controller. From now onwards in this code, the ADC will always start converting when the external trigger pin falls low. After each DMA operation, some registers need resetting. This task is performed by the detect() function described in the next section.

7.2.4 detect(short*)

The detect routine forces the process to wait until the input data is valid, i.e. a new sample. The assembly listing of this function is as follows :-

```
_detect:
          CLRT
          MOV.L   CHCR0,R1
LOOP:   MOV.W   @R1,R2
          EXTU.W  R2,R0
          TST     #H'02,R0
;Test bit goes to one if AND is zero
          BT      LOOP



;Convert Sampled Data pointed to by R4
       MOV.W   @R4,R1
       EXTU.W  R1,R2
       SHLR    R2
       SHLR    R2
       SHLR    R2
       SHLR    R2
       ADD     #H'03,R2
       MOV.L   DATA,R1
       SUB     R1,R2
       MOV.W   R2,@R4

;Start DMA And ADC Again

;Reset TCR0 & CHCR0 & DMAOR

       MOV.L   CHCR0,R1
       MOV.L   DATA1,R2
       MOV.W   R2,@R1
       MOV.L   TCR0,R1
       MOV     #H'01,R2
       MOV.W   R2,@R1
       MOV.L   DMAOR,R1
       MOV     #H'01,R2
       MOV.W   R2,@R1
```

```
;Reset ADCSR

       MOV.L   ADCSR,R1
       MOV     #H'048,R2
       RTS
       MOV.B   R2,@R1

       .ALIGN 4

DAC     .data.l H'0E000000
TCR0    .data.l H'05FFFF4A
CHCR0   .data.l H'05FFFF4E
DMAOR   .data.l H'05FFFF48
DATA1   .data.l H'00000D09
ADCSR   .data.l H'05FFFEE8
DATA    .data.l H'000008A3

       .END
```

The first part of this code consists of a loop which tests for the end of a DMAC operation. The program will stay in that loop until the DMA has finished, then come out and execute the remaining code. The end of a DMA means that a new sample is ready for processing. By making the CPU wait for a new sample, the risk of processing the same sample twice is removed.

Next, the new sample is converted into the correct form ready for presentation to the DSP task. The sample from the on chip ADC sits in the upper 10 bits of a sixteen bit word. The ten bits are therefore shifted right by four places and the two least significant bits set to one. The result of this process is that the data is now in 12 bit form suitable for the DAC. Next, the DC offset introduced by the hardware is removed by a single cycle subtraction. If the chosen DSP task is operating correctly, it should return a 12 bit solution which can have an offset applied to it, and then sent to the DAC. That process is described in section 7.2.5.

Once the data is in suitable form, the DMA controller is reset and started again. The transfer count register is reset, and the controller is started again. This means that a DMA operation can take place during the DSP task, and the new sample is ready for processing immediately.

Finally, the ADC is started again by writing to the ADCSR register. For full details on the setting and controlling of SH-1 peripherals, the SH-1 hardware manual should be consulted.

7.2.5 dac(short*)

This final routine simply takes the solution returned by the DSP process, and writes it to the memory mapped DAC. The SH assembly listing for this function is as follows :-

```
_dac:
;Output Value To DAC

       MOV.L   DAC,R3
       MOV.W   @R4,R2
       MOV.L   OFFSET,R1
       ADD     R1,R2
       RTS
       MOV.W   R2,@R3

       .ALIGN 4

DAC     .data.l H'0E000000
OFFSET  .data.l H'00000800

       .END
```

As can be seen, the DAC is located at address E000000. This location is in area 6, and ties in with the hardware description earlier. The value returned by the DSP process has a DC offset added to it. The offset is hex 800, which corresponds to half of the twelve bit output. Therefore, with this offset, the maximum output swing is achievable. Half of full scale corresponds to 2.5volts, which as the hardware section will confirm, is the same as the offset applied to the input.

Once the offset is applied the value is written to the DAC as if it were a piece of memory. The write process has already been set-up in the system() function described in 7.2.2. When the value has been written, the program counter returns to detect() and waits until a valid sample is in memory, and then the process starts again.

7.3 Timing

When writing code for a real time process it is essential that the operation stays within the allotted time space. For the system described in this section, the processing time can be calculated as follows.

CPU	=	16MHz
Sample Rate	=	7.8 KHz

Between each sample there is 128uS. This time corresponds to 2048 CPU clock cycles. From this total, the IO operations steal cycles as does the conversion of data.

For optimal execution speed DO NOT compile the ANSI C functions included in DSPLib. These are included for reference only, and will not lead to an efficient implementation of a given DSP task. The reader should use the optimised source code in the optsrc directory in DSPLib, or alternatively simply use the ensigma.lib during the link operation, which is in turn built from the optimised code.

Appendix One

Ensigma Software Guide For DSPLib

DSPLib for SH
User Guide

E. Andrews

Sept 15, 1995

Disclaimer

When using this document, keep the following in mind,

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during the operation of the user's unit according to this document.
4. Circuitry and other examples described herein are meant only to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples therein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. MEDICAL APPLICATIONS: Hitachi's products are not authorised for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi's sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant sales office when planning to use the products in MEDICAL APPLICATIONS.

Contents

1 Introduction

This document describes **DSPLib-SH**, a library of signal processing routines developed for use with Hitachi's SH RISC processors.

Two versions of the library are provided. In the first version all of the time-critical routines are C-callable, but have been written in SH assembler to minimise execution time whilst maintaining accuracy.

The second version of the library contains no assembly code. All routines are written in portable ANSI-C and aim to give results that are bit identical to the optimised assembly library. These libraries allow a development methodology beginning with simulation and program development on a workstation using the ANSI-C library. Development will then transfer to the target hardware, using the optimised version of the library, where hardware dependencies and real-time issues can be resolved.

This methodology enables designers to gain the benefits of using efficient hand-coded software for the time-critical sections of their programs, whilst retaining the benefits of developing software written in 'C'.

Five groups of routines are included in **DSPLib-SH**—

- Fast Fourier Transforms
- Windowing Functions
- Filters
- Convolution and Correlation
- Miscellaneous

The routines are all re-entrant. That is, they do not use any local static variables. This enables them to be used in a multi-tasking environment, or to be called from interrupt routines without affecting their use in the main program.

Timings are given for the most time-critical routines. The timings are for a 10 MIPS SH-1, with all internal program and data accesses taking a single

processor cycle over the 32 bit memory bus. Timings for other clock speeds can be derived by scaling those given here.

The routines provided in this library have been optimised for the SH-1 processor. SH-2 and SH-3 processors will also benefit from **DSPLib-SH**, but there will be some differences in execution times compared to the SH-1.

2 Installation

DSPLib-SH is shipped on a single $3\frac{1}{2}$" MS-DOS disk, the contents of which are listed in Appendix A. It comprises a C header file, `ensigma.h`, the library routines themselves and examples of the library's use. The optimised library is in the `library` directory, in the pre-assembled object library `ensigma.lib`. The source for the optimised and ansi-C libraries are supplied in the `optsrc` and `ansisrc` Examples showing the use of each routine are provided in the `examples` directory on the disk.

The header file and libraries should be installed in a directory visible to the C compiler and linker respectively. It is recommended that the same directories as the compiler support files is used. Assuming that this directory is `C:\shc\shc2.0` and the floppy disk drive is `A:`, the installation procedure on an MSDOS PC is as follows—

1. Make a backup of the distribution disk.
2. Copy **DSPLib-SH** onto the system disk. Put the distribution disk into drive `A:` and enter the following:

   ```
   XCOPY A:\* C:\shc\shc2.0 /S
   ```

3. Installation is now complete.

Should it be necessary to rebuild the optimised library from source—

1. Change to the `optsrc` directory.
2. Edit `make.bat` and `lbr.sub` to check that the configuration matches your system.
3. Type `make`.
4. `ensigma.lib` has now been rebuilt.

For the C version of the library, the build process is both operating system and compiler dependent. For a Unix operating system, the procedure is as follows—

1. Copy **DSPLib-SH** onto the system disk.
2. Change to the `ansisrc` directory.
3. Edit `makefile` to correspond to your system.
4. Type `make`.
5. `ensigmaC.a` has now been built.

3 Data Formats

The majority of calculations performed in **DSPLib-SH** use fixed point arithmetic. Both integer and fractional number representations are used; these formats are discussed below.

