FFT Routines for the C8051F12x Family

Relevant Devices

This application note applies to the following devices: C8051F124, C8051F125, C8051F126, and C8051F127.

Introduction

The Fast Fourier Transform (FFT) is an efficient method for calculating the Discrete Fourier Transform (DFT) of a signal. This note provides a brief introduction to the FFT, and describes two example FFT routines written in 'C' that have been optimized for execution time and RAM storage space on Silicon Labs microcontrollers. The example routines use the 10 or 12-bit ADC to collect the input data for the FFT routine, and the results are sent out through the UART, where they can be displayed using terminal software on a PC.

Only the very basic aspects of the FFT that are necessary to describe the algorithms are presented here. A more detailed explanation of the DFT and the FFT can be found in References [1] and [2].

Radix-2 FFT Algorithms

The output of the FFT is identical to the output of the DFT, but a number of redundant calculations have been eliminated to allow for faster computation. For an N-point DFT, the required number of complex multiplications is N^2. For an N-point FFT, the number of complex multiplications required is:

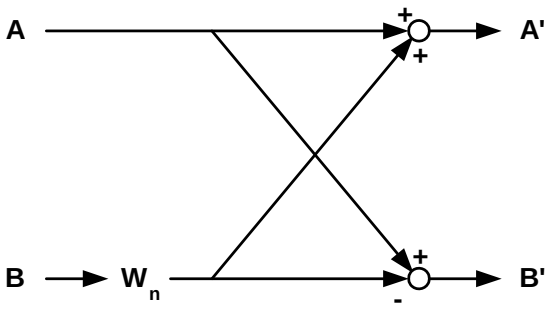
$$\frac{N}{2} \cdot \log_2 N$$

This optimization leads to a drastic speed improvement over the DFT as N becomes large. For example, a 64-point DFT requires 4096 complex multiplications, while the corresponding FFT requires only 192. Using this and other optimizations, an FFT can be calculated in a relatively short amount of time on a Silicon Labs 8051 processor.

The FFT routines presented in this note are both Radix-2 Decimation-in-Time algorithms. Radix-2 algorithms operate by separating the original DFT into a number of 2-point DFT computations. First, the original N-point DFT is split into two DFTs of N/2 points each. The resulting N/2-point DFTs are then each split into two N/4-point DFTs, and so on, until the number of points in each smaller DFT is reduced to two. This method requires that the FFT size be a power of two.

The basic 2-point DFT performed in the Radix-2 Decimation-in-Time algorithm is shown in Figure 1. This structure, named a "butterfly", is used to perform all of the computations necessary for the FFT. The inputs (A and B) and the outputs (A' and B') of the butterfly are complex numbers containing the data that is being processed. W_n represents a complex sinusoidal value that is applied at each stage of the FFT.

Figure 1. Radix-2 Decimation-in-Time Butterfly Structure



Index Bit Reversal

The FFT algorithms presented here are performed on the data in-place, to minimize the amount of temporary storage space required for intermediate data. To perform these algorithms in-place, either the input data or the output data of the FFT routine will be sorted in bit reversed order. To change between normal order and bit reversed order, each data point is swapped with another location in the data set determined by reversing the order of the bits in the sample index. For example, in a 16-point FFT, the sample stored at index 0001b (1 decimal) would swap locations with the sample stored at index 1000b (8 decimal). Locations where the bit reversed index are equal to the not bit-reversed index, such as 0110b (6 decimal) are not swapped.

The order of operations in the FFT computation is determined by whether the inputs or the outputs of the FFT are sorted in bit-reversed order.

Windowing Input Data

The FFT algorithm operates on a data set that represents a finite length of time, but makes the assumption that the data set is periodic and repeated infinitely. When the sample set is repeated in this way, the last sample (index [N-1]) is adjacent to the first sample (index [0]). As shown in Figure 2, this can lead to a discontinuity in the signal that the FFT “sees” when the data is not periodic over the sample set. Because of this, data is normally windowed before it is processed by an FFT routine. Windowing makes the data periodic over the sample set and removes any discontinuity between the first and last samples in the set.

Because windowing changes the input data set, it produces some artifacts in the frequency domain. Windowing “spreads” the signal energy among multiple bins, as shown in Figure 3. This energy spreading has the effect of attenuating the peak value of the signal. Most of the signal’s original content is stored in the “main lobe”, while a small amount leaks into the “side lobes”. The width of the main lobe and the height of the side lobes are dependent on the window algorithm that is applied to the data. Some common windows and their properties are summarized in Table 1 . Some equations for computing the window coefficients for an N-point FFT are listed in Table 2 . More extensive information on window algorithms and their parameters can be found in Reference [3].

Figure 2. Time Domain Windowing

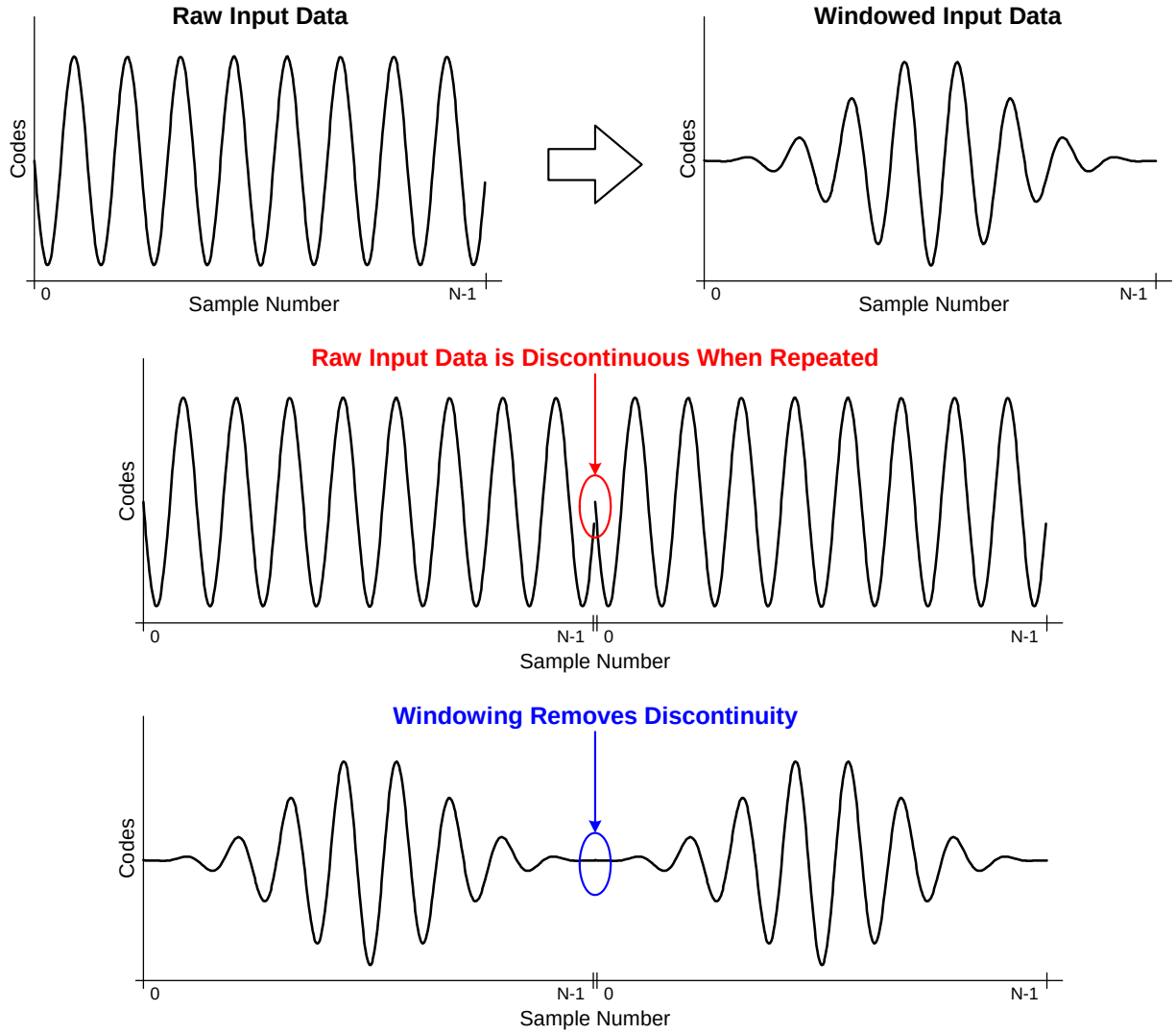


Figure 3. Window Effects in the Frequency Domain

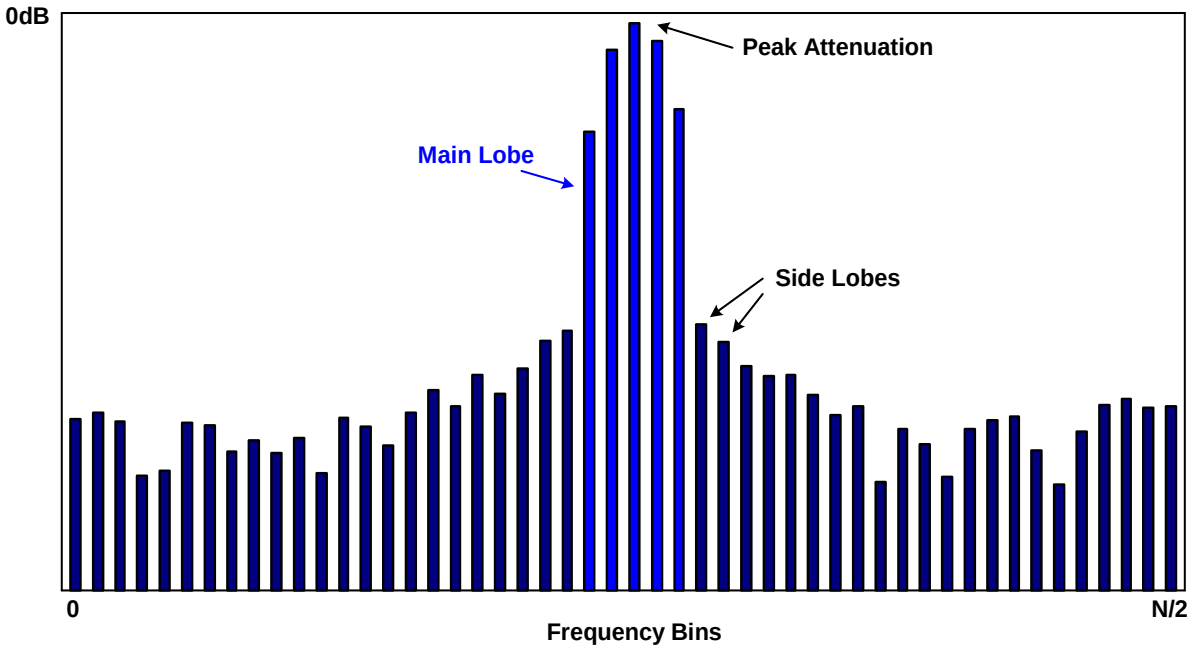


Table 1. Window Properties

Window Type	Main Lobe Width	Processing Gain	Side Lobe Height
None (Rectangular)	1 Bin	1.0	-13 dB
Triangular	3 Bins	0.5	-27 dB
Hanning	3 Bins	0.5	-32 dB
Hamming	3 Bins	0.54	-43 dB
Blackman	5 Bins	0.42	-58 dB

Table 2. Window Coefficient Equations

Window Type	Window Equation	
None (Rectangular)	$W(n) = 1$	$0 \le n < N$
Triangular	$W(n) = n / (N / 2)$ $W(n) = 2 - n / (N / 2)$	$0 \le n \le N / 2$ $N / 2 < n < N$
Hanning	$W(n) = 0.5 - 0.5\cos(2\pi n / N)$	$0 \le n < N$
Hamming	$W(n) = 0.54 - 0.46\cos(2\pi n / N)$	$0 \le n < N$
Blackman	$W(n) = 0.42 - 0.5\cos(2\pi n / N) + 0.08\cos(4\pi n / N)$	$0 \le n < N$

Software Examples

This application note contains two FFT routines that are very similar. Both examples collect data from the ADC, window the collected data, perform an FFT, and print the results to the UART running at 115200 baud. The difference between the two example files is whether the inputs or the outputs of the FFT routine are in bit-reversed order. The first routine, "IntFFT_BRIN.c", accepts input data sorted in bit-reversed order and produces output sorted in normal order. The second routine, "IntFFT_BROUT.c", accepts data sorted in normal order and produces output sorted in bit-reversed order.

