Using The SigLib DSP Library On Fixed Point Processors

Numerix Ltd.

September 2000

Numerix Ltd.
7 Dauphine Close, Coalville, Leics, LE67 4QQ, UK
Phone : +44 (0)7050 803996, Fax : +44 (0)7050 803997
Internet : http://www.numerix-dsp.com
Email : support@numerix-dsp.com

Introduction

SigLib is an ANSI C source library of DSP functions. All the data is defined using the C/C++ 'typedef' declaration and can therefore be redefined to support any required data type. By default the library is compiled and tested on floating-point devices however it can be easily recompiled to support fixed-point as well.

For fixed point processors, the most common method of use is to compile the functions with floating point data, using the run-time support library functions. Once the application is running correctly the functions can be modified to use the required fixed point data type. When using the library on fixed-point devices it is necessary to consider the fixed-point scaling issues and these are both application and device specific.

This applications note takes one specific function (SDA_complex_multiply_2) and shows how to modify it to support a 16 bit wordlength.

Another issue is the fact that the ANSI standard math functions (e.g. sin, cos etc.) all require floating point parameters and return floating point results. This applications note will also look at the scaling of the output of these functions.

Scaling Results

The problem with fixed point processing is that the wordlength of the data grows for each multiplication or adition operation. when performing a multiplication of two N bit numbers, the result will have a wordlength of 2*N. If the numbers are for example 16 bit signed numbers then the result will be a 32 bit number with 30 magnitude bits and 2 sign bits. When adding (or subtracting) two N bit numbers the result will be N+1 bits long.

The original function is shown in code section 1, where SFLOAT is typedef'd to type 'float' :

```
void SIGLIB_FUNC_DECL SDA_complex_multiply_2 (const SFLOAT *pSrc1_r,
      const SFLOAT *pSrc1_i,
      const SFLOAT *pSrc2_r,
      const SFLOAT *pSrc2_i,
      SFLOAT *pDst_r,
      SFLOAT *pDst_i,
      const SFIX SampleLength)

{
      register SFIX     i;
      register SFLOAT   R1, I1, R2, I2;

      for (i = 0; i < SampleLength; i++)
      {
            R1 = *pSrc1_r++;
            I1 = *pSrc1_i++;
            R2 = *pSrc2_r++;
            I2 = *pSrc2_i++;
            *pDst_r++ = (R1 * R2) - (I1 * I2);
            *pDst_i++ = (R1 * I2) + (I1 * R2);
      }

}           /* End of SDA_complex_multiply_2() */
```

Code Section 1. Original SigLib ANSI C Code

If SFLOAT is typedef'd to type 'short' then the function shown in code section 2 can be used and supports both addition and multiplication scaling :

```
void SIGLIB_FUNC_DECL SDA_complex_multiply_2 (const SFLOAT *pSrc1_r,
      const SFLOAT *pSrc1_i,
      const SFLOAT *pSrc2_r,
      const SFLOAT *pSrc2_i,
      SFLOAT *pDst_r,
      SFLOAT *pDst_i,
      const SFIX SampleLength)

{
      register SFIX     i;
      register SFLOAT   R1, I1, R2, I2;

      for (i = 0; i < SampleLength; i++)
      {
            R1 = *pSrc1_r++;
            I1 = *pSrc1_i++;
            R2 = *pSrc2_r++;
            I2 = *pSrc2_i++;
            *pDst_r++ = (((R1 * R2) >> 15) - ((I1 * I2) >> 15)) >> 1;
            *pDst_i++ = (((R1 * I2) >> 15) + ((I1 * R2) >> 15)) >> 1;
      }

}           /* End of SDA_complex_multiply_2() */
```

Code Section 2. Fixed Point SigLib ANSI C Code

ANSI Standard Math Functions

The ANSI standard math functions, include sin, cos, tan, sqrt and many more cause several problems when using them on floating point devices. Not only do they require floating point parameters but also return floating point results and they are often very inefficient at run time.

It is possible to overcome these problems by casting and scaling the source and result data types. The code loop in Code Section 3 is the floating point code to generate a ¾ sine FFT coefficient table and its corresponding fixed point version is shown in Code Section 4.

```
for (i = 0; i < (3 * FFT_SIZE) / 4; i++)
{
  pFFTCoefficients[i] = sin (((double)i) * (2.0 * PI / FFT_SIZE));
}
```

Code Section 3. Floating Point C Code For Initialising FFT Coefficient Table

```
for (i = 0; i < (3 * FFT_SIZE) / 4; i++)
{
  pFFTCoefficients[i] = (short)(32767.0 * sin (((double)i) * \
                                        (2.0 * PI / FFT_SIZE)));
}
```

Code Section 4. Fixed Point C Code For Initialising FFT Coefficient Table

The solution to the performance issue is to use look up tables for the resultant data so that these can be accessed at run-time.