Integer arithmetic represents numbers with the decimal point fixed at the right of the LSB (least significant bit). A signed 16 bit integer variable can represent values in the range [-2^{15}:2^{15}–1]. When two signed 16 bit integers are multiplied, the signed 31 bit product is aligned at the LSB. If the result is stored in a 16 bit integer numerical overflow may occur.

Fractional arithmetic is also useful, especially in DSP applications. Here the decimal point is fixed immediately to the right of the most significant bit (MSB) for signed numbers (or immediately to the left of the MSB for unsigned numbers). Signed 16 bit fractions can represent values in the range [-1:1-2^{-15}]. When two signed 16 bit fractions are multiplied the signed 31 bit product is aligned at the MSB. If the result is stored as a 16 bit fraction numerical underflow may occur.

DSPLib-SH makes extensive use of both integer and fractional arithmetic. To maximise the flexibility of the library many of the routines allow the alignment of each output to be specified. For example, calling Fir with *res_shift* = 0 corresponds to LSB aligned (integer) filter coefficients. When *res_shift* = 15 the processing corresponds to MSB aligned (fractional) filter coefficients.

For most possible values of *res_shift*, there is a possibility of numerical overflow. Except where stated otherwise, **DSPLib-SH** detects this condition and saturates the value to the appropriate full-scale value. This minimises the error introduced, at the expense of a small increase in processing time when overflow occurs.

Implementation

The C `short` and `int` types provide fixed point arithmetic. On the SH, signed 16 bit integers correspond to `short` and signed 32 bit integers correspond to `long` or `int`.

4 ANSI C Library

An ANSI-C version of **DSPLib-SH** is provided in `A:\dsplibsh\ansisrc`. This version is supplied to ease application development, allowing the early stages to be performed without the constraints imposed by particular hardware.

The C library and optimised SH-1 library give bit identical results for most valid inputs, for all routines except the FFT and double precision filter routines, where errors in the least significant bits may occur.

The exception to this rule occurs when the 42 bit accumulator (MACH/MACL) overflows. In this case the assembler library on an SH-1, the assembler library on an SH-2 and the C library on a host can all give different results. However, this can only happen with input vectors containing more than 1023 samples, so it is extremely unlikely in any practical use of **DSPLib-SH**.

Compiler Restrictions

The C library must be build with an ANSI C compiler, with the restriction that variables of type `double` allow the exact representation of 42 bit integers, and the pointer representation must require four or fewer bytes.

In practise all Unix and MSDOS ANSI C compilers are suitable as they use 8 bytes to hold double variables, typically allowing the exact representation of integers of up to 53 bits.

5 Efficiency

The routines provided by **DSPLib-SH** have been hand optimised for maximum speed on SH-1 processors. In many cases the library routines approach the theoretical optimum performance of the architecture, with minimum instruction counts and minimal pipeline conflicts.

As SH is a RISC architecture, it is relatively straightforward to maximise the efficiency of the library based on the following two features: The assembly routines provided by **DSPLib-SH** have been written so that most program fetches do not cause processor stalls, as long as the program memory supports 32 bit reads; furthermore, most of the processing in **DSPLib-SH** operates on 16 bit data.

Therefore, when defining the memory map of a target system, the following two recommendations should be obeyed whenever possible:

- the program code segment should be located in memory that supports single cycle 32 bit reads, and
- the data segment should be located in memory that supports single cycle 16 (or 32) bit reads and writes.

If there is sufficient internal 32 bit memory, this would be a suitable location for the library code and data. If other memory must be used the recommendations above should be followed whenever possible — because different devices in the SH family contain different types of memory and support different caching schemes, it is not possible to provide completely general advice.

6 Test and Example Programs

Test programs for all library routines are provided in the `examples` directory on the distribution disk. These carefully test each library routine in turn. Among other things they test for correct argument checking and perform a number of bit-exact tests comparing the results with precalculated data. These tests also illustrate the use of the library routines.

The test functions are supplied in the subdirectory `test`. Test data is in the `testdat` subdirectory. The main test files are—

`testfft.c`	FFT test functions.
`testwin.c`	Window test functions.
`testfilt.c`	Filter test functions.
`testconv.c`	Convolution and correlation test functions.
`testmisc.c`	Miscellaneous test functions.

These are built using the make utility. Typing `make all` builds `testfft.abs` and `testfft.mot` etc. Alternatively individual test functions may be made by typing eg. `make testconv.abs`.

In addition to the individual test files and function library, the following files are provided—

`testhead.h`	Prototypes for test functions.
`link.cmd`	Linker subcommand file based on the SH7030 memory map.
`c0.src`	C startup file.
`make.bat`	MSDOS build commands.
`makefile`	(Unix) makefile.

A simple demonstration program that illustrates the use of common library functions is provided in the directory `demo`. This illustrates the use of fft, window, filter and miscellaneous functions.

The demonstration program is built using the following files:

`demo.c`	The demonstration program.
`demoio.c`	Data input/output functions. These functions simulate the data interfaces that would be present in a complete system.
`demoio.h`	Function prototypes for `demoio.c`.
`link.cmd`	Linker subcommand file based on the SH7030 memory map.
`c0.src`	C startup file.
`make.bat`	MSDOS build commands.
`makefile`	(Unix) makefile.

7 Fast Fourier Transforms

Contents—

These functions calculate the classical forward and inverse Fourier transforms, with a user defined scaling. The forward transform is defined by

$$y_n = 2^{-s} \sum_{n=0}^{N} e^{-2j\pi n/N} . x_n$$

where s (shift) is the number of stages where an additional halving is performed (see the *Scaling* section below) and N is the number of complex points.

The inverse transform is defined by

$$y_n = 2^{-s} \sum_{n=0}^{N} e^{2j\pi n/N} . x_n$$

Routine Timings

The execution time in milliseconds of the FFT routines on a 10 MIPS SH-1 are—

	Not in place†				In place			
	FFT		Inverse FFT		FFT		Inverse FFT	
Size	Cmplx	Real	Cmplx	Real	Cmplx	Real	Cmplx	Real
128	1.94	1.06	2.19	1.20	1.94	1.06	2.20	1.19
256	4.43	2.37	4.94	2.66	4.46	2.37	4.97	2.64
512	9.98	5.29	11.00	5.88	10.03	5.32	11.06	5.86
1024	22.23	11.70	24.28	12.88	22.39	11.75	24.44	12.83

Complex Array Format

Time and frequency series of complex samples are used as inputs and outputs to the Fourier transform routines. The complex array format used in all these routines is alternating real-imaginary i.e. a sequence of N complex numbers c is stored in an array x—

$$x[2n] = \mathrm{Re}\{c(n)\}, \quad x[2n+1] = \mathrm{Im}\{c(n)\} \qquad 0 \le n < N$$

There is one exception to this format, when the frequency representation of a real signal of N samples is stored in a complex array of N coefficients. In this case element $x[0]$ contains the real component of the DC component of the signal and $x[1]$ contains the real component of the $F_s/2$ frequency (both DC and $F_s/2$ components are real, their imaginary components are zero).

Scaling

In an FFT the signal power doubles at each natural radix-2 stage; the peak signal amplitude can also double. This doubling can cause arithmetic overflow when transforming a high power signal, but can be prevented by a division by two at each radix-2 stage. However, excessive halving of the signal will add unnecessary quantisation noise.

The optimum balance between overflow and quantisation noise is highly dependent on the characteristics of the input signal. A high power pure tone, for example, will require the maximum scaling to avoid overflow, whereas an impulsive signal will require very little.

The safest approach is to halve the signal amplitude at every radix-2 stage. As long as each complex input samples is scaled to have power less than 2^{30} this approach guarantees that overflow will not occur. At the other extreme, a forward FFT of a low amplitude signal with a flat spectrum could be performed with no halving without overflow occurring.

A middle path is to halve the input signal on alternate stages. **DSPLib-SH** allows this approach, and also allows finer control of scaling, with halving selectable individually for each radix-2 stage. Careful selection of this scaling allows the combined effects of saturation and quantisation to be minimised.

It should be emphasised that the conservative approach of checking the input sample power and halving at each stage should always be used unless you are confident of the properties of the input signal.

To specify the required approach each FFT function has a *scale* parameter. *scale* is interpreted least significant bit first, with one bit corresponding to each radix-2 stage. A division by two is performed at all stages for which the corresponding *scale* bit has been set.