The two examples produce identical results, but each has its advantages. Performing a bit-reversal sort takes some processing time and is not always necessary for every application. For example, an application that is performing peak frequency detection needs only to know which output bin of the FFT has the highest magnitude. Once this is determined, the frequency for that particular bin can be calculated by bit reversing only the bin of interest. In this example, the routine that accepts input data in normal order and produces outputs in bit-reversed order (IntFFT_BROUT.c) can be used without having to re-sort the output data afterwards.

If the application requires that the output of the FFT appears in normal order, and the input to the routine is a real signal (not a complex input), the routine that requires bit-reversed input data (IntFFT_BRIN.c) may be a better choice. The reason for this is that the imaginary locations in the real input set can be assumed to be zero, and some processor time can be saved by only sorting the real data. In this example, the complex output of the FFT appears in normal order.

Parameter Specification

The file "FFT_Code_Tables.h" contains code tables for FFT sizes from 4 points to 1024 points, and four different window types. Conditional compilation is used so that only the necessary tables for the desired FFT size and window type are included. The parameter *NUM_FFT* should contain the size of the FFT to perform. Valid FFT sizes for this example code are 4, 8, 16, 32, 64, 128, 256, 512, and 1024 points. The parameter *WINDOW_TYPE* should contain a number from 0 to 4, which specifies the desired window algorithm, according to Table 3 .

Table 3. Window Selection

WINDOW_TYPE	Window Used
0	None (Rectangular)
1	Triangular
2	Hanning
3	Hamming
4	Blackman

One other parameter that affects the program operation is the *RUN_ONCE* definition, which is contained in the main source file. If *RUN_ONCE* is zero, the program will continue to collect new data sets and perform FFTs on them. If *RUN_ONCE* is a non-zero value, the program will stop after one iteration.

Data Collection

ADC0 collects the data to process using Timer 3 as a start-of-conversion source. ADC0 is configured to sample at a speed determined by the *SAMPLE_RATE* constant, in single-ended mode on channel AIN0.0. Samples are collected and stored in the *Real[]* array. The 12-bit ADC data is left-justified and stored as 16-bit data with trailing zeros. Once *NUM_FFT* samples have been collected, ADC0 interrupts are turned off, and the data is windowed.

Window Implementation

After the data has been collected, it is windowed using the window selected by the *WINDOW_TYPE* constant (see Table 3). The window information is stored in code space as a table of unsigned integer (16-bit) values. The integers represent a fractional number which can be computed by dividing the stored integer by 65536. For example, the value 32768 represents a multiplication value of 32768 / 65536, or 0.5. Because the windows are all symmetrical about *NUM_FFT*/2, the window tables only contain values for one half of the window to save storage space. The window value at index *NUM_FFT*/2 is assumed to be equal to 1.0, and is not stored. The *WindowCalc()* function performs a multiplication of each input sample with its corresponding value in the window table. Table 4 shows how the window table is indexed when multiplying.

Table 4. Window Index Decoding

Input Sample Index (M)	Window Table Index
0 to (N / 2 - 1)	M
N / 2	No Multiplication
(N / 2 + 1) to (N -1)	N - M
N is represented by *NUM_FFT* in the software examples.	

The *WindowCalc()* function also changes data that has been stored in single-ended (unsigned) format into differential (signed 2's complement) format. This centers the data about 0x0000 to remove the DC bias. When the *SE_data* input variable is non-zero, each sample is XORed with the value 0x8000. This inverts the MSB of the data, which has the same effect as subtracting 0x8000 from all samples. If the *SE_data* variable is zero, the data is assumed to already be in 2's complement format, and is not changed.

FFT Optimizations

A number of optimizations were implemented in the example routines. The primary goals of these optimizations were to maximize the speed of the routine and to minimize the amount of RAM needed for data storage. The specifics of each optimization are detailed in the sections that follow.

Integer Storage and Computation

The FFT algorithm has an inherent processing gain. For a real input, the processing gain is equal to N/2, where N is the number of points in the FFT (the processing gain for complex inputs is equal to N). Because of this gain, more bits are required to store the output information of the FFT than are required to store the input information. Instead of using additional space, the example FFT routines use 16-bit signed integer values to store all input, output, and intermediate data. To compensate for the FFT processing gain, and to store all data as 16-bit numbers, the computed values are divided by 2 after each butterfly calculation. Using this method, the overall gain of the FFT routine, not including window gain, is 1/2 for a real input, and 1 for a complex input.

Integer math is used to perform all calculations. This improves the speed of the FFT routine over fixed-point or floating-point math. Multiplication is avoided whenever possible, and all divide operations are implemented using right shifts. Whenever this type of divide is performed, the result is rounded to counter the asymmetrical effects of truncating a 2's complement number. If the number is negative, and the deleted bits are non-zero, a one is added to the result so that both negative and positive numbers are always rounded towards zero.

Sinusoid Table Storage

To reduce computation time, sine and cosine values are not calculated in real-time. Instead, they are pre-calculated and stored in code space. The *SinTable[]* array declared in "FFT_Code_Tables.h"

contains the data for 1⁄4 period of a sine function. The values stored in the *SinTable[]* array are 2's complement integer (16-bit) values. The integers represent fractional numbers between -1.0 and 1.0 (though negative values and 1.0 are not stored). The actual multiplication value can be computed by dividing the integer value by 32768. For example, a value of 8192 in the *SinTable[]* array represents a multiplication value of 8192⁄32768, or 0.25. The FFT routine needs sine and cosine values for 1⁄2 of the sinusoid period. The 1⁄4 sine wave stored in code space is indexed according to Table 5 to generate the necessary values.

Table 5. Sine Table Index Decoding

Condition	Sine Value	Cosine Value
M = 0	0	1
0 < M < N / 4	*SinTable[M]*	*SinTable[(N / 4) - M]*
M = N / 4	1	0
N / 4 < M < N / 2	*SinTable[(N / 2) - M]*	*SinTable[M - (N / 4)]*
M and N are represented by the following parameters in the software examples: N = *NUM_FFT* M = sin_index		

Real Input Only

The example routines have been optimized to compute an FFT on a real input. During the first stage of the FFT, it is assumed that all imaginary locations are equal to zero, which eliminates a number of calculations. The real input is modified as needed, and the imaginary locations are set to zero. The FFT examples can easily be modified to operate on complex input data, as detailed in "Using Complex Inputs," on page 11.

Bit Reversal Sorting

Instead of fully computing a bit-reversed value for each index, the *Bit_Reverse()* function uses the *BRTable[]* array to look up the bit-reversed index locations. Bit-reversed index values for the first half of the indices (0 through *NUM_FFT* / 2 - 1) are stored in the table directly. Bit-reversed index values for the second half of the indices (*NUM_FFT* / 2 through *NUM_FFT* - 1) are calculated by adding 1 to the table value stored at Index - *NUM_FFT* / 2. For example, the bit-reversed value for index *NUM_FFT* / 2 is calculated as:

BRTable[*NUM_FFT* / 2 - *NUM_FFT* / 2] + 1.

Unnecessary Multiplications

Each iteration of the butterfly calculation requires one complex multiplication (a total of four long integer multiplications). Long integer multiplications take many processor cycles to complete. Many of these multiplications are not necessary and can be eliminated. Specifically, when multiplying by zero, the result will be zero, and when multiplying by one, the result will be the original number. The example code checks for the cases where the sine and cosine values are equal to zero or one, and takes these shortcuts when it can. The

number of complex multiplications saved with this optimization is equal to:

$$\frac{N}{2} + \sum_{i=1}^{\log_2(N)-1} 2^i$$

where N is the number of points in the FFT.

Performance

The approximate number of clock cycles the processor requires for each of the key routines in the example code is shown in Table 6 and Table 7 , for FFT sizes of 16, 64, 256, and 1024 points. For larger FFT sizes, the number of clock cycles required for the *WindowCalc()* and *BitReverse()* routines increases linearly, while the number of clock cycles required for the *IntFFT()* routine increases according to the number of complex multiplications required, as discussed in "Radix-2 FFT Algorithms," on page 1. The total execution time for these routines when the controller is operating with a 49.7664 MHz clock is shown in Table 8 .

Table 6. Approximate Timing in SYSCLK Cycles (IntFFT_BRIN.c)

FFT Size	WindowCalc()	Bit Reverse()	IntFFT()	Total
16	6,500	1,600	38,000	46,000
64	27,000	7,500	260,000	294,000
256	113,000	30,000	1.5 Million	1.64 Million
1024	452,000	145,000	7.8 Million	8.4 Million

Table 7. Approximate Timing in SYSCLK Cycles (IntFFT_BROUT.c)

FFT Size	WindowCalc()	Bit Reverse()	IntFFT()	Total
16	6,500	2,900	38,000	47,000
64	27,000	13,000	260,000	299,000
256	113,000	55,000	1.5 Million	1.66 Million
1024	452,000	244,000	7.8 Million	8.5 Million

Table 8. Total Time With a 49.7664 MHz System Clock

FFT Size	IntFFT_BRIN.c	IntFFT_BROUT.c
16	924 µs	945 µs
64	5.9 ms	6 ms
256	33 ms	33.4 ms
1024	169 ms	171 ms

Code and XRAM Size

Table 9 and Table 10 list the code space requirements for the key functions and code tables used in the example software. The total amount of XRAM required for the storage of the *Real[]* and *Imag[]* integer arrays is equal to the FFT size times 4. For example, a 1024-point FFT requires 4096 bytes (4 x 1024) of XRAM storage space.

Table 9. Code Space Requirements in Bytes (IntFFT_BRIN.c)

FFT Size	WindowCalc()	Bit Reverse()	IntFFT()	Code Tables	Total
4	560	195	2291	8	3054
8	560	197	2291	16	3064
16	560	197	2291	32	3080
32	560	197	2291	64	3112
64	560	197	2291	128	3176
128	560	197	2291	256	3304
256	617	197	2291	512	3617
512	619	234	2417	1024	4294
1024	619	252	2419	2560	5850

Table 10. Code Space Requirements in Bytes (IntFFT_BROUT.c)

FFT Size	WindowCalc()	Bit Reverse()	IntFFT()	Code Tables	Total
4	560	353	2292	8	3213
8	560	355	2292	16	3223
16	560	355	2292	32	3239
32	560	355	2292	64	3271
64	560	355	2292	128	3335
128	560	355	2292	256	3463
256	617	355	2292	512	3776
512	619	406	2426	1024	4475
1024	619	424	2437	2560	6040

Possible Modifications

The FFT routines can be expanded or optimized further depending on the application. The following sections describe a few of the possible modifications.

FFT Size

The example routines are limited to between 4 and 1024 samples with the included sinusoid, window, and bit reversal tables. Larger tables can be defined to allow the software to perform larger FFTs. As written, the routines will perform FFTs of up to 32768 points, provided that there is enough code space to store the tables and enough RAM to store the *Real[]* and *Imag[]* arrays.

Using Complex Inputs

The example sfotware is optimized for real input data only. With some minor changes, the FFT routines can be used to compute the FFT of complex data sets as well. The following things must be changed to perform FFTs of complex input data:

1. Remove or comment out the section of code labeled “FIRST STAGE” in the *Int_FFT()* routine.

2. For “IntFFT_BRIN.c”, the *Bit_Reverse()* routine should be changed to perform a bit reversal on both the Real and Imaginary data. The *Bit_Reverse()* routine in “IntFFT_BROUT.c” does the bit reversal sort on both arrays, and can be used here. Alternately, the *Bit_Reverse()* function in “IntFFT_BRIN.c” can be called twice: once to sort the *Real[]* array, and once to sort the *Imag[]* array.

3. For “IntFFT_BRIN.c”, the variable *group* in the function *Int_FFT()* should be initialized to ‘1’, and the variable *stage* should be initialized to *NUM_FFT*/2.

4. For “IntFFT_BROUT.c”, the variable *group* in the function *Int_FFT()* should be initialized to *NUM_FFT*/2, and the variable *stage* should be initialized to ‘1’.

5. If the data is to be windowed, the function *WindowCalc()* must be modified to window the Imaginary data as well as the Real data. Alternately, the *WindowCalc()* function can be called twice: once to window the *Real[]* array, and once to window the *Imag[]* array.

Increasing Output Precision

A more involved software modification would entail changing the way that data is computed and stored. To help maximize the speed of the FFT, the routines presented here store data as 16-bit integer values. This limits the output precision of the FFT. If larger variables or floating-point numbers are used to store the data, the output precision of the routine can be increased to make it more suitable for measurements such as Signal-to-Noise or Signal-to-Distortion.

References

[1]Lyons, R. G. *Understanding Digital Signal Processing*, Addison Wesley Longman, Inc., Reading, Massachusetts, 1997, pp. 49-154.

[2]Oppenheim, A. V. and Schafer, R. W. *Discrete-Time Signal Processing*, 2nd Ed. Prentice Hall, Inc., Upper Saddle River, New Jersey, 1999, pp. 541-669.

[3]Harris, F. J. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform,” *Proceedings of the IEEE*, Vol. 66, No. 1, January 1978.