For example,

- *scale* = `0XFFFFFFFF` (or *size*-1) specifies halving on every stage, giving a final output with $1/size^2$ times the power of the input, with a guarantee that no overflow has occurred if the initial complex samples all have power less than 2^{30}.
- *scale* = `0X55555555` specifies halving on alternate stages, giving a final output with the same power as the input (if there are an even number of stages).
- *scale* = `0` specifies no halving, giving a final output with *scale* times the power of the input,

`EFFTALLSCALE`, `EFFTMIDSCALE` and `EFFTNOSCALE`, defined in ensigma.h, can be used to provide these constants.

Test Functions

The test file `testfft.c` is supplied for the routines in this section. The functions TestFft and TestIfft test the FFT and IFFT transforms respectively.

FftComplex

Definition :

`int` FftComplex(*output, input, size, scale*)

Arguments :

`short`	*output[ ]*	complex output data.
`short`	*input[ ]*	complex input data.
`long`	*size*	size of FFT.
`long`	*scale*	scaling specification.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 4.
	size not a power of 2.
	size > 32768.

Description :

This routine calculates a complex Fast Fourier Transform. The calculation is not-in-place, so the input and output arrays must not overlap (see FftInComplex for an in-place version of this routine).

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.
- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.
- *scale* should be `0xffffffff` and the peak signal power should be $< 2^{30}$ unless the properties of the input signal are well understood.
- Only the bottom $\log_2(size)$ bits of *scale* are used; the MSB's are ignored.
- To minimise bus conflicts *output* should be located in on-chip memory.

FftReal

Definition :

`int` FftReal(*output, input, size, scale*)

Arguments :

`short`	*output[ ]*	positive frequency complex output data.
`short`	*input[ ]*	real input data
`long`	*size*	size of FFT.
`long`	*scale*	scaling specification.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 8.
	size not a power of 2.
	size > 32768.

Description :

This routine calculates a real Fast Fourier Transform by performing a complex FFT of half the size and then transforming the data.

On returning, *output* contains positive frequencies only. The negative frequencies are simply the complex conjugate of the positive frequencies. Since the frequency values at both 0 and $F_s/2$ are real, the $F_s/2$ output is placed in the second element of *output* i.e. where the imaginary component of zero frequency would have been stored.

The calculation is not-in-place, so the input and output arrays must not overlap (see FftInReal for an in-place version of this routine).

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.
- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.

- *scale* should be `0xffffffff` and the peak signal power should be $< 2^{30}$ unless the properties of the input signal are well understood.
- Only the bottom $\log_2(size)$ bits of *scale* are used; the MSB's are ignored.
- To minimise bus conflicts *output* should be located in on-chip memory.

IfftComplex

Definition :

`int` IfftComplex(*output*, *input*, *size*, *scale*)

Arguments :

`short`	*output[ ]*	complex output data.
`short`	*input[ ]*	complex input data.
`long`	*size*	size of inverse FFT.
`long`	*scale*	scaling specification.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 4.
	size not a power of 2.
	size > 32768.

Description :

This routine calculates an inverse Fast Fourier Transform. The calculation is not-in-place, so the input and output arrays must not overlap (see IfftInComplex for an in-place version of this routine).

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.
- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.
- *scale* should be `0xffffffff` and the peak signal power should be $< 2^{30}$ unless the properties of the input signal are well understood.
- Only the bottom $\log_2(size)$ bits of *scale* are used; the MSB's are ignored.
- To minimise bus conflicts *output* should be located in on-chip memory.

IfftReal

Definition :

`int` IfftReal(*output*, *input*, *size*, *scale*)

Arguments :

`short`	*output[ ]*	real output data.
`short`	*input[ ]*	positive frequency complex input data.
`long`	*size*	size of inverse FFT.
`long`	*scale*	scaling specification.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 8.
	size not a power of 2.
	size > 32768.

Description :

This routine calculates an inverse Fast Fourier Transform by transforming the data and performing a complex FFT of half the size.

input should contain the complex frequency data for the positive frequencies only. The routine assumes that the negative frequencies are simply the complex conjugate of the positive frequencies. Since the frequency values at both 0 and $F_s/2$ can only be real, the $F_s/2$ value should be placed in the second element of *input* where the imaginary component of 0 would have been stored. When IfftReal returns *output* contains a series of real data samples.

The calculation is not-in-place, so the input and output arrays must not overlap (see IfftInReal for an in-place version of this routine).

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.

- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.
- *scale* should be `0xffffffff` and the peak signal power should be $< 2^{30}$ unless the properties of the input signal are well understood.
- Only the bottom $\log_2(size)$ bits of *scale* are used; the MSB's are ignored.
- To minimise bus conflicts *input* and *output* should both be located in on-chip memory.

FftInComplex

Definition :

`int` FftInComplex(*data, size, scale*)

Arguments :

`short`	*data[ ]*	complex data.
`long`	*size*	size of FFT.
`long`	*scale*	scaling specification.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 4.
	size not a power of 2.
	size > 32768.

Description :

This routine calculates an in-place complex Fast Fourier Transform.

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.
- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.
- *scale* should be `0xffffffff` and the peak signal power should be $< 2^{30}$ unless the properties of the input signal are well understood.
- Only the bottom $\log_2(size)$ bits of *scale* are used; the MSB's are ignored.
- To minimise bus conflicts *data* should be located in on-chip memory.

FftInReal

Definition :

`int` FftInReal(*data, size, scale*)

Arguments :

`short`	*data[ ]*	real data on input, complex positive frequency data on output.
`long`	*size*	size of FFT.
`long`	*scale*	scaling specification.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 8.
	size not a power of 2.
	size > 32768.

Description :

This routine calculates an in-place real Fast Fourier Transform by performing a complex FFT of half the size and then transforming the data.

On returning, *data* contains the complex transform data for the positive frequencies only. The negative frequencies are simply the complex conjugate of the positive frequencies. Since the frequency values at both 0 and $F_s/2$ are real, the $F_s/2$ output is placed in the second element of *data* i.e. where the imaginary component of 0 would have been stored. Although this slightly complicates the output format, it enables the FFT to be carried out in place.

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.

- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.
- *scale* should be `0xffffffff` and the peak signal power should be $< 2^{30}$ unless the properties of the input signal are well understood.
- Only the bottom $\log_2(size)$ bits of *scale* are used; the MSB's are ignored.
- To minimise bus conflicts *data* should be located in on-chip memory.

IfftInComplex

Definition :

`int` IfftInComplex(*data, size, scale*)

Arguments :

`short`	*data[ ]*	complex data.
`long`	*size*	size of inverse FFT.
`long`	*scale*	scaling specification.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 4.
	size not a power of 2.
	size > 32768.

Description :

This routine calculates an in-place complex inverse Fast Fourier Transform.

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.
- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.
- *scale* should be `0xffffffff` and the peak signal power should be $< 2^{30}$ unless the properties of the input signal are well understood.
- Only the bottom $\log_2(size)$ bits of *scale* are used; the MSB's are ignored.
- To minimise bus conflicts *data* should be located in on-chip memory.

IfftInReal

Definition :

`int` IfftInReal(*data, size, scale*)

Arguments :

`short`	*data[ ]*	complex positive frequency data on input, real data on output.
`long`	*size*	size of inverse FFT.
`long`	*scale*	scaling specification.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 8.
	size not a power of 2.
	size > 32768.

Description :

This routine calculates an in-place real inverse Fast Fourier Transform by transforming the data and performing a complex FFT of half the size.

data should contain the complex frequency data for the positive frequencies only. The routine assumes that the negative frequencies are simply the complex conjugate of the positive frequencies. Since the frequency values at both 0 and $F_s/2$ can only be real, the $F_s/2$ value should be placed in the second element of *data* where the imaginary component of 0 would have been stored. When IfftInReal returns *data* contains a series of real samples.

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.

- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.
- *scale* should be `0xffffffff` and the peak signal power should be $< 2^{30}$ unless the properties of the input signal are well understood.
- Only the bottom $\log_2(size)$ bits of *scale* are used; the MSB's are ignored.
- To minimise bus conflicts *data* should be located in on-chip memory.

LogMagnitude

Definition :

`int` LogMagnitude(*output, input, no_elements, fscale*)

Arguments :

`short`	*output[ ]*	real output y.
`short`	*input[ ]*	complex input x
`long`	*no_elements*	number of log magnitude outputs N required.
`float`	*fscale*	output scaling factor

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_elements* < 1.
	$\lvert fscale \rvert \geq 2^{15}/(10 \log_{10} 2^{31})$.

Description :

This routine calculates the log magnitude of the complex input data in decibels, and writes the scaled result into the output array—

$$y(n) = 10 fscale \log_{10}(x(2n)^2 + x(2n+1)^2) \qquad 0 \leq n < N$$

Notes :

- The ordering of real and imaginary components in a complex array is described on page 16.
- Before calling this routine the bit reversal and twiddle tables should be initialised by calling InitFft.
- *output* can be specified to be the same as *input*. In this case, the output overwrites the first half of the input array.

InitFft

Definition :

`int` InitFft(*max_size*)

Arguments :

`long`	*max_size*	Maximum size of FFT that will be required.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*max_size* < 1.
	max_size not a power of 2.
	max_size > 32768.

Description :

This routine generates the sine/cosine and bit reversal tables used by the FFT functions. This function need only be called once, before the first FFT is called. The lookup tables are stored in memory allocated by malloc.

Notes :

- The lookup tables are generated for the specified transform size. Smaller transforms will be performed using the same lookup tables.
- The addresses of the lookup tables are stored in internal variables; they should not be accessed by user code.

FreeFft

Definition :

`void` FreeFft(`void`)

Arguments : `none`

Returns : `void`

Description :

This routine returns the memory used for FFT lookup tables to the heap via free. No further FFT functions can be called before a subsequent InitFft.

8 Windowing

Contents—

Test Functions

The test file `testwin.c` is supplied for the routines in this section. The function Tstwin compares the window functions against precalculated data to ensure accuracy. Note that the windows produced on different architectures may differ in the least significant bit.

GenBlackman

Definition :

`int` GenBlackman(*output*, *win_size*)

Arguments :

`short`	*output[ ]*	array to contain window $w(n)$.
`long`	*win_size*	window size N required.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*win_size* ≤ 1.

Description :

This routine generates a Blackman window in *output*. VectorMult can be used to apply the window to a data array.

The function used is—

$$w(n) = (2^{15}-1)\left[0.42 - 0.5\cos\left(\frac{2\pi n}{N}\right) + 0.08\cos\left(\frac{4\pi n}{N}\right)\right] \quad 0 \leq n < N$$

GenHamming

Definition :

`int` GenHamming(*output*, *win_size*)

Arguments :

`short`	*output[ ]*	array to contain window $w(n)$.
`long`	*win_size*	window size N required.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*win_size* ≤ 1.

Description :

This routine generates a Hamming window in *output*. VectorMult can be used to apply the window to a data array.

The function used is—

$$w(n) = (2^{15} - 1)\left[0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right)\right] \qquad 0 \leq n < N$$

GenHanning

Definition :

`int` GenHanning(*output*, *win_size*)

Arguments :

`short`	*output[ ]*	array to contain window $w(n)$.
`long`	*win_size*	window size N required.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*win_size* ≤ 1.

Description :

This routine generates a Hanning window in array *output*. The routine VectorMult can be used to apply the window to a data array.

The function used is—

$$w(n) = \frac{(2^{15} - 1)}{2}\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] \qquad 0 \leq n < N$$

GenTriangle

Definition :

`int` GenTriangle(*output, win_size*)

Arguments :

`short`	*output[ ]*	array to contain window $w(n)$.
`long`	*win_size*	window size N required.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*win_size* ≤ 1.

Description :

This routine generates a Triangular window in *output*. VectorMult can be used to apply the window to a data array.

The function used is—

$$w(n) = (2^{15} - 1)\left[1 - \left|\frac{2n - N + 1}{N + 1}\right|\right] \qquad 0 \leq n < N$$

9 Filters

Contents—

Routine Timings

The execution time in microseconds of the filter routines on a 10 MIPS SH-1 are given below. The times for the single sample versions are per routine call. The times for the multiple sample versions are on a per sample basis, computed by dividing the time to process a 100 sample frame by 100. Note that the times vary for different result shift values.

No. of coeffs	Function			
	Fir	Lms	Fir1	Lms1
5	8.6	24.9	17.2	34.7
10	10.3	40.6	18.9	50.4
100	46.1	328.4	54.7	338.2

No. of biquads	Function			
	Iir	DIir	Iir1	DIir1
1	10.5	40.1	18.1	48.0
2	19.7	77.6	28.8	86.4
20	185.5	752.8	221.4	784.3

Coefficient Scaling

All digital signal processing is likely to introduce noise into the signal due to saturation or quantisation. The scaling of coefficients must be chosen carefully to balance the effects of saturation and quantisation.

If the coefficients are too large saturation may occur; if they are too low excessive quantisation noise may be introduced.

For FIR (finite impulse response) filters on the SH-1 no saturation will occur if

$$\sum |coeffs| < 2^{26} \text{ and } res\_shift = 26$$

However, for many input signals this scaling is overly pessimistic; smaller result shifts (or larger coefficient values) could be used with a low likelihood of saturation, but with significantly reduced quantisation noise.

IIR (infinite impulse response) filters have a recursive structure, which means that the scaling approach described above is inappropriate.

LMS (least mean squares) adaptive filters obey the same conventions as FIR filters. However, the coefficients may be pushed into saturation as the coefficients are adapted.

Workspace

Digital filters have state that must be preserved from the processing of one sample to the next. This filter state, or workspace, must be stored in memory that can be accessed with minimum overhead. Moreover, the filter state must be initialised before the first sample can be processed.

The structure of the workspace memory is liable to change in the future, so user programs should not attempt to read or modify this memory — it should only be accessed by the library functions.

Test Functions

The filter functions are tested for bit exactness by the file `testfilt.c`. The FIR, IIR and DIIR filters are tested by filtering a number of sinusoids through a low-pass filter with a cut-off at exactly one-quarter the sample rate. The LMS filter is tested by passing coloured (FIR filtered) noise through it and requiring it to adapt to a set of precalculated filter coefficients.

The test functions are Tstfir, Tstfir1, Tstiir, Tstiir1, Tstdiir, Tstdiir1, Tstlms and Tstlms1.

Fir

Definition :

`int` Fir(*output*, *input*, *no_samples*, *coeff*, *no_coeffs*, *res_shift*, *workspace*)

Arguments :

`short`	*output[ ]*	output samples y.
`short`	*input[ ]*	input samples x.
`long`	*no_samples*	number of samples N to be filtered.
`short`	*coeff[ ]*	array of filter coefficients h.
`long`	*no_coeffs*	number of coefficients K (length of filter).
`int`	*res_shift*	right shift applied to each output.
`short`	**workspace*	variable containing pointer to filter memory.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_samples* < 1.
	no_coeffs ≤ 2.
	res_shift < 0.
	res_shift > 27.

Description :

This routine implements a finite impulse response (FIR) filter. It uses *workspace* to record the most recent input samples. The result of filtering the data in *input* is written to *output*—

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)x(n-k) \right] .2^{-res\_shift}$$

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Notes :

- Filtering is likely to introduce noise due to saturation or quantisation. If the coefficients are too large saturation may occur; if they are too low excessive quantisation noise may be introduced. The scaling of coefficients must be carefully considered to balance the effects of quantisation and saturation.

  One scheme that avoids saturation is to scale the coefficients so that $\sum \mathrm{abs}(\mathit{coeffs}) < 2^{26}$ and $\mathit{res\_shift} = 26$. For many input signals smaller result shifts could be used with a low likelihood of saturation, but with significantly reduced quantisation noise.

- Before calling this routine for a new filter, initialise the filter workspace by calling InitFir.

- Execution is fastest when *res_shift* is set to 17 or 18.

- *output* may be the same as *input* in which case *input* is overwritten.

Fir1

Definition :

`int` Fir1(*output, input, coeff, no_coeffs, res_shift, workspace*)

Arguments :

`short`	**output*	output sample $y(n)$.
`short`	*input*	input sample $x(n)$.
`short`	*coeff[ ]*	array of filter coefficients h.
`long`	*no_coeffs*	number of coefficients K (length of filter).
`int`	*res_shift*	right shift applied to each output.
`short`	**workspace*	variable containing pointer to filter memory.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_coeff* ≤ 2.
	res_shift < 0.
	res_shift > 27.