IntFFT_BRIN.c

```
//-----------------------------------------------------------------------------
// IntFFT_BRIN.c
//-----------------------------------------------------------------------------
// Copyright 2003 Cygnal Integrated Products, Inc.
//
// AUTH: BD
// DATE: 30 JAN 03
//
// This program collects data using ADC0 at <SAMPLE_RATE> Hz and performs
// an FFT on the data.  The Real and Imaginary parts of the results are then
// sent to the UART peripheral at <BAUDRATE> bps, where they can be displayed
// or captured using a terminal program.
//
// Note that the FFT performed in this software is optimized for storage space
// (RAM).  The resulting Frequency-domain data is not suitable for analyzing
// Signal-to-noise or distortion performance.
//
// This program uses a 22.1184 MHz crystal oscillator multiplied by (9/4)
// for an effective SYSCLK of 49.7664 Mhz. This program also initializes and
// uses UART0 at <BAUDRATE> bits per second.
//
// Target: C8051F12x
// Tool chain: KEIL C51 6.03
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f120.h>                 // SFR declarations
#include <stdio.h>

#include “FFT_Code_Tables.h”           // Code Tables for FFT routines

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for ‘F12x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                 // data pointer
sfr16 ADC0     = 0xbe;                 // ADC0 data
sfr16 ADC0GT   = 0xc4;                 // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                 // ADC0 less than window
sfr16 RCAP2    = 0xca;                 // Timer2 capture/reload
sfr16 RCAP3    = 0xca;                 // Timer3 capture/reload
sfr16 RCAP4    = 0xca;                 // Timer4 capture/reload
sfr16 TMR2     = 0xcc;                 // Timer2
sfr16 TMR3     = 0xcc;                 // Timer3
sfr16 TMR4     = 0xcc;                 // Timer4
sfr16 DAC0     = 0xd2;                 // DAC0 data
sfr16 DAC1     = 0xd2;                 // DAC1 data
sfr16 PCA0CP5  = 0xe1;                 // PCA0 Module 5 capture
sfr16 PCA0CP2  = 0xe9;                 // PCA0 Module 2 capture
sfr16 PCA0CP3  = 0xeb;                 // PCA0 Module 3 capture
sfr16 PCA0CP4  = 0xed;                 // PCA0 Module 4 capture
sfr16 PCA0     = 0xf9;                 // PCA0 counter
sfr16 PCA0CP0  = 0xfb;                 // PCA0 Module 0 capture
sfr16 PCA0CP1  = 0xfd;                 // PCA0 Module 1 capture
```

```
//-----------------------------------------------------------------------------
// Global CONSTANTS and Variable Type Definitions
//-----------------------------------------------------------------------------
#define NUM_BITS 16                     // Number of Bits in Data

#define DATA_BEGIN 0x0000               // Beginning of XRAM Data

#define EXTCLK       22118400           // External oscillator frequency in Hz
#define SYSCLK       49760000           // Output of PLL derived from
                                        // (EXTCLK*9/4)
#define BAUDRATE     115200             // Baud Rate for UART0

#define SAMPLE_RATE  10000              // Sample frequency in Hz

#define RUN_ONCE     1                  // Setting to a non-zero value will
                                        // cause the program to stop after one
                                        // data set.


typedef union IBALONG {                 // Integer or Byte-addressable LONG
   long l;                              // long: Var.l
   unsigned int i[2];                   // u int: Var.i[0]:Var.i[1]
   unsigned char b[4];                  // u char: Var.b[0]:Var.b[1]:
                                        //          Var.b[2]:Var.b[3]
} IBALONG;

typedef union BAINT {                   // Byte-addressable INT
   int i;                               // int: Var.i
   unsigned char b[2];                  // u char: Var.b[0]:Var.b[1]
} BAINT;

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void WindowCalc(int Win_Array[], unsigned char SE_data);
void Int_FFT(int ReArray[], int ImArray[]);
void Bit_Reverse(int BR_Array[]);

void SYSCLK_Init (void);
void PORT_Init (void);
void UART0_Init (void);
void ADC0_Init (void);
void TIMER3_Init (int counts);
void ADC0_ISR (void);


//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------

// XRAM storage of FFT: requires NUM_FFT*4 Bytes after DATA_BEGIN address
int xdata Real[NUM_FFT] _at_ DATA_BEGIN;
int xdata Imag[NUM_FFT] _at_ (DATA_BEGIN + (NUM_FFT * 2));

// NUM_FFT is defined in the “FFT_Code_Tables.h” header file
#if (NUM_FFT >= 256)
unsigned int index, ADC_Index;
```

```
#endif

#if (NUM_FFT < 256)
unsigned char index, ADC_Index;
#endif

unsigned int BinNum;

bit Conversion_Set_Complete;           // This indicates when the data has been
                                       // stored, and is ready to be processed
                                       // using the FFT routines

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main()
{

   // disable watchdog timer
   WDTCN = 0xde;
   WDTCN = 0xad;


   SYSCLK_Init();                      // initialize external clock and PLL
   PORT_Init ();                       // set up Port I/O
   UART0_Init ();                      // initialize UART0
   TIMER3_Init (SYSCLK/SAMPLE_RATE);   // initialize Timer3 to overflow at
                                       // <SAMPLE_RATE>
   ADC0_Init ();                       // init ADC0

   EA = 1;                             // globally enable interrupts

   while (1)
   {
      ADC_Index = 0;
      Conversion_Set_Complete = 0;

      EIE2 |= 0x02;                    // enable ADC interrupts

      SFRPAGE = LEGACY_PAGE;

      while(!Conversion_Set_Complete);


      SFRPAGE = UART0_PAGE;
      printf("\nCollected Data\nSample\tValue\n");
      for (BinNum = 0; BinNum < NUM_FFT; BinNum++)
      {
         // Print Data in the format: Sample <tab> Value <tab>
         printf("%d\t%u\n", BinNum, Real[BinNum]);
      }

      WindowCalc(Real, 1);             // Window Real Data, and convert to
                                       // differential if it is single-ended

      Bit_Reverse(Real);               // Sort Real (Input) Data in bit-reverse
                                       // order
      Int_FFT(Real, Imag);             // Perform FFT on data
```

```
      SFRPAGE = UART0_PAGE;
      printf("\nBin\tReal\tImag\n");

      // Output the FFT data to the UART
      for (BinNum = 0; BinNum < NUM_FFT; BinNum++)
      {
         // Print Data in the format: Bin <tab> Real <tab> Imaginary
         printf("%d\t%d\t%d\n", BinNum, Real[BinNum], Imag[BinNum]);
      }

      if (RUN_ONCE)
         while(1);

   }

} // END MAIN


//-------------------------------------------------------------------------
// WindowCalc
//-------------------------------------------------------------------------
//
// Uses the values in WindowFunc[] to window the stored data.
//
// The WindowFunc[] Array contains window coefficients between samples
// 0 and (NUM_FFT/2)-1, and samples from NUM_FFT/2 to NUM_FFT-1 are the mirror
// image of the other side.
// Window values are interpreted as a fraction of 1 (WindowFunc[x]/65536).
// The value at NUM_FFT/2 is assumed to be 1.0 (65536).
//
// If SE_data = 1, the input data is assumed to be single-ended, and is
// converted to a 2's complement, differential representation, to cancel the DC
// offset.
//
void WindowCalc(int Win_Array[], unsigned char SE_data)
{

#if (WINDOW_TYPE != 0)  // Use this section if a window has been specified
IBALONG NewVal;

   if (SE_data)                              // If data is single-ended,
      Win_Array[0] ^= 0x8000;                // convert it to differential
   NewVal.l = (long)Win_Array[0] * WindowFunc[0];
   if ((NewVal.l < 0)&&(NewVal.i[1]))
     Win_Array[0] = NewVal.i[0] + 1;
   else Win_Array[0] = NewVal.i[0];

   if (SE_data)                              // If data is single-ended,
      Win_Array[NUM_FFT/2] ^= 0x8000;        // convert it to differential

  for (index = 1; index < NUM_FFT/2; index++)
  {
      // Array positions 1 to (NUM_FFT/2 - 1)
      if (SE_data)                           // If data is single-ended,
         Win_Array[index] ^= 0x8000;         // convert it to differential
      NewVal.l = (long)Win_Array[index] * WindowFunc[index];
      if ((NewVal.l < 0)&&(NewVal.i[1]))
        Win_Array[index] = NewVal.i[0] + 1;
```

```
      else Win_Array[index] = NewVal.i[0];

      // Array positions (NUM_FFT/2 + 1) to (NUM_FFT - 1)
      if (SE_data)                          // If data is single-ended,
         Win_Array[NUM_FFT-index] ^= 0x8000; // convert it to differential
      NewVal.l = (long)Win_Array[NUM_FFT-index] * WindowFunc[index];
      if ((NewVal.l < 0)&&(NewVal.i[1]))
         Win_Array[NUM_FFT-index] = NewVal.i[0] + 1;
      else Win_Array[NUM_FFT-index] = NewVal.i[0];

   }

#endif

#if (WINDOW_TYPE == 0)  // Compile this if no window has been specified

   if (SE_data)                             // If data is single-ended,
   {                                        // convert it to differential

      for (index = 0; index < NUM_FFT; index++)
      {
         Win_Array[index] ^= 0x8000;        // XOR MSB with '1' to invert
      }
   }

#endif

}  // END WindowCalc


//---------------------------------------------------------------------------
// Bit_Reverse
//---------------------------------------------------------------------------
//
// Sorts data in Bit Reversed Address order
//
// The BRTable[] array is used to find which values must be swapped.  Only
// half of this array is stored, to save code space.  The second half is
// assumed to be a mirror image of the first half.
//
void Bit_Reverse(int BR_Array[])
{

#if (NUM_FFT >= 512)
unsigned int swapA, swapB, sw_cnt;          // Swap Indices
#endif

#if (NUM_FFT <= 256)
unsigned char swapA, swapB, sw_cnt;         // Swap Indices
#endif

int TempStore;

   // Loop through locations to swap
   for (sw_cnt = 1; sw_cnt < NUM_FFT/2; sw_cnt++)
   {
      swapA = sw_cnt;                       // Store current location
      swapB = BRTable[sw_cnt] * 2;          // Retrieve bit-reversed index
      if (swapB > swapA)                    // If the bit-reversed index is
```

```
      {                                    // larger than the current index,
         TempStore = BR_Array[swapA];      // the two data locations are
         BR_Array[swapA] = BR_Array[swapB];// swapped. Using this comparison
         BR_Array[swapB] = TempStore;      // ensures that locations are only
      }                                    // swapped once, and never with
                                           // themselves

      swapA += NUM_FFT/2;                  // Now perform the same operations
      swapB++;                             // on the second half of the data
      if (swapB > swapA)
      {
         TempStore = BR_Array[swapA];
         BR_Array[swapA] = BR_Array[swapB];
         BR_Array[swapB] = TempStore;
      }
   }

}  // END Bit Reverse Order Sort


//-------------------------------------------------------------------------
// Int_FFT
//-------------------------------------------------------------------------
//
// Performs a Radix-2 Decimation-In-Time FFT on the input array ReArray[]
//
// During each stage of the FFT, the values are calculated using a set of
// “Butterfly” equations, as listed below:
//
// Re1 = Re1 + (Cos(x)*Re2 + Sin(x)*Im2)
// Re2 = Re1 - (Cos(x)*Re2 + Sin(x)*Im2)
// Im1 = Im1 + (Cos(x)*Im2 - Sin(x)*Re2)
// Im2 = Im1 - (Cos(x)*Im2 - Sin(x)*Re2)
//
// The routine implements this calculation using the following values:
//
// Re1 = ReArray[indexA], Re2 = ReArray[indexB]
// Im1 = ImArray[indexA], Im2 = ImArray[indexB]
// x = the angle: 2*pi*(sin_index/NUM_FFT), in radians.  The necessary values
//    are stored in code space in the SinTable[] array.
//
//
// Key Points for using this FFT routine:
//
// 1) It expects REAL data (in ReArray[]), in 2’s complement, 16-bit binary
//    format and assumes a value of 0 for all imaginary locations
//    (in ImArray[]).
//
// 2) It expects the REAL input data to be sorted in bit-reversed index order.
//
// 3) SIN and COS values are retrieved and calculated from a table consisting
//    of 1/4 of a period of a SIN function.
//
// 4) It is optimized to use integer math only (no floating-point operations),
//    and for storage space.  The input, all intermediate stages, and the
//    output of the FFT are stored as 16-bit INTEGER values. This limits the
//    precision of the routine.  When using input data of less than 16-bits,
//    the best results are produced by left-justifying the data prior to
//    windowing and performing the FFT.
```

```
//
// 5) The algorithm is a Radix-2 type, meaning that the number of samples must
//    be 2^N, where N is an integer.  The minimum number of samples to process
//    is 4.  The constant NUM_FFT contains the number of samples to process.
//
//
void Int_FFT(int ReArray[], int ImArray[])
{

#if (NUM_FFT >= 512)
unsigned int sin_index, g_cnt, s_cnt;        // Keeps track of the proper index
unsigned int indexA, indexB;                 // locations for each calculation
#endif

#if (NUM_FFT <= 256)
unsigned char sin_index, g_cnt, s_cnt;       // Keeps track of the proper index
unsigned char indexA, indexB;                // locations for each calculation
#endif

unsigned int group = NUM_FFT/4, stage = 2;
long CosVal, SinVal;
long TempImA, TempImB, TempReA, TempReB, TempReA2, TempReB2;
IBALONG ReTwid, ImTwid, TempL;

// FIRST STAGE - optimized for REAL input data only.  This will set all
// Imaginary locations to zero.
//
// Shortcuts have been taken to remove unnecessary multiplications during this
// stage. The angle “x” is 0 radians for all calculations at this point, so
// the SIN value is equal to 0.0 and the COS value is equal to 1.0.
// Additionally, all Imaginary locations are assumed to be ‘0’ in this stage of
// the algorithm, and are set to ‘0’.

   indexA = 0;
   for (g_cnt = 0; g_cnt < NUM_FFT/2; g_cnt++)
   {
      indexB = indexA + 1;

      TempReA = ReArray[indexA];
      TempReB = ReArray[indexB];

      // Calculate new value for ReArray[indexA]
      TempL.l = (long)TempReA + TempReB;
      if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
         TempReA2 = (TempL.l >> 1) + 1;
      else TempReA2 = TempL.l >> 1;

      // Calculate new value for ReArray[indexB]
      TempL.l = (long)TempReA - TempReB;
      if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
         ReArray[indexB] = (TempL.l >> 1) + 1;
      else ReArray[indexB] = TempL.l >> 1;

      ReArray[indexA] = TempReA2;

      ImArray[indexA] = 0;                   // set Imaginary locations to ‘0’
      ImArray[indexB] = 0;

      indexA = indexB + 1;
```

```
   }

// END OF FIRST STAGE


while (stage <= NUM_FFT/2)
{
   indexA = 0;
   sin_index = 0;

   for (g_cnt = 0; g_cnt < group; g_cnt++)
   {
      for (s_cnt = 0; s_cnt < stage; s_cnt++)
      {
         indexB = indexA + stage;

         TempReA = ReArray[indexA];
         TempReB = ReArray[indexB];
         TempImA = ImArray[indexA];
         TempImB = ImArray[indexB];

// The following first checks for the special cases when the angle “x” is
// equal to either 0 or pi/2 radians.  In these cases, unnecessary
// multiplications have been removed to improve the processing speed.

         if (sin_index == 0)  // corresponds to “x” = 0 radians
         {

            // Calculate new value for ReArray[indexA]
            TempL.l = (long)TempReA + TempReB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
               TempReA2 = (TempL.l >> 1) + 1;
            else TempReA2 = TempL.l >> 1;

            // Calculate new value for ReArray[indexB]
            TempL.l = (long)TempReA - TempReB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
               TempReB2 = (TempL.l >> 1) + 1;
            else TempReB2 = TempL.l >> 1;

            // Calculate new value for ImArray[indexB]
           TempL.l = (long)TempImA - TempImB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
               TempImB = (TempL.l >> 1) + 1;
            else TempImB = TempL.l >> 1;

            // Calculate new value for ImArray[indexA]
            TempL.l = (long)TempImA + TempImB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
               TempImA = (TempL.l >> 1) + 1;
            else TempImA = TempL.l >> 1;

         }
         else if (sin_index == NUM_FFT/4) // corresponds to “x” = pi/2 radians
         {

            // Calculate new value for ReArray[indexB]
            TempL.l = (long)TempReA - TempImB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
```

```
      TempReB2 = (TempL.l >> 1) + 1;
   else TempReB2 = TempL.l >> 1;

   // Calculate new value for ReArray[indexA]
   TempL.l = (long)TempReA + TempImB;
   if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
      TempReA2 = (TempL.l >> 1) + 1;
   else TempReA2 = TempL.l >> 1;

   // Calculate new value for ImArray[indexB]
   TempL.l = (long)TempImA + TempReB;
   if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
      TempImB = (TempL.l >> 1) + 1;
   else TempImB = TempL.l >> 1;

   // Calculate new value for ImArray[indexA]
   TempL.l = (long)TempImA - TempReB;
   if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
      TempImA = (TempL.l >> 1) + 1;
   else TempImA = TempL.l >> 1;

}

else
{
   // If no multiplication shortcuts can be taken, the SIN and COS
   // values for the Butterfly calculation are fetched from the
   // SinTable[] array.

   if (sin_index > NUM_FFT/4)
   {
      SinVal = SinTable[(NUM_FFT/2) - sin_index];
      CosVal = -SinTable[sin_index - (NUM_FFT/4)];
   }
   else
   {
      SinVal = SinTable[sin_index];
      CosVal = SinTable[(NUM_FFT/4) - sin_index];
   }

   // The SIN and COS values are used here to calculate part of the
   // Butterfly equation
   ReTwid.l = ((long)TempReB * CosVal) +
         ((long)TempImB * SinVal);

   ImTwid.l = ((long)TempImB * CosVal) -
         ((long)TempReB * SinVal);

   // Using the values calculated above, the new variables
   // are computed

   // Calculate new value for ReArray[indexA]
   TempL.i[1] = 0;
   TempL.i[0] = TempReA;
   TempL.l = TempL.l >> 1;
   ReTwid.l += TempL.l;
   if ((ReTwid.l < 0)&&(ReTwid.i[1]))
      TempReA2 = ReTwid.i[0] + 1;
   else TempReA2 = ReTwid.i[0];
```

```
            // Calculate new value for ReArray[indexB]
            TempL.l = TempL.l << 1;
            TempL.l -= ReTwid.l;
            if ((TempL.l < 0)&&(TempL.i[1]))
               TempReB2 = TempL.i[0] + 1;
            else TempReB2 = TempL.i[0];

            // Calculate new value for ImArray[indexA]
            TempL.i[1] = 0;
            TempL.i[0] = TempImA;
            TempL.l = TempL.l >> 1;
            ImTwid.l += TempL.l;
            if ((ImTwid.l < 0)&&(ImTwid.i[1]))
               TempImA = ImTwid.i[0] + 1;
            else TempImA = ImTwid.i[0];

            // Calculate new value for ImArray[indexB]
            TempL.l = TempL.l << 1;
            TempL.l -= ImTwid.l;
            if ((TempL.l < 0)&&(TempL.i[1]))
               TempImB = TempL.i[0] + 1;
            else TempImB = TempL.i[0];

         }

         ReArray[indexA] = TempReA2;
         ReArray[indexB] = TempReB2;
         ImArray[indexA] = TempImA;
         ImArray[indexB] = TempImB;

         indexA++;
         sin_index += group;
      }                                    // END of stage FOR loop (s_cnt)
      indexA = indexB + 1;
      sin_index = 0;
   }                                       // END of group FOR loop (g_cnt)

   group /= 2;
   stage *= 2;
}                                          // END of While loop

}  // END Int_FFT


//-----------------------------------------------------------------------------
// Initialization Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports
//
void PORT_Init (void)
{
char old_SFRPAGE = SFRPAGE;         // Store current SFRPAGE
```

```
   SFRPAGE = CONFIG_PAGE;           // Switch to configuration page


   XBR0    = 0x04;                  // Enable UART0 on crossbar
   XBR1    = 0x00;
   XBR2    = 0x40;                  // Enable crossbar and weak pull-ups
   P0MDOUT |= 0x01;                 // Set TX0 pin to push-pull

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
}

//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <baudrate> and 8-N-1. In order to
// increase the clocking flexibility of Timer0, Timer1 is configured to count
// SYSCLKs.
//
// To use this routine SYSCLK/BAUDRATE/16 must be less than 256. For example,
// if SYSCLK = 50 MHz, the lowest standard baud rate supported by this
// routine is 19,200 bps.
void UART0_Init (void)
{
char old_SFRPAGE = SFRPAGE;         // Store current SFRPAGE

   SFRPAGE = UART0_PAGE;            // Switch to UART0 Page

   SCON0  = 0x50;                   // SCON0: mode 0, 8-bit UART, enable RX
   SSTA0  = 0x10;                   // Timer 1 generates UART0 baud rate and
                                    // UART0 baud rate divide by two disabled
   SFRPAGE = TIMER01_PAGE;
   TMOD   &= ~0xF0;
   TMOD   |=  0x20;                 // TMOD: timer 1, mode 2, 8-bit reload

   TH1 = -(SYSCLK/BAUDRATE/16);     // Set the Timer1 reload value
                                    // When using a low baud rate, this
                                    // equation should be checked to ensure
                                    // that the reload value will fit in
                                    // 8-bits.

   CKCON |= 0x10;                   // T1M = 1; SCA1:0 = xx


   TL1 = TH1;                       // initialize Timer1
   TR1 = 1;                         // start Timer1

   SFRPAGE = UART0_PAGE;
   TI0 = 1;                         // Indicate TX0 ready

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE

}

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use an external 22.1184 MHz
```

```
// crystal oscillator multiplied by a factor of 9/4 using the PLL as its
// clock source. The resulting frequency is 22.1184 MHz * 9/4 = 49.7664 MHz
//
void SYSCLK_Init (void)
{
   int i;                           // delay counter

   char old_SFRPAGE = SFRPAGE;      // Store current SFRPAGE

   SFRPAGE = CONFIG_PAGE;           // set SFR page

   OSCXCN = 0x67;                   // start external oscillator with
                                    // 22.1184MHz crystal

   for (i=0; i < 256; i++) ;        // Wait for osc. to start up

   while (!(OSCXCN & 0x80)) ;       // Wait for crystal osc. to settle

   CLKSEL = 0x01;                   // Select the external osc. as
                                    // the SYSCLK source

   OSCICN = 0x00;                   // Disable the internal osc.

   //Turn on the PLL and increase the system clock by a factor of M/N = 9/4
   SFRPAGE = PLL0_PAGE;

   PLL0CN  = 0x04;                  // Set PLL source as external osc.
   SFRPAGE = LEGACY_PAGE;
   FLSCL   = 0x10;                  // Set FLASH read time for 50MHz clk
                                    // or less
   SFRPAGE = PLL0_PAGE;
   PLL0CN |= 0x01;                  // Enable Power to PLL
   PLL0DIV = 0x04;                  // Set Pre-divide value to N (N = 4)
   PLL0FLT = 0x01;                  // Set the PLL filter register for
                                    // a reference clock from 19 - 30 MHz
                                    // and an output clock from 45 - 80 MHz
   PLL0MUL = 0x09;                  // Multiply SYSCLK by M (M = 9)

   for (i=0; i < 256; i++) ;        // Wait at least 5us
   PLL0CN  |= 0x02;                 // Enable the PLL
   while(!(PLL0CN & 0x10));         // Wait until PLL frequency is locked
   CLKSEL  = 0x02;                  // Select PLL as SYSCLK source

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
}

//-------------------------------------------------------------------------
// ADC0_Init
//-------------------------------------------------------------------------
//
// Configure ADC0 to use Timer3 overflows as conversion source, to
// generate an interrupt on conversion complete, and to use left-justified
// output mode.  Enables ADC0 end of conversion interrupt. Enables ADC0.
//
void ADC0_Init (void)
{
   char old_SFRPAGE = SFRPAGE;      // Store current SFRPAGE

   SFRPAGE = ADC0_PAGE;             // Switch to ADC0 Setup Page
```

```
   ADC0CN = 0x05;                   // ADC disabled; normal tracking
                                    // mode; ADC conversions are initiated
                                    // on overflow of Timer3, left-justify

   REF0CN = 0x03;                   // enable on-chip VREF and output buffer

   AMX0CF = 0x00;                   // Single-ended AIN0.0 input
   AMX0SL = 0x00;

   ADC0CF = (SYSCLK/(2*2500000)) << 3;  // ADC conversion clock <= 2.5MHz
   ADC0CF |= 0x00;                  // PGA gain = 1

   AD0EN = 1;                       // enable ADC0

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
}

//-----------------------------------------------------------------------------
// TIMER3_Init
//-----------------------------------------------------------------------------
//
// Configure Timer3 to auto-reload at interval specified by <counts> (no
// interrupt generated) using SYSCLK as its time base.
//
void TIMER3_Init (int counts)
{
   char old_SFRPAGE = SFRPAGE;      // Save Current SFR page

   SFRPAGE = TMR3_PAGE;             // Switch to Timer3 Setup Page

   TMR3CN = 0x00;                   // Stop Timer3; Clear TF3
   TMR3CF = 0x08;                   // use SYSCLK as timebase

   RCAP3  = -counts;                // Init reload values
   TMR3    = 0xffff;                // set to reload immediately
   EIE2   &= ~0x01;                 // disable Timer3 interrupts
   TR3 = 0x01;                      // start Timer3

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
}


//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------------
//
// ADC end-of-conversion ISR
// The ADC sample is stored in memory, and an index variable is incremented.
// If enough samples have been taken to process the FFT, then a flag is set,
// and ADC interrupts are disabled until the next set is requested.
//
void ADC0_ISR (void) interrupt 15 using 3
{
```

```
   AD0INT = 0;                     // clear ADC conversion complete
                                   // flag

   Real[ADC_Index] = ADC0;         // store ADC value

   ADC_Index++;                    // Increment the index into memory

   if (ADC_Index >= NUM_FFT)       // If enough samples have been collected
   {
      Conversion_Set_Complete = 1; // Tell the Main Routine and...
      EIE2 &= ~0x02;               // disable ADC interrupts
   }

}
```