Description :

This routine implements a finite impulse response (FIR) filter for one sample only. It uses *workspace* to record the most recent input samples. The result of filtering the data in *input* is written to *output*—

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)x(n-k) \right] .2^{-res\_shift}$$

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Notes :

- Filtering is likely to introduce noise due to saturation or quantisation. If the coefficients are too large saturation may occur; if they are too low excessive quantisation noise may be introduced. the scaling of coefficients must be carefully considered to balance the effects of quantisation and saturation.

  One scheme that avoids saturation is to scale the coefficients so that $\sum \text{abs}(\textit{coeffs}) < 2^{26}$ and $\textit{res\_shift} = 26$. For many input signals smaller result shifts could be used with a low likelihood of saturation, but with significantly reduced quantisation noise.

- Before calling this routine for a new filter, initialise the filter workspace by calling InitFir.

- Execution is fastest when *res_shift* is set to 17 or 18.

lir

Definition :

`int` lir(*output, input, no_samples, coeff, no_sections, workspace*)

Arguments :

`short`	*output[ ]*	output samples y_{K-1}.
`short`	*input[ ]*	input samples x_0.
`long`	*no_samples*	number of samples N to be filtered.
`short`	*coeff[ ]*	filter coefficients.
`long`	*no_sections*	number of second order filter sections K.
`short`	**workspace*	variable containing pointer to filter memory.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_samples* < 1.
	no_sections < 1.
	$a_{0k} < 0$.
	$a_{0k} > 16$.

Description :

This routine implements an infinite impulse response (IIR) filter.

The filter is implemented as a cascade of K second order filters called biquads, with an additional scaling performed on the biquad output. This implementation uses the Direct Form II representation, where the a coefficients are specified in fractional (Q15) format and the output of each biquad is given by

$$d_k(n) = \left[a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{15}x(n)\right].2^{-15}$$

$$y_k(n) = \left[b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)\right].2^{-a_{0k}}$$

The input $x_k(n)$ to the k^{th} section is the output $y_{k-1}(n)$ of the previous section. The input to the first $(k = 0)$ section is taken from *input*. The output from the last $(k = K - 1)$ section is written to *output*.

Each biquad is calculated in 32 bits using saturating arithmetic. Following the multiply by 2^{15} or $2^{-a_{0k}}$, 16 LSB's are extracted from the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

The filter coefficients should be specified in *coeff* in the order—

$$a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01}...b_{2K-1}$$

Notes :

- Before calling this routine for a new filter, initialise the filter by calling Initlir.
- a_{0k} describes a shift rather than a multiply. The equivalent multiply would be by $2^{-a_{0k}}$. If $a_{0k} < 0$ or $a_{0k} > 16$ the output of the biquad is undefined.
- *output* may be the same as *input* in which case *input* is overwritten.
- To minimise bus conflicts *workspace* should be located in on-chip memory.

lir1

Definition :

`int` lir1(*output, input, coeff, no_sections, workspace*)

Arguments :

`short`	**output*	output sample $y_{K-1}(n)$.
`short`	*input*	input sample $x_0(n)$.
`short`	*coeff[ ]*	filter coefficients.
`long`	*no_sections*	number of second order filter sections K.
`short`	**workspace*	variable containing pointer to filter memory.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_sections* < 1.
	$a_{0k} < 0$.
	$a_{0k} > 16$.

Description :

This routine implements an infinite impulse response (IIR) filter for one sample only.

The filter is implemented as a cascade of K second order filters called biquads, with an additional scaling performed on the biquad output. This implementation uses the Direct Form II representation, where the a coefficients are specified in fractional (Q15) format and the output of each biquad is given by

$$d_k(n) = \left[a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{15}x(n)\right].2^{-15}$$

$$y_k(n) = \left[b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)\right].2^{-a_{0k}}$$

The input $x_k(n)$ to the kth section is the output $y_{k-1}(n)$ of the previous section. The input to the first ($k = 0$) section is taken from *input*. The output from the last ($k = K - 1$) section is written to *output*. *coeff* should contain coefficients in the order—

$$a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}$$

Each biquad is calculated in 32 bits using saturating arithmetic. Following the multiply by 2^{15} or $2^{-a_{0k}}$, 16 LSB's are extracted from the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Notes :

- Before calling this routine for a new filter, initialise the filter by calling Initlir.
- a_{0k} describes a shift rather than a multiply. The equivalent multiply would be by $2^{-a_{0k}}$. If $a_{0k} < 0$ or $a_{0k} > 16$ the output of the biquad is undefined.
- To minimise bus conflicts *workspace* should be located in on-chip memory.

Dlir

Definition :

`int` Dlir(*output, input, no_samples, coeff, no_sections, workspace*)

Arguments :

`short`	*output[ ]*	output samples y_{K-1}.
`short`	*input[ ]*	input samples x.
`long`	*no_samples*	number of samples N to be filtered.
`long`	*coeff[ ]*	filter coefficients.
`long`	*no_sections*	number of second order filter sections K.
`long`	**workspace*	variable containing pointer to filter memory.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_samples* < 1.
	no_sections < 1.
	$a_{0k} < 0$.
	$a_{0k} > 32$ for $k < K-1$.
	$a_{0k} > 48$ for $k = K-1$.

Description :

This routine implements an infinite impulse response (IIR) filter with double precision coefficients and delay node storage.

The filter is implemented as a cascade of K second order filters called biquads, with an additional scaling performed on the biquad output. This implementation uses the Direct Form II representation, where the a coefficients are specified in fractional (Q31) format and the output of each biquad is given by

$$d_k(n) = \left[a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{31}x(n)\right].2^{-31}$$

$$y_k(n) = \left[b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)\right].2^{-a_{0k}}$$

The input $x_k(n)$ to the kth section is the output $y_{k-1}(n)$ of the previous section. The input to the first ($k = 0$) section is taken from *input* after a multiply by 2^{16}. The output from the last ($k = K - 1$) section is written to *output*. *coeff* should contain coefficients in the order—

$$a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}$$

Dlir differs from lir in that the filter coefficients are specified as 32 rather than 16 bit values. Consequently 64 bit accumulators is used, with intermediate biquad outputs stored as 32 bit quantities following the shifts by 31 or a_{0k}. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.In the final stage only 16 bits are retained following the shift by a_{0K-1}; again surplus LSB's are rounded and surplus MSB's are saturated.

Notes :

- Before calling this routine for a new filter, initialise the filter by calling InitDlir.
- a_{0k} describes a shift rather than a multiply. The equivalent multiply would be by $2^{-a_{0k}}$. If $a_{0k} < 0$ or $a_{0k} > 32$ (or $a_{0K-1} > 48$) the output of the biquad is undefined.
- The most common use of Dlir specifies the coefficients in Q31 format. In this case a_{0k} should be set to 31 for $k < K - 1$ and to 47 for $k = K - 1$.
- The least significant bit of each 32 bit coefficient and delay node value is ignored.
- When possible lir should be used in preference to Dlir as it runs an order of magnitude faster on the SH-1.
- *output* may be the same as *input* in which case *input* is overwritten.
- To minimise bus conflicts *workspace* should be located in on-chip memory.

Dlir1

Definition :

`int` Dlir1(*output*, *input*, *coeff*, *no_sections*, *workspace*)

Arguments :

`short`	**output*	output sample $y_{K-1}(n)$.
`short`	*input*	input sample $x_0(n)$.
`long`	*coeff[ ]*	filter coefficients.
`long`	*no_sections*	number of second order filter sections K.
`long`	**workspace*	variable containing pointer to filter memory.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_sections* < 1.
	$a_{0k} < 0$.
	$a_{0k} > 32$ for $k < K-1$.
	$a_{0k} > 48$ for $k = K-1$.

Description :

This routine implements a double precision infinite impulse response (IIR) filter for one sample only.

The filter is implemented as a cascade of K second order filters called biquads, with an additional scaling performed on the biquad output. This implementation uses the Direct Form II representation, where the a coefficients are specified in fractional (Q31) format and the output of each biquad is given by

$$d_k(n) = \left[a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{31}x(n)\right].2^{-31}$$

$$y_k(n) = \left[b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)\right].2^{-a_{0k}}$$

The input $x_k(n)$ to the kth section is the output $y_{k-1}(n)$ of the previous section. The input to the first ($k = 0$) section is taken from *input* after

a multiply by 2^{16}. The output from the last ($k = K - 1$) section is written to *output*. *coeff* should contain coefficients in the order—

$$a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}$$

As with Dlir the filter coefficients and delay node values are stored as 32 rather than 16 bit values. Consequently 64 bit accumulators are used, with intermediate biquad outputs stored as 32 bit quantities following the shift by 31 or a_{0k}. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.In the final stage only 16 bits are retained following the shift by a_{0K-1}; again surplus LSB's are rounded and surplus MSB's are discarded.