IntFFT_BROUT.c

```
//-----------------------------------------------------------------------------
// IntFFT_BROUT.c
//-----------------------------------------------------------------------------
// Copyright 2003 Cygnal Integrated Products, Inc.
//
// AUTH: BD
// DATE: 30 JAN 03
//
// This program collects data using ADC0 at <SAMPLE_RATE> Hz, and performs
// an FFT on the data.  The Real and Imaginary parts of the results are then
// sent to the UART peripheral at <BAUDRATE> bps, where they can be displayed
// or captured using a terminal program.
//
// Note that the FFT performed in this software is optimized for storage space
// (RAM).  The resulting Frequency-domain data is not suitable for analyzing
// Signal-to-noise or distortion performance.
//
// This program uses a 22.1184 Mhz crystal oscillator multiplied by (9/4)
// for an effective SYSCLK of 49.7664 Mhz. This program also initializes and
// uses UART0 at <BAUDRATE> bits per second.
//
// Target: C8051F12x
// Tool chain: KEIL C51 6.03
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f120.h>                 // SFR declarations
#include <stdio.h>

#include “FFT_Code_Tables.h”           // Code Tables for FFT routines

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for ‘F12x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                 // data pointer
sfr16 ADC0     = 0xbe;                 // ADC0 data
sfr16 ADC0GT   = 0xc4;                 // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                 // ADC0 less than window
sfr16 RCAP2    = 0xca;                 // Timer2 capture/reload
sfr16 RCAP3    = 0xca;                 // Timer3 capture/reload
sfr16 RCAP4    = 0xca;                 // Timer4 capture/reload
sfr16 TMR2     = 0xcc;                 // Timer2
sfr16 TMR3     = 0xcc;                 // Timer3
sfr16 TMR4     = 0xcc;                 // Timer4
sfr16 DAC0     = 0xd2;                 // DAC0 data
sfr16 DAC1     = 0xd2;                 // DAC1 data
sfr16 PCA0CP5  = 0xe1;                 // PCA0 Module 5 capture
sfr16 PCA0CP2  = 0xe9;                 // PCA0 Module 2 capture
sfr16 PCA0CP3  = 0xeb;                 // PCA0 Module 3 capture
sfr16 PCA0CP4  = 0xed;                 // PCA0 Module 4 capture
sfr16 PCA0     = 0xf9;                 // PCA0 counter
sfr16 PCA0CP0  = 0xfb;                 // PCA0 Module 0 capture
sfr16 PCA0CP1  = 0xfd;                 // PCA0 Module 1 capture
```

```
//-----------------------------------------------------------------------------
// Global CONSTANTS and Variable Type Definitions
//-----------------------------------------------------------------------------
#define NUM_BITS 16                     // Number of Bits in Data

#define DATA_BEGIN 0x0000               // Beginning of XRAM Data

#define EXTCLK       22118400           // External oscillator frequency in Hz
#define SYSCLK       49760000           // Output of PLL derived from
                                        // (EXTCLK*9/4)
#define BAUDRATE     115200             // Baud Rate for UART0

#define SAMPLE_RATE  10000              // Sample frequency in Hz

#define RUN_ONCE     1                  // Setting to a non-zero value will
                                        // cause the program to stop after one
                                        // data set.

typedef union IBALONG {                 // Integer or Byte-addressable LONG
   long l;                              // long: Var.l
   unsigned int i[2];                   // u int: Var.i[0]:Var.i[1]
   unsigned char b[4];                  // u char: Var.b[0]:Var.b[1]:
                                        //          Var.b[2]:Var.b[3]
} IBALONG;

typedef union BAINT {                   // Byte-addressable INT
   int i;                               // int: Var.i
   unsigned char b[2];                  // u char: Var.b[0]:Var.b[1]
} BAINT;

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void WindowCalc(int Win_Array[], unsigned char SE_data);
void Int_FFT(int ReArray[], int ImArray[]);
void Bit_Reverse(int BR_Array[], int BR_Array2[]);

void SYSCLK_Init (void);
void PORT_Init (void);
void UART0_Init (void);
void ADC0_Init (void);
void TIMER3_Init (int counts);
void ADC0_ISR (void);


//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------

// XRAM storage of FFT: requires NUM_FFT*4 Bytes after DATA_BEGIN address
int xdata Real[NUM_FFT] _at_ DATA_BEGIN;
int xdata Imag[NUM_FFT] _at_ (DATA_BEGIN + (NUM_FFT * 2));

// NUM_FFT is defined in the “FFT_Code_Tables.h” header file
#if (NUM_FFT >= 256)
unsigned int index, ADC_Index;
#endif
```

```
#if (NUM_FFT < 256)
unsigned char index, ADC_Index;
#endif

unsigned int BinNum;

bit Conversion_Set_Complete;           // This indicates when the data has been
                                       // stored, and is ready to be processed
                                       // using the FFT routines

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main()
{

   // disable watchdog timer
   WDTCN = 0xde;
   WDTCN = 0xad;


   SYSCLK_Init();                      // initialize external clock and PLL
   PORT_Init ();                       // set up Port I/O
   UART0_Init ();                      // initialize UART0
   TIMER3_Init (SYSCLK/SAMPLE_RATE);   // initialize Timer3 to overflow at
                                       // <SAMPLE_RATE>
   ADC0_Init ();                       // init ADC0

   EA = 1;                             // globally enable interrupts

   while (1)
   {
      ADC_Index = 0;
      Conversion_Set_Complete = 0;

      EIE2 |= 0x02;                    // enable ADC interrupts

      SFRPAGE = LEGACY_PAGE;

      while(!Conversion_Set_Complete);


      SFRPAGE = UART0_PAGE;
      printf("\nCollected Data\nSample\tValue\n");
      for (BinNum = 0; BinNum < NUM_FFT; BinNum++)
      {
         // Print Data in the format: Sample <tab> Value <tab>
         printf("%d\t%u\n", BinNum, Real[BinNum]);
      }

      WindowCalc(Real, 1);             // Window Real Data and convert to
                                       // differential if it is single-ended

      Int_FFT(Real, Imag);             // Perform FFT on data

      Bit_Reverse(Real, Imag);         // Sort Output in bit-reverse order

      SFRPAGE = UART0_PAGE;
```

```
      printf("\nBin\tReal\tImag\n");

      // Output the FFT data to the UART
      for (BinNum = 0; BinNum < NUM_FFT; BinNum++)
      {
         // Print Data in the format: Bin <tab> Real <tab> Imaginary
         printf("%d\t%d\t%d\n", BinNum, Real[BinNum], Imag[BinNum]);
      }

      if (RUN_ONCE)
         while(1);

   }

} // END MAIN


//-------------------------------------------------------------------------
// WindowCalc
//-------------------------------------------------------------------------
//
// Uses the values in WindowFunc[] to window the stored data.
//
// The WindowFunc[] Array contains window coefficients between samples
// 0 and (NUM_FFT/2)-1, and samples from NUM_FFT/2 to NUM_FFT-1 are the mirror
// image of the other side.
// Window values are interpreted as a fraction of 1 (WindowFunc[x]/65536).
// The value at NUM_FFT/2 is assumed to be 1.0 (65536).
//
// If SE_data = 1, the input data is assumed to be single-ended, and is
// converted to a 2's complement, differential representation, to cancel the DC
// offset.
//
void WindowCalc(int Win_Array[], unsigned char SE_data)
{

#if (WINDOW_TYPE != 0)  // Use this section if a window has been specified
IBALONG NewVal;

   if (SE_data)                              // If data is single-ended,
      Win_Array[0] ^= 0x8000;                // convert it to differential
   NewVal.l = (long)Win_Array[0] * WindowFunc[0];
   if ((NewVal.l < 0)&&(NewVal.i[1]))
     Win_Array[0] = NewVal.i[0] + 1;
   else Win_Array[0] = NewVal.i[0];

   if (SE_data)                              // If data is single-ended,
      Win_Array[NUM_FFT/2] ^= 0x8000;        // convert it to differential

  for (index = 1; index < NUM_FFT/2; index++)
  {
      // Array positions 1 to (NUM_FFT/2 - 1)
      if (SE_data)                           // If data is single-ended,
         Win_Array[index] ^= 0x8000;         // convert it to differential
      NewVal.l = (long)Win_Array[index] * WindowFunc[index];
      if ((NewVal.l < 0)&&(NewVal.i[1]))
        Win_Array[index] = NewVal.i[0] + 1;
      else Win_Array[index] = NewVal.i[0];
```

```
         // Array positions (NUM_FFT/2 + 1) to (NUM_FFT - 1)
      if (SE_data)                          // If data is single-ended,
         Win_Array[NUM_FFT-index] ^= 0x8000; // convert it to differential
      NewVal.l = (long)Win_Array[NUM_FFT-index] * WindowFunc[index];
      if ((NewVal.l < 0)&&(NewVal.i[1]))
         Win_Array[NUM_FFT-index] = NewVal.i[0] + 1;
      else Win_Array[NUM_FFT-index] = NewVal.i[0];

   }

#endif

#if (WINDOW_TYPE == 0)  // Compile this if no window has been specified

   if (SE_data)                             // If data is single-ended,
   {                                        // convert it to differential

      for (index = 0; index < NUM_FFT; index++)
      {
         Win_Array[index] ^= 0x8000;        // XOR MSB with '1' to invert
      }
   }

#endif

}  // END WindowCalc


//-----------------------------------------------------------------------------
// Bit_Reverse
//-----------------------------------------------------------------------------
//
// Sorts two arrays in Bit Reversed Address order
//
// The BRTable[] array is used to find which values must be swapped.  Only
// half of this array is stored, to save code space.  The second half is
// assumed to be a mirror image of the first half.
//
void Bit_Reverse(int BR_Array[], int BR_Array2[])
{

#if (NUM_FFT >= 512)
unsigned int swapA, swapB, sw_cnt;          // Swap Indices
#endif

#if (NUM_FFT <= 256)
unsigned char swapA, swapB, sw_cnt;         // Swap Indices
#endif

int TempStore;

   // Loop through locations to swap
   for (sw_cnt = 1; sw_cnt < NUM_FFT/2; sw_cnt++)
   {
      swapA = sw_cnt;                       // Store current location
      swapB = BRTable[sw_cnt] * 2;          // Retrieve bit-reversed index
      if (swapB > swapA)                    // If the bit-reversed index is
      {                                     // larger than the current index,
         TempStore = BR_Array[swapA];       // the two data locations are
```

```
         BR_Array[swapA] = BR_Array[swapB];  // swapped. Using this comparison
         BR_Array[swapB] = TempStore;        // ensures that locations are only
                                             // swapped once, and never with
         TempStore = BR_Array2[swapA];       // themselves
         BR_Array2[swapA] = BR_Array2[swapB];
         BR_Array2[swapB] = TempStore;
      }


      swapA += NUM_FFT/2;                    // Now perform the same operations
      swapB++;                               // on the second half of the data
      if (swapB > swapA)
      {
         TempStore = BR_Array[swapA];
         BR_Array[swapA] = BR_Array[swapB];
         BR_Array[swapB] = TempStore;

         TempStore = BR_Array2[swapA];
         BR_Array2[swapA] = BR_Array2[swapB];
         BR_Array2[swapB] = TempStore;
      }
   }

}  // END Bit Reverse Order Sort


//---------------------------------------------------------------------------
// Int_FFT
//---------------------------------------------------------------------------
//
// Performs a Radix-2 Decimation-In-Time FFT on the input array ReArray[]
//
// During each stage of the FFT, the values are calculated using a set of
// “Butterfly” equations, as listed below:
//
// Re1 = Re1 + (Cos(x)*Re2 + Sin(x)*Im2)
// Re2 = Re1 - (Cos(x)*Re2 + Sin(x)*Im2)
// Im1 = Im1 + (Cos(x)*Im2 - Sin(x)*Re2)
// Im2 = Im1 - (Cos(x)*Im2 - Sin(x)*Re2)
//
// The routine implements this calculation using the following values:
//
// Re1 = ReArray[indexA], Re2 = ReArray[indexB]
// Im1 = ImArray[indexA], Im2 = ImArray[indexB]
// x = the angle: 2*pi*(sin_index/NUM_FFT), in radians.  The necessary values
//    are stored in code space in the SinTable[] array.
//
//
// Key Points for using this FFT routine:
//
// 1) It expects REAL data (in ReArray[]), in 2’s complement, 16-bit binary
//    format and assumes a value of 0 for all imaginary locations
//    (in ImArray[]).
//
// 2) It expects the REAL input data to be sorted in normal order, and the
//    output data produced is in bit-reversed index order.
//
// 3) SIN and COS values are retrieved and calculated from a table consisting
//    of 1/4 of a period of a SIN function.
```

```
//
// 4) It is optimized to use integer math only (no floating-point operations),
//    and for storage space.  The input, all intermediate stages, and the
//    output of the FFT are stored as 16-bit INTEGER values. This limits the
//    precision of the routine.  When using input data of less than 16-bits,
//    the best results are produced by left-justifying the data prior to
//    windowing and performing the FFT.
//
// 5) The algorithm is a Radix-2 type, meaning that the number of samples must
//    be 2^N, where N is an integer.  The minimum number of samples to process
//    is 4.  The constant NUM_FFT contains the number of samples to process.
//
//
void Int_FFT(int ReArray[], int ImArray[])
{

#if (NUM_FFT >= 512)
unsigned int sin_index, sinB_index, g_cnt, s_cnt;  // Keeps track of the proper
unsigned int indexA, indexB;                       // index locations for each
#endif                                             // calculation

#if (NUM_FFT <= 256)
unsigned char sin_index, sinB_index, g_cnt, s_cnt; // Keeps track of the proper
unsigned char indexA, indexB;                      // index locations for each
#endif                                             // calculation

unsigned int group = 2, stage = NUM_FFT/4;
long CosVal, SinVal;
long TempImA, TempImB, TempReA, TempReB, TempReA2, TempReB2;
IBALONG ReTwid, ImTwid, TempL;

// FIRST STAGE - optimized for REAL input data only.  This will set all
// Imaginary locations to zero.
//
// Shortcuts have been taken to remove unnecessary multiplications during this
// stage. The angle “x” is 0 radians for all calculations at this point, so
// the SIN value is equal to 0.0 and the COS value is equal to 1.0.
// Additionally, all Imaginary locations are assumed to be ‘0’ in this stage of
// the algorithm, and are set to ‘0’.

   indexA = 0;
   for (s_cnt = 0; s_cnt < NUM_FFT/2; s_cnt++)
   {
      indexB = indexA + NUM_FFT/2;

      TempReA = ReArray[indexA];
      TempReB = ReArray[indexB];

      // Calculate new value for ReArray[indexA]
      TempL.l = TempReA + TempReB;
      if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
         TempReA2 = (TempL.l >> 1) + 1;
      else TempReA2 = TempL.l >> 1;

      // Calculate new value for ReArray[indexB]
      TempL.l = TempReA - TempReB;
      if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
         ReArray[indexB] = (TempL.l >> 1) + 1;
      else ReArray[indexB] = TempL.l >> 1;
```

```
      ReArray[indexA] = TempReA2;

      ImArray[indexA] = 0;                  // set Imaginary locations to '0'
      ImArray[indexB] = 0;

      indexA++;
   }

// END OF FIRST STAGE


while (group <= NUM_FFT/2)
{
   sinB_index = 0;
   sin_index = 0;
   indexA = 0;

   for (g_cnt = 0; g_cnt < group; g_cnt++)
   {
      for (s_cnt = 0; s_cnt < stage; s_cnt++)
      {
         indexB = indexA + stage;

         TempReA = ReArray[indexA];
         TempReB = ReArray[indexB];
         TempImA = ImArray[indexA];
         TempImB = ImArray[indexB];

// The following first checks for the special cases when the angle "x" is
// equal to either 0 or pi/2 radians.  In these cases, unnecessary
// multiplications have been removed to improve the processing speed.

         if (sin_index == 0)  // corresponds to "x" = 0 radians
         {

            // Calculate new value for ReArray[indexA]
            TempL.l = TempReA + TempReB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
               TempReA2 = (TempL.l >> 1) + 1;
            else TempReA2 = TempL.l >> 1;

            // Calculate new value for ReArray[indexB]
            TempL.l = TempReA - TempReB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
               TempReB2 = (TempL.l >> 1) + 1;
            else TempReB2 = TempL.l >> 1;

            // Calculate new value for ImArray[indexB]
           TempL.l = TempImA - TempImB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
               TempImB = (TempL.l >> 1) + 1;
            else TempImB = TempL.l >> 1;

            // Calculate new value for ImArray[indexA]
            TempL.l = TempImA + TempImB;
            if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
               TempImA = (TempL.l >> 1) + 1;
            else TempImA = TempL.l >> 1;
```

```
}
else if (sin_index == NUM_FFT/4) // corresponds to “x” = pi/2 radians
{

   // Calculate new value for ReArray[indexB]
   TempL.l = TempReA - TempImB;
   if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
      TempReB2 = (TempL.l >> 1) + 1;
   else TempReB2 = TempL.l >> 1;

   // Calculate new value for ReArray[indexA]
   TempL.l = TempReA + TempImB;
   if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
      TempReA2 = (TempL.l >> 1) + 1;
   else TempReA2 = TempL.l >> 1;

   // Calculate new value for ImArray[indexB]
   TempL.l = TempImA + TempReB;
   if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
      TempImB = (TempL.l >> 1) + 1;
   else TempImB = TempL.l >> 1;

   // Calculate new value for ImArray[indexA]
   TempL.l = TempImA - TempReB;
   if ((TempL.l < 0)&&(0x01 & TempL.b[3]))
      TempImA = (TempL.l >> 1) + 1;
   else TempImA = TempL.l >> 1;

}

else
{
   // If no multiplication shortcuts can be taken, the SIN and COS
   // values for the Butterfly calculation are fetched from the
   // SinTable[] array.

   if (sin_index > NUM_FFT/4)
   {
      SinVal = SinTable[(NUM_FFT/2) - sin_index];
      CosVal = -SinTable[sin_index - (NUM_FFT/4)];
   }
   else
   {
      SinVal = SinTable[sin_index];
      CosVal = SinTable[(NUM_FFT/4) - sin_index];
   }

   // The SIN and COS values are used here to calculate part of the
   // Butterfly equation
   ReTwid.l = (TempReB * CosVal) +
         (TempImB * SinVal);

   ImTwid.l = (TempImB * CosVal) -
         (TempReB * SinVal);

   // Using the values calculated above, the new variables
   // are computed
```

```
            // Calculate new value for ReArray[indexA]
            TempL.i[1] = 0;
            TempL.i[0] = TempReA;
            TempL.l = TempL.l >> 1;
            ReTwid.l += TempL.l;
            if ((ReTwid.l < 0)&&(ReTwid.i[1]))
               TempReA2 = ReTwid.i[0] + 1;
            else TempReA2 = ReTwid.i[0];

            // Calculate new value for ReArray[indexB]
            TempL.l = TempL.l << 1;
            TempL.l -= ReTwid.l;
            if ((TempL.l < 0)&&(TempL.i[1]))
               TempReB2 = TempL.i[0] + 1;
            else TempReB2 = TempL.i[0];

            // Calculate new value for ImArray[indexA]
            TempL.i[1] = 0;
            TempL.i[0] = TempImA;
            TempL.l = TempL.l >> 1;
            ImTwid.l += TempL.l;
            if ((ImTwid.l < 0)&&(ImTwid.i[1]))
               TempImA = ImTwid.i[0] + 1;
            else TempImA = ImTwid.i[0];

            // Calculate new value for ImArray[indexB]
            TempL.l = TempL.l << 1;
            TempL.l -= ImTwid.l;
            if ((TempL.l < 0)&&(TempL.i[1]))
               TempImB = TempL.i[0] + 1;
            else TempImB = TempL.i[0];

         }

         ReArray[indexA] = TempReA2;
         ReArray[indexB] = TempReB2;
         ImArray[indexA] = TempImA;
         ImArray[indexB] = TempImB;


         indexA++;
      }                                       // END of stage FOR loop (s_cnt)
      indexA = indexB + 1;
      sin_index = BRTable[++sinB_index];
   }                                          // END of group FOR loop (g_cnt)

   group *= 2;
   stage /= 2;
}                                             // END of While loop

}  // END Int_FFT


//---------------------------------------------------------------------------
// Initialization Routines
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// PORT_Init
```

```
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports
//
void PORT_Init (void)
{
char old_SFRPAGE = SFRPAGE;         // Store current SFRPAGE

   SFRPAGE = CONFIG_PAGE;           // Switch to configuration page


   XBR0    = 0x04;                  // Enable UART0 on crossbar
   XBR1    = 0x00;
   XBR2    = 0x40;                  // Enable crossbar and weak pull-ups
   P0MDOUT |= 0x01;                 // Set TX0 pin to push-pull

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
}

//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <baudrate> and 8-N-1. In order to
// increase the clocking flexibility of Timer0, Timer1 is configured to count
// SYSCLKs.
//
// To use this routine SYSCLK/BAUDRATE/16 must be less than 256. For example,
// if SYSCLK = 50 MHz, the lowest standard baud rate supported by this
// routine is 19,200 bps.
void UART0_Init (void)
{
char old_SFRPAGE = SFRPAGE;         // Store current SFRPAGE

   SFRPAGE = UART0_PAGE;            // Switch to UART0 Page

   SCON0  = 0x50;                   // SCON0: mode 0, 8-bit UART, enable RX
   SSTA0  = 0x10;                   // Timer 1 generates UART0 baud rate and
                                    // UART0 baud rate divide by two disabled
   SFRPAGE = TIMER01_PAGE;
   TMOD   &= ~0xF0;
   TMOD   |=  0x20;                 // TMOD: timer 1, mode 2, 8-bit reload

   TH1 = -(SYSCLK/BAUDRATE/16);     // Set the Timer1 reload value
                                    // When using a low baud rate, this
                                    // equation should be checked to ensure
                                    // that the reload value will fit in
                                    // 8-bits.

   CKCON |= 0x10;                   // T1M = 1; SCA1:0 = xx


   TL1 = TH1;                       // initialize Timer1
   TR1 = 1;                         // start Timer1

   SFRPAGE = UART0_PAGE;
   TI0 = 1;                         // Indicate TX0 ready

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
```

```
}

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use an external 22.1184 MHz
// crystal oscillator multiplied by a factor of 9/4 using the PLL as its
// clock source. The resulting frequency is 22.1184 MHz * 9/4 = 49.7664 MHz
//
void SYSCLK_Init (void)
{
   int i;                           // delay counter

   char old_SFRPAGE = SFRPAGE;      // Store current SFRPAGE

   SFRPAGE = CONFIG_PAGE;           // set SFR page

   OSCXCN = 0x67;                   // start external oscillator with
                                    // 22.1184MHz crystal

   for (i=0; i < 256; i++) ;        // Wait for osc. to start up

   while (!(OSCXCN & 0x80)) ;       // Wait for crystal osc. to settle

   CLKSEL = 0x01;                   // Select the external osc. as
                                    // the SYSCLK source

   OSCICN = 0x00;                   // Disable the internal osc.

   //Turn on the PLL and increase the system clock by a factor of M/N = 9/4
   SFRPAGE = PLL0_PAGE;

   PLL0CN  = 0x04;                  // Set PLL source as external osc.
   SFRPAGE = LEGACY_PAGE;
   FLSCL   = 0x10;                  // Set FLASH read time for 50MHz clk
                                    // or less
   SFRPAGE = PLL0_PAGE;
   PLL0CN |= 0x01;                  // Enable Power to PLL
   PLL0DIV = 0x04;                  // Set Pre-divide value to N (N = 4)
   PLL0FLT = 0x01;                  // Set the PLL filter register for
                                    // a reference clock from 19 - 30 MHz
                                    // and an output clock from 45 - 80 MHz
   PLL0MUL = 0x09;                  // Multiply SYSCLK by M (M = 9)

   for (i=0; i < 256; i++) ;        // Wait at least 5us
   PLL0CN  |= 0x02;                 // Enable the PLL
   while(!(PLL0CN & 0x10));         // Wait until PLL frequency is locked
   CLKSEL  = 0x02;                  // Select PLL as SYSCLK source

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
}

//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
//
// Configure ADC0 to use Timer3 overflows as conversion source, to
```

```
// generate an interrupt on conversion complete, and to use left-justified
// output mode.  Enables ADC0 end of conversion interrupt. Enables ADC0.
//
void ADC0_Init (void)
{
   char old_SFRPAGE = SFRPAGE;      // Store current SFRPAGE

   SFRPAGE = ADC0_PAGE;             // Switch to ADC0 Setup Page

   ADC0CN = 0x05;                   // ADC disabled; normal tracking
                                    // mode; ADC conversions are initiated
                                    // on overflow of Timer3, left-justify

   REF0CN = 0x03;                   // enable on-chip VREF and output buffer

   AMX0CF = 0x00;                   // Single-ended AIN0.0 input
   AMX0SL = 0x00;

   ADC0CF = (SYSCLK/(2*2500000)) << 3;  // ADC conversion clock <= 2.5MHz
   ADC0CF |= 0x00;                  // PGA gain = 1

   AD0EN = 1;                       // enable ADC0

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
}

//-----------------------------------------------------------------------------
// TIMER3_Init
//-----------------------------------------------------------------------------
//
// Configure Timer3 to auto-reload at interval specified by <counts> (no
// interrupt generated) using SYSCLK as its time base.
//
void TIMER3_Init (int counts)
{
   char old_SFRPAGE = SFRPAGE;      // Save Current SFR page

   SFRPAGE = TMR3_PAGE;             // Switch to Timer3 Setup Page

   TMR3CN = 0x00;                   // Stop Timer3; Clear TF3
   TMR3CF = 0x08;                   // use SYSCLK as timebase

   RCAP3  = -counts;                // Init reload values
   TMR3    = 0xffff;                // set to reload immediately
   EIE2   &= ~0x01;                 // disable Timer3 interrupts
   TR3 = 0x01;                      // start Timer3

   SFRPAGE = old_SFRPAGE;           // restore SFRPAGE
}


//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------------
//
```

```
// ADC end-of-conversion ISR
// The ADC sample is stored in memory, and an index variable is incremented
// If enough samples have been taken to process the FFT, then a flag is set.
// and ADC interrupts are disabled until the next set is requested.
//
void ADC0_ISR (void) interrupt 15 using 3
{

   AD0INT = 0;                     // clear ADC conversion complete
                                   // flag

   Real[ADC_Index] = ADC0;         // store ADC value

   ADC_Index++;                    // Increment the index into memory

   if (ADC_Index >= NUM_FFT)       // If enough samples have been collected
   {
      Conversion_Set_Complete = 1; // Tell the Main Routine and...
      EIE2 &= ~0x02;               // disable ADC interrupts
   }

}
```