Notes :

- Before calling this routine for a new filter, initialise the filter by calling InitDlir.
- a_{0k} describes a shift rather than a multiply. The equivalent multiply would be by $2^{-a_{0k}}$. If $a_{0k} < 0$ or $a_{0k} > 32$ (or $a_{0K-1} > 48$) the output of the biquad is undefined.
- The most common use of Dlir specifies the coefficients in Q31 format. In this case a_{0k} should be set to 31 for $k < K - 1$ and to 47 for $k = K - 1$.
- The least significant bit of each 32 bit coefficient and delay node value is ignored.
- When possible lir should be used in preference to Dlir as it runs an order of magnitude faster on the SH-1.
- To minimise bus conflicts *workspace* should be located in on-chip memory.

Lms

Definition :

`int` Lms(*output*, *input*, *ref*, *no_samples*, *coeff*, *no_coeffs*, *res_shift*, *conv_fact*, *workspace*)

Arguments :

`short`	*output[ ]*	output samples y.
`short`	*input[ ]*	input samples x.
`short`	*ref_output[ ]*	desired output value d.
`long`	*no_samples*	number of samples N to be filtered.
`short`	*coeff[ ]*	adaptive filter coefficients h.
`long`	*no_coeffs*	number of coefficients K.
`int`	*res_shift*	right shift applied to each output.
`short`	*conv_fact*	convergence factor 2μ.
`short`	**workspace*	variable containing pointer to filter memory.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_samples* < 1.
	no_coeffs ≤ 2.
	res_shift < 0.
	res_shift > 27.

Description :

This routine implements a real adaptive FIR filter using the least mean square algorithm.

The FIR filter is defined as—

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k) x(n-k) \right] .2^{-res\_shift}$$

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction

rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

The Widrow-Hoff algorithm is used to update the filter coefficients—

$$h_{n+1}(k) = h_n(k) + 2\mu e(n)x(n-k)$$

where the addition is saturating and $e(n)$ is the (saturated) error between the desired signal and the actual filter output—

$$e(n) = d(n) - y(n)$$

The calculation of $2\mu e(n)x(n-k)$ requires two 16×16 multiplies. In both multiplies bits [31:16] of the product are retained, rounded and saturated if necessary.

Notes :

- Any digital signal processing can introduce noise due to saturation or quantisation effects. If the coefficients are too large saturation may occur; if they are too low excessive quantisation noise may be introduced. The scaling of coefficients must be carefully considered to balance the effects of quantisation and saturation.

  One scheme that avoids saturation is to scale the coefficients so that $\sum \text{abs}(coeffs) < 2^{26}$ and *res_shift* = 26. For many input signals smaller result shifts could be used with a low likelihood of saturation, but with significantly reduced quantisation noise.

  However, it is not possible to guarantee that the coefficients generated by an LMS filter will conform to the scaling convention described above.

- *conv_fact* should normally be positive; it should never be 0x8000.
- Before calling this routine for a new filter, initialise the filter workspace by calling InitLms.
- Execution is fastest when *res_shift* is set to 17 or 18.
- *output* may be the same as *input* or *ref_output* in which case *input* or *ref_output* is overwritten.

- If no adaption is required 2μ may be set to zero. Alternatively Fir or Fir1 may be used with the same filter coefficients and workspace.

Lms1

Definition :

`int Lms1(` *output, input, ref_output, coeff, no_coeffs, res_shift, conv_fact, workspace*)

Arguments :

`short`	**output*	output sample $y(n)$.
`short`	*input*	input sample $x(n)$.
`short`	*ref*	desired output value $d(n)$.
`short`	*coeff[ ]*	adaptive filter coefficients h.
`long`	*no_coeffs*	number of coefficients K.
`int`	*res_shift*	right shift applied to each output.
`short`	*conv_fact*	convergence factor 2μ.
`short`	**workspace*	variable containing pointer to filter memory.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_coeffs* ≤ 2.
	res_shift < 0.
	res_shift > 27.

Description :

This routine implements a real adaptive FIR filter using the least mean square algorithm, for a single input sample.

The FIR filter is defined as—

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k) x(n-k) \right] .2^{-res\_shift}$$

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction

rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

The Widrow-Hoff algorithm is used to update the filter coefficients—

$$h_{n+1}(k) = h_n(k) + 2\mu e(n)x(n-k)$$

where the addition is saturating and $e(n)$ is the (saturated) error between the desired signal and the actual filter output—

$$e(n) = d(n) - y(n)$$

The calculation of $2\mu e(n)x(n-k)$ requires two 16×16 multiplies. In both multiplies bits [31:16] of the product are retained, rounded and saturated if necessary.

Notes :

- Any digital signal processing can introduce noise due to saturation or quantisation effects. If the coefficients are too large saturation may occur; if they are too low excessive quantisation noise may be introduced. The scaling of coefficients must be carefully considered to balance the effects of quantisation and saturation.

  One scheme that avoids saturation is to scale the coefficients so that $\sum \text{abs}(\textit{coeffs}) < 2^{26}$ and $\textit{res\_shift} = 26$. For many input signals smaller result shifts could be used with a low likelihood of saturation, but with significantly reduced quantisation noise.

  However, it is not possible to guarantee that the coefficients generated by an LMS filter will conform to the scaling convention described above.

- *conv_fact* should normally be positive; it should never be 0x8000.
- Before calling this routine for a new filter, initialise the filter workspace by calling InitLms.
- Execution is fastest when *res_shift* is set to 17 or 18.
- *output* may be the same as *input* or *ref_output* in which case the *input* or ref_output is overwritten.

- If no adaption is required 2μ may be set to zero. Alternatively Fir or Fir1 may be used with the same filter coefficients and workspace.

InitFir

Definition :

`int` InitFir(*workspace*, *no_coeffs*)

Arguments :

`short`	***workspace*	address of variable containing pointer to buffer reserved for filter memory.
`long`	*no_coeffs*	number of coefficients K.

Returns : `int`

`EDSP_OK`	success.
`EDSP_NO_HEAP`	insufficient space available from malloc.
`EDSP_BAD_ARG`	*no_coeffs* ≤ 2.

Description :

This routine allocates, via malloc, the memory required for subsequent calls to Fir and Fir1. The history of previous input samples is initialised to zero.

Notes :

- The *workspace* buffer allocated by InitFir should only be manipulated by Fir, Fir1, Lms and Lms1.

Initlir

Definition :

`int` Initlir(*workspace*, *no_sections*)

Arguments :

`short`	***workspace*	address of variable containing pointer to buffer reserved for filter memory.
`long`	*no_sections*	number of second order filter sections K.

Returns : `int`

`EDSP_OK`	success.
`EDSP_NO_HEAP`	insufficient space available from malloc.
`EDSP_BAD_ARG`	*no_sections* < 1.

Description :

This routine allocates, via malloc, the memory required for subsequent calls to lir and lir1. All delay node values are initialised to zero.

Notes :

- The *workspace* buffer allocated by Initlir should only be manipulated by lir and lir1.

InitDlir

Definition :

`int` InitDlir(*workspace*, *no_sections*)

Arguments :

`long`	***workspace*	address of variable containing pointer to buffer reserved for filter memory.
`long`	*no_sections*	number of second order filter sections K.

Returns : `int`

`EDSP_OK`	success.
`EDSP_NO_HEAP`	insufficient space available from malloc.
`EDSP_BAD_ARG`	*no_sections* < 1.

Description :

This routine allocates, via malloc, the memory required for subsequent calls to Dlir and Dlir1. All delay node values are initialised to zero.

Notes :

- The *workspace* buffer allocated by InitDlir should only be manipulated by Dlir and Dlir1.

InitLms

Definition :

`int` InitLms(*workspace*, *no_coeffs*)

Arguments :

`short`	***workspace*	address of variable containing pointer to buffer reserved for filter memory.
`long`	*no_coeffs*	number of coefficients K.

Returns : `int`

`EDSP_OK`	success.
`EDSP_NO_HEAP`	insufficient space available from malloc.
`EDSP_BAD_ARG`	*no_coeffs* ≤ 2.

Description :

This routine allocates, via malloc, the memory required for subsequent calls to Lms and Lms1.The history of previous input samples is initialised to zero.

Notes :

- The *workspace* buffer allocated by InitLms should only be manipulated by Fir, Fir1, Lms and Lms1.

10 Convolution and Correlation

Contents—

Routine Timings

The execution time in microseconds of the filter routines on a 10 MIPS SH-1 are given below for ConvPartial, ConvCyclic, CorrCyclic and Correlate (with no correlations past the end of the longer input). The times are given per output sample, computed by dividing the time to produce K output samples by K. Note that *iw* is always the smaller input array.

Size of *iw*	K	Routine			
		ConvPartial	ConvCyclic	CorrCyclic	Correlate
5	5	9.7	9.7	9.0	8.1
10	10	10.6	11.2	10.2	9.0
100	100	54.6	55.4	47.7	46.4

Where correlations or convolutions past the ends of arrays take place, as happens in ConvComplete and sometimes in Correlate, the output samples are produced with less computation. The most extreme case of this is when the two input arrays are the same size n, and the computations continue until the arrays overlap by only one sample. In this case ConvComplete produces $2n - 1$ samples, and Correlate produces n samples, with all but one of those samples formed from computation past array ends.

For this extreme case, the timings in microseconds per output sample, again computed by dividing the time to produce K output samples by K, are—

Size of *iw*		Routine	
	K	ConvComplete	Correlate
5	9/5	6.5	8.4
10	19/10	7.1	8.2
100	199/100	29.0	26.1

Test Functions

The Convolution/Correlation functions are tested for bit exactness by the file `testconv.c` which calls the functions Tconvcmp, Tconvcyc, Tconvpar, Tcorr and Tcorrcyc.

ConvComplete

Definition :

`int` ConvComplete(*output, iw, ix, iw_size, ix_size, res_shift*)

Arguments :

`short`	*output[ ]*	output y.
`short`	*iw[ ]*	input w.
`short`	*ix[ ]*	input x.
`long`	*iw_size*	size of *iw* W.
`long`	*ix_size*	size of *ix* X.
`int`	*res_shift*	right shift applied to each output.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*iw_size* < 1.
	ix_size < 1.
	res_shift < 0.
	res_shift > 27.

Description :

This routine completely convolves the two input arrays and puts the result in the output array—

$$y(m) = \left[\sum_{i=0}^{W-1} w(i)x(m-i) \right] .2^{-res\_shift} \qquad 0 \le m < W + X - 1$$

The routine uses zeros in place of elements before the start and after the end of the two arrays.

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Notes :

- The output array size must be at least $W + X - 1$.
- To minimise bus conflicts *iw* should be located in on-chip memory.

ConvCyclic

Definition :

`int` ConvCyclic(*output, iw, ix, size, res_shift*)

Arguments :

`short`	*output[ ]*	output y.
`short`	*iw[ ]*	input w.
`short`	*ix[ ]*	input x.
`long`	*size*	size of arrays N.
`int`	*res_shift*	right shift applied to each output.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 1.
	res_shift < 0.
	res_shift > 27.

Description :

This routine cyclically convolves the two input arrays and puts the result in the output array—

$$y(m) = \left[\sum_{i=0}^{N-1} w(i) x(|m - i + N|_N) \right] .2^{-res\_shift} \qquad 0 \le m < N$$

where

$|i|_n$ = residue of i modulo n.

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Notes :

- To minimise bus conflicts *iw* should be located in on-chip memory.

ConvPartial

Definition :

`int` ConvPartial(*output, iw, ix, iw_size, ix_size, res_shift*)

Arguments :

`short`	*output[ ]*	output y.
`short`	*iw[ ]*	smaller input w.
`short`	*ix[ ]*	larger input x.
`long`	*iw_size*	size of *iw* W.
`long`	*ix_size*	size of *ix* X.
`int`	*res_shift*	right shift applied to each output.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*iw_size* < 1.
	ix_size < 1.
	ix_size $<$ *iw_size*.
	res_shift < 0.
	res_shift > 27.

Description :

This routine convolves the two input arrays but does not include outputs derived from elements outside the array boundaries—

$$y(m) = \left[\sum_{i=0}^{W-1} w(i)x(m+W-1-i) \right] .2^{-res\_shift} \qquad 0 \le m \le X - W$$

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Notes :

- The output array size must be at least $X - W + 1$.
- x must be the larger array, i.e. $X \geq W$.
- To minimise bus conflicts *iw* should be located in on-chip memory.

Correlate

Definition :

`int` Correlate(*output, iw, ix, iw_size, ix_size, no_corr, res_shift*)

Arguments :

`short`	*output[ ]*	output y.
`short`	*iw[ ]*	smaller input w.
`short`	*ix[ ]*	larger input x.
`long`	*iw_size*	size of *iw* W.
`long`	*ix_size*	size of *ix* X.
`long`	*no_corr*	number of correlations M to compute.
`int`	*res_shift*	right shift applied to each output.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*iw_size* < 1.
	ix_size < 1.
	no_corr < 1.
	ix_size $<$ *iw_size*.
	res_shift < 0.
	res_shift > 27.

Description :

This routine correlates the two input arrays and puts the result in the output array—

$$y(m) = \left[\sum_{i=0}^{W-1} w(i)x(i+m) \right] .2^{-res\_shift} \qquad 0 \le m < M$$

Correlation past the end of the x array is permissible, i.e. $X < W + M$. In this case the routine uses zeros in place of the extra array elements.

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction

rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Notes :

- *res_shift* = 0 corresponds to a normal integer calculation.
  res_shift = 15 corresponds to a fractional calculation.
- x must be the larger array, i.e. $X \geq W$, if the arrays are not the same size.
- To minimise bus conflicts *iw* should be located in on-chip memory.

CorrCyclic

Definition :

`int` CorrCyclic(*output, iw, ix, size, res_shift*)

Arguments :

`short`	*output[ ]*	output y.
`short`	*iw[ ]*	input w.
`short`	*ix[ ]*	input x.
`long`	*size*	size of arrays N.
`int`	*res_shift*	right shift applied to each output.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*size* < 1.
	res_shift < 0.
	res_shift > 27.

Description :

This routine cyclically correlates the two input arrays and puts the result in the output array—

$$y(m) = \left[\sum_{i=0}^{N-1} w(i)x(|i+m|_N) \right] .2^{-res\_shift} \qquad 0 \le m < N$$

where

$|i|_N$ = residue of i modulo N.

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Notes :

- To minimise bus conflicts *iw* should be located in on-chip memory.

11 Miscellaneous

Contents—

Test Functions

The Miscellaneous functions are tested for bit exactness by the file `testmisc.c`. The Gaussian white noise generator is tested by measuring the distribution and spectrum of the samples. The 'whiteness' of the spectrum and a Chi-squared test are checked against performance targets. The remaining functions are compared against precalculated data.

The functions Tgwn, Tmatrixm, Tvectm, Tmean, Tvar, Tmspower and Textr perform the testing.

GenGWnoise

Definition :

`int` GenGWnoise(*output, no_samples, variance*)

Arguments :

`short`	*output[ ]*	output white noise samples.
`long`	*no_samples*	number of samples required.
`float`	*variance*	variance of noise distribution σ^2.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_samples* < 1.
	variance ≤ 0.0.

Description :

This routine generates Gaussian white noise with zero mean and user-specified variance. Samples are produced in pairs using the modified Box-Muller method described in [1]. To produce a pair of output samples, the standard random number generator provided by rand is used to generate pairs of random numbers r_1, r_2 between -1 and 1, until a pair is found whose sum of squares x is less than 1. The pair of output samples o_1, o_2 are then calculated

$$o_1 = \sigma r_1 \sqrt{-2ln(x)/x}$$

$$o_2 = \sigma r_2 \sqrt{-2ln(x)/x}$$

Notes :

- If an odd number of samples is requested, the second sample of the last pair is discarded.
- The random number generator seed can be initialised to any integer between 1 and $2^{31} - 1$ using srand.

- This routine is not strictly re-entrant since any calls to rand in an interrupt routine or between calls to this routine will affect the sequence of random numbers used. However, such calls will *not* affect the random properties of the white noise generated.
- Floating point arithmetic is used in this function, so its use should be restricted to test programs rather than application programs whenever possible.

MatrixMult

Definition :

`int` MatrixMult(*op_matrix, ip_matrix1, ip_matrix2,*
no_rows1, no_cols1, no_cols2, res_shift)

Arguments :

`void`	**op_matrix*	pointer to first element of output.
`void`	**ip_matrix1*	pointer to first element of input 1.
`void`	**ip_matrix2*	pointer to first element of input 2.
`long`	*m*	row dimension of matrix1.
`long`	*n*	column dimension of matrix1, row dimension of matrix2.
`long`	*p*	column dimension of matrix2.
`int`	*res_shift*	right shift applied to each output.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*m, n or p* < 1.
	res_shift < 0.
	res_shift > 27.

Description :

This routine multiplies the two matrices *ip_matrix1* and *ip_matrix2* and stores the result in *op_matrix*. *ip_matrix1* is $m \times n$, *ip_matrix2* is $n \times p$ and *op_matrix* is $m \times p$.

The sum is accumulated in 42 bits. Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the accumulator. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.

Each matrix is stored in the normal 'C' manner (row major order)—

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \end{pmatrix}$$

Notes :

- To minimise bus conflicts *ip_matrix1* should be located in on-chip memory.
- The function prototype specifies the array parameters as `void *` to allow arbitrary array sizes to be specified. These parameters should point to `short` data.

VectorMult

Definition :

`int` VectorMult(*output, ip1, ip2, no_elements, res_shift*)

Arguments :

`short`	*output[ ]*	output.
`short`	*ip1[ ]*	input.
`short`	*ip2[ ]*	input.
`long`	*no_elements*	number of elements to evaluate.
`int`	*res_shift*	right shift applied to each output.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_elements* < 1.
	res_shift < 0.
	res_shift > 16.

Description :

This routine multiplies pairs of elements from *ip1* and *ip2* and stores the results in *output*.

Notes :

- Each 16 bit output is extracted from bit positions [res_shift+15:res_shift] of the product. This extraction rounds the surplus LSB's and checks the surplus MSB's for overflow, saturating if necessary.
- This routine performs elementwise multiplications. To calculate a dot product use MatrixMult with n set to 1.

MsPower

Definition :

`int` MsPower(*output, input, no_elements*)

Arguments :

`long`	**output*	result.
`short`	*input[ ]*	input.
`long`	*no_elements*	number of elements N to evaluate.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_elements* < 1.

Description :

This routine calculates the Mean Square Power of the input data—

$$\text{Mean Square Power} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2$$

Notes :

- The division result is rounded to the nearest integral value.
- The sum is accumulated in 64 bits. If *no_elements* is more than 2^{32}-1 overflow may occur.

Mean

Definition :

`int` Mean(*mean, input, no_elements*)

Arguments :

`short`	**mean*	mean of data in *input* $\overline{x}$.
`short`	*input[ ]*	input x.
`long`	*no_elements*	number of elements N to process.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_elements* < 1.

Description :

This routine calculates the mean of *input*—

$$\overline{x} = \frac{1}{N} \sum_{i=0}^{N-1} x(i)$$

Notes :

- The division result is rounded to the nearest integral value.
- The sum is accumulated in 32 bits. If *no_elements* is more than 2^{16}-1 overflow may occur.

Variance

Definition :

`int` Variance(*variance, mean, input, no_elements*)

Arguments :

`long`	**variance*	The variance of input σ^2.
`short`	**mean*	mean of data $\overline{x}$.
`short`	*input[ ]*	input x.
`long`	*no_elements*	number of elements N to process.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_elements* < 1.

Description :

This routine first calculates the mean of *input*—

$$\overline{x} = \frac{1}{N} \sum_{i=0}^{N-1} x(i)$$

It then computes the variance—

$$\sigma^2 = \frac{1}{N} \sum_{i=0}^{N-1} (x(i) - \overline{x})^2$$

Notes :

- The division results are rounded to the nearest integral values.
- $\overline{x}$ is accumulated in 32 bits and is not checked for overflow. If *no_elements* is more than 2^{16}-1 overflow may occur.
- σ^2 is accumulated in 64 bits and is not checked for overflow.

MaxI

Definition :

`int` MaxI(*max_ptr*, *input*, *no_elements*)

Arguments :

`short`	***max_ptr*	address of pointer to the maximum element.
`short`	*input[ ]*	input.
`long`	*no_elements*	number of elements to process.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_elements* < 1.

Description :

This routine searches *input* to find the element with the maximum value, and returns its address in *max_ptr*.

Notes :

- If several elements have the same maximum value the one nearest the start of *input* is returned

MinI

Definition :

`int` MinI(*min_ptr, input, no_elements*)

Arguments :

`short`	***min_ptr*	address of pointer to the minimum element.
`short`	*input[ ]*	input.
`long`	*no_elements*	number of elements to process.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_elements* < 1.

Description :

This routine searches *input* to find the element with the minimum value, and returns its address in *min_ptr*.

Notes :

- If several elements have the same minimum value the one nearest the start of *input* is returned

PeakI

Definition :

`int` PeakI(*peak_ptr, input, no_elements*)

Arguments :

`short`	***peak_ptr*	address of pointer to the peak element.
`short`	*input[ ]*	input.
`long`	*no_elements*	number of elements to process.

Returns : `int`

`EDSP_OK`	success.
`EDSP_BAD_ARG`	*no_elements* < 1.

Description :

This routine searches *input* to find the element with the maximum absolute value, and returns its address in *peak_ptr*.

Notes :

- If several elements have the same peak value the one nearest the start of *input* is returned

A Contents of Distribution Disk

The distribution disk has six main directories, `include`, `lib`, `ansisrc`, `optsrc`, `demo` and `test` with the contents listed below.
The `lib` directory contains the file—

```
ensigma.lib
```

The `include` directory contains the file—

```
ensigdsp.h
```

The `optsrc` directory contains the files—

```
black.c       fftinc.asm    iir1.asm      logmag.c      peak.asm
convcomp.asm  fftinr.asm    initdiir.c    make.bat      shift.asm
convcycl.asm  fftr.asm      initfft.c     makefile      triangle.c
convpart.asm  fir.asm       initfir.c     matrix.asm    variance.asm
corrcycl.asm  fir1.asm      initiir.c     max.asm       vectmult.asm
correlat.asm  gengwn.c      initlms.c     mean.asm
diir.asm      hamming.c     lbr.sub       min.asm
diir1.asm     hanning.c     lms.asm       mspower.asm
fftc.asm      iir.asm       lms1.asm      mult.asm
```

The `ansisrc` directory contains the files—

```
black.c     diir1.c     fir1.c      ifftreal.c  initlms.c   mean.c
convcomp.c  fftcom.c    gengwn.c    iir.c       lms.c       min.c
convcycl.c  fftcore.c   hamming.c   iir1.c      lms1.c      mspower.c
convpart.c  fftincom.c  hanning.c   initdiir.c  logmag.c    peak.c
corrcycl.c  fftireal.c  ifftcom.c   initfft.c   makefile    triangle.c
correlat.c  fftreal.c   iffticom.c  initfir.c   matrix.c    variance.c
diir.c      fir.c       ifftirel.c  initiir.c   max.c       vectmult.c
```

The `demo` directory contains the files—

```
c0.src    demo.c    demoio.c  demoio.h  link.cmd  make.bat  makefile
```

The `test` directory contains the files—

```
c0.src      tconvpar.c  testfilt.c  tgwn.c      tstdiir1.c  tstlms1.c
link.cmd    tcorr.c     testhead.h  tifft.c     tstfir.c    tstwin.c
make.bat    tcorrcyc.c  testmisc.c  tmatrixm.c  tstfir1.c   tvar.c
makefile    testconv.c  testwin.c   tmean.c     tstiir.c    tvectm.c
tconvcmp.c  testdat     textr.c     tmspower.c  tstiir1.c
tconvcyc.c  testfft.c   tfft.c      tstdiir.c   tstlms.c
```

It also contains the `testdat` directory, which contains the files—

```
filtdat.txt   tconvpar.txt  tmatrixm.txt  tstfir.txt    tvar.txt
miscdat.txt   tcorr.txt     tmean.txt     tstiir.txt    tvectm.txt
tconvcmp.txt  tcorrcyc.txt  tmspower.txt  tstlms.txt
tconvcyc.txt  textr.txt     tstdiir.txt   tstwin.txt
```

References

[1] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press 1988.