FFT_Code_Tables.h

```
//-----------------------------------------------------------------------------
// FFT_Code_Tables.h
//-----------------------------------------------------------------------------
// Copyright 2003 Cygnal Integrated Products, Inc.
//
// AUTH: BD
// DATE: 30 JAN 03
//
// This header file is used to provide Sine, Bit Reversal, and Window tables
// for calculating an FFT.  The tables are stored in FLASH memory (code space).
// All of the tables are conditionally used at compile time, so only one table
// of each type is used in the software.
//
// Target: C8051F12x
// Tool chain: KEIL C51 6.03
//

#define NUM_FFT 256     // Length of FFT to process
                        // Must be 2^N, where N is an integer >= 2

#define WINDOW_TYPE 4   // WINDOW_TYPE specifies the window to use on the data
                        // The available window functions are:
                        // 0 = No Window
                        // 1 = Triangle Window
                        // 2 = Hanning Window
                        // 3 = Hamming Window
                        // 4 = Blackman Window

// SinTable[] - SIN Tables are first 1/4 of a SIN wave - used to perform
// complex math functions.  These are encoded such that a value of 1.0
// corresponds to 32768, and a value of -1.0 corresponds to -32768.

// BRTable[] - Bit Reversal tables are used to bit-reverse sort the data and
// perform other indexing functions.  The Bit Reversal tables are stored
// as 1/2 of their actual value, and the real value is computed at
// runtime.

// WindowFunc[] - Tables used to window data.  These are encoded such that
// 1.0 corresponds to 65536, and 0.0 corresponds to 0.


//-----------------------------------------------------------------------------
// SIN and BR Tables for NUM_FFT = 1024
//-----------------------------------------------------------------------------
#if (NUM_FFT == 1024)
int code SinTable[256] =
{
0x0000, 0x00C9, 0x0192, 0x025B, 0x0324, 0x03ED, 0x04B6, 0x057F,
0x0647, 0x0710, 0x07D9, 0x08A2, 0x096A, 0x0A33, 0x0AFB, 0x0BC3,
0x0C8B, 0x0D53, 0x0E1B, 0x0EE3, 0x0FAB, 0x1072, 0x1139, 0x1201,
0x12C8, 0x138E, 0x1455, 0x151B, 0x15E2, 0x16A8, 0x176D, 0x1833,
0x18F8, 0x19BD, 0x1A82, 0x1B47, 0x1C0B, 0x1CCF, 0x1D93, 0x1E56,
0x1F19, 0x1FDC, 0x209F, 0x2161, 0x2223, 0x22E5, 0x23A6, 0x2467,
0x2528, 0x25E8, 0x26A8, 0x2767, 0x2826, 0x28E5, 0x29A3, 0x2A61,
0x2B1F, 0x2BDC, 0x2C98, 0x2D55, 0x2E11, 0x2ECC, 0x2F87, 0x3041,
0x30FB, 0x31B5, 0x326E, 0x3326, 0x33DE, 0x3496, 0x354D, 0x3604,
```

```
0x36BA, 0x376F, 0x3824, 0x38D8, 0x398C, 0x3A40, 0x3AF2, 0x3BA5,
0x3C56, 0x3D07, 0x3DB8, 0x3E68, 0x3F17, 0x3FC5, 0x4073, 0x4121,
0x41CE, 0x427A, 0x4325, 0x43D0, 0x447A, 0x4524, 0x45CD, 0x4675,
0x471C, 0x47C3, 0x4869, 0x490F, 0x49B4, 0x4A58, 0x4AFB, 0x4B9E,
0x4C3F, 0x4CE1, 0x4D81, 0x4E21, 0x4EBF, 0x4F5E, 0x4FFB, 0x5097,
0x5133, 0x51CE, 0x5269, 0x5302, 0x539B, 0x5433, 0x54CA, 0x5560,
0x55F5, 0x568A, 0x571D, 0x57B0, 0x5842, 0x58D4, 0x5964, 0x59F3,
0x5A82, 0x5B10, 0x5B9D, 0x5C29, 0x5CB4, 0x5D3E, 0x5DC7, 0x5E50,
0x5ED7, 0x5F5E, 0x5FE3, 0x6068, 0x60EC, 0x616F, 0x61F1, 0x6271,
0x62F2, 0x6371, 0x63EF, 0x646C, 0x64E8, 0x6563, 0x65DD, 0x6657,
0x66CF, 0x6746, 0x67BD, 0x6832, 0x68A6, 0x6919, 0x698C, 0x69FD,
0x6A6D, 0x6ADC, 0x6B4A, 0x6BB8, 0x6C24, 0x6C8F, 0x6CF9, 0x6D62,
0x6DCA, 0x6E30, 0x6E96, 0x6EFB, 0x6F5F, 0x6FC1, 0x7023, 0x7083,
0x70E2, 0x7141, 0x719E, 0x71FA, 0x7255, 0x72AF, 0x7307, 0x735F,
0x73B5, 0x740B, 0x745F, 0x74B2, 0x7504, 0x7555, 0x75A5, 0x75F4,
0x7641, 0x768E, 0x76D9, 0x7723, 0x776C, 0x77B4, 0x77FA, 0x7840,
0x7884, 0x78C7, 0x7909, 0x794A, 0x798A, 0x79C8, 0x7A05, 0x7A42,
0x7A7D, 0x7AB6, 0x7AEF, 0x7B26, 0x7B5D, 0x7B92, 0x7BC5, 0x7BF8,
0x7C29, 0x7C5A, 0x7C89, 0x7CB7, 0x7CE3, 0x7D0F, 0x7D39, 0x7D62,
0x7D8A, 0x7DB0, 0x7DD6, 0x7DFA, 0x7E1D, 0x7E3F, 0x7E5F, 0x7E7F,
0x7E9D, 0x7EBA, 0x7ED5, 0x7EF0, 0x7F09, 0x7F21, 0x7F38, 0x7F4D,
0x7F62, 0x7F75, 0x7F87, 0x7F97, 0x7FA7, 0x7FB5, 0x7FC2, 0x7FCE,
0x7FD8, 0x7FE1, 0x7FE9, 0x7FF0, 0x7FF6, 0x7FFA, 0x7FFD, 0x7FFF
};

unsigned int code BRTable[512] =
{
0, 256, 128, 384, 64, 320, 192, 448,
32, 288, 160, 416, 96, 352, 224, 480,
16, 272, 144, 400, 80, 336, 208, 464,
48, 304, 176, 432, 112, 368, 240, 496,
8, 264, 136, 392, 72, 328, 200, 456,
40, 296, 168, 424, 104, 360, 232, 488,
24, 280, 152, 408, 88, 344, 216, 472,
56, 312, 184, 440, 120, 376, 248, 504,
4, 260, 132, 388, 68, 324, 196, 452,
36, 292, 164, 420, 100, 356, 228, 484,
20, 276, 148, 404, 84, 340, 212, 468,
52, 308, 180, 436, 116, 372, 244, 500,
12, 268, 140, 396, 76, 332, 204, 460,
44, 300, 172, 428, 108, 364, 236, 492,
28, 284, 156, 412, 92, 348, 220, 476,
60, 316, 188, 444, 124, 380, 252, 508,
2, 258, 130, 386, 66, 322, 194, 450,
34, 290, 162, 418, 98, 354, 226, 482,
18, 274, 146, 402, 82, 338, 210, 466,
50, 306, 178, 434, 114, 370, 242, 498,
10, 266, 138, 394, 74, 330, 202, 458,
42, 298, 170, 426, 106, 362, 234, 490,
26, 282, 154, 410, 90, 346, 218, 474,
58, 314, 186, 442, 122, 378, 250, 506,
6, 262, 134, 390, 70, 326, 198, 454,
38, 294, 166, 422, 102, 358, 230, 486,
22, 278, 150, 406, 86, 342, 214, 470,
54, 310, 182, 438, 118, 374, 246, 502,
14, 270, 142, 398, 78, 334, 206, 462,
46, 302, 174, 430, 110, 366, 238, 494,
30, 286, 158, 414, 94, 350, 222, 478,
62, 318, 190, 446, 126, 382, 254, 510,
```

```
1, 257, 129, 385, 65, 321, 193, 449,
33, 289, 161, 417, 97, 353, 225, 481,
17, 273, 145, 401, 81, 337, 209, 465,
49, 305, 177, 433, 113, 369, 241, 497,
9, 265, 137, 393, 73, 329, 201, 457,
41, 297, 169, 425, 105, 361, 233, 489,
25, 281, 153, 409, 89, 345, 217, 473,
57, 313, 185, 441, 121, 377, 249, 505,
5, 261, 133, 389, 69, 325, 197, 453,
37, 293, 165, 421, 101, 357, 229, 485,
21, 277, 149, 405, 85, 341, 213, 469,
53, 309, 181, 437, 117, 373, 245, 501,
13, 269, 141, 397, 77, 333, 205, 461,
45, 301, 173, 429, 109, 365, 237, 493,
29, 285, 157, 413, 93, 349, 221, 477,
61, 317, 189, 445, 125, 381, 253, 509,
3, 259, 131, 387, 67, 323, 195, 451,
35, 291, 163, 419, 99, 355, 227, 483,
19, 275, 147, 403, 83, 339, 211, 467,
51, 307, 179, 435, 115, 371, 243, 499,
11, 267, 139, 395, 75, 331, 203, 459,
43, 299, 171, 427, 107, 363, 235, 491,
27, 283, 155, 411, 91, 347, 219, 475,
59, 315, 187, 443, 123, 379, 251, 507,
7, 263, 135, 391, 71, 327, 199, 455,
39, 295, 167, 423, 103, 359, 231, 487,
23, 279, 151, 407, 87, 343, 215, 471,
55, 311, 183, 439, 119, 375, 247, 503,
15, 271, 143, 399, 79, 335, 207, 463,
47, 303, 175, 431, 111, 367, 239, 495,
31, 287, 159, 415, 95, 351, 223, 479,
63, 319, 191, 447, 127, 383, 255, 511
};
#endif

//-----------------------------------------------------------------------------
// SIN and BR Tables for NUM_FFT = 512
//-----------------------------------------------------------------------------
#if (NUM_FFT == 512)
int code SinTable[128] =
{
0x0000, 0x0192, 0x0324, 0x04B6, 0x0647, 0x07D9, 0x096A, 0x0AFB,
0x0C8B, 0x0E1B, 0x0FAB, 0x1139, 0x12C8, 0x1455, 0x15E2, 0x176D,
0x18F8, 0x1A82, 0x1C0B, 0x1D93, 0x1F19, 0x209F, 0x2223, 0x23A6,
0x2528, 0x26A8, 0x2826, 0x29A3, 0x2B1F, 0x2C98, 0x2E11, 0x2F87,
0x30FB, 0x326E, 0x33DE, 0x354D, 0x36BA, 0x3824, 0x398C, 0x3AF2,
0x3C56, 0x3DB8, 0x3F17, 0x4073, 0x41CE, 0x4325, 0x447A, 0x45CD,
0x471C, 0x4869, 0x49B4, 0x4AFB, 0x4C3F, 0x4D81, 0x4EBF, 0x4FFB,
0x5133, 0x5269, 0x539B, 0x54CA, 0x55F5, 0x571D, 0x5842, 0x5964,
0x5A82, 0x5B9D, 0x5CB4, 0x5DC7, 0x5ED7, 0x5FE3, 0x60EC, 0x61F1,
0x62F2, 0x63EF, 0x64E8, 0x65DD, 0x66CF, 0x67BD, 0x68A6, 0x698C,
0x6A6D, 0x6B4A, 0x6C24, 0x6CF9, 0x6DCA, 0x6E96, 0x6F5F, 0x7023,
0x70E2, 0x719E, 0x7255, 0x7307, 0x73B5, 0x745F, 0x7504, 0x75A5,
0x7641, 0x76D9, 0x776C, 0x77FA, 0x7884, 0x7909, 0x798A, 0x7A05,
0x7A7D, 0x7AEF, 0x7B5D, 0x7BC5, 0x7C29, 0x7C89, 0x7CE3, 0x7D39,
0x7D8A, 0x7DD6, 0x7E1D, 0x7E5F, 0x7E9D, 0x7ED5, 0x7F09, 0x7F38,
0x7F62, 0x7F87, 0x7FA7, 0x7FC2, 0x7FD8, 0x7FE9, 0x7FF6, 0x7FFD
};
```

```
unsigned char code BRTable[256] =
{
0, 128, 64, 192, 32, 160, 96, 224,
16, 144, 80, 208, 48, 176, 112, 240,
8, 136, 72, 200, 40, 168, 104, 232,
24, 152, 88, 216, 56, 184, 120, 248,
4, 132, 68, 196, 36, 164, 100, 228,
20, 148, 84, 212, 52, 180, 116, 244,
12, 140, 76, 204, 44, 172, 108, 236,
28, 156, 92, 220, 60, 188, 124, 252,
2, 130, 66, 194, 34, 162, 98, 226,
18, 146, 82, 210, 50, 178, 114, 242,
10, 138, 74, 202, 42, 170, 106, 234,
26, 154, 90, 218, 58, 186, 122, 250,
6, 134, 70, 198, 38, 166, 102, 230,
22, 150, 86, 214, 54, 182, 118, 246,
14, 142, 78, 206, 46, 174, 110, 238,
30, 158, 94, 222, 62, 190, 126, 254,
1, 129, 65, 193, 33, 161, 97, 225,
17, 145, 81, 209, 49, 177, 113, 241,
9, 137, 73, 201, 41, 169, 105, 233,
25, 153, 89, 217, 57, 185, 121, 249,
5, 133, 69, 197, 37, 165, 101, 229,
21, 149, 85, 213, 53, 181, 117, 245,
13, 141, 77, 205, 45, 173, 109, 237,
29, 157, 93, 221, 61, 189, 125, 253,
3, 131, 67, 195, 35, 163, 99, 227,
19, 147, 83, 211, 51, 179, 115, 243,
11, 139, 75, 203, 43, 171, 107, 235,
27, 155, 91, 219, 59, 187, 123, 251,
7, 135, 71, 199, 39, 167, 103, 231,
23, 151, 87, 215, 55, 183, 119, 247,
15, 143, 79, 207, 47, 175, 111, 239,
31, 159, 95, 223, 63, 191, 127, 255
};
#endif

//-----------------------------------------------------------------------------
// SIN and BR Tables for NUM_FFT = 256
//-----------------------------------------------------------------------------
#if (NUM_FFT == 256)
int code SinTable[64] =
{
0x0000, 0x0324, 0x0647, 0x096A, 0x0C8B, 0x0FAB, 0x12C8, 0x15E2,
0x18F8, 0x1C0B, 0x1F19, 0x2223, 0x2528, 0x2826, 0x2B1F, 0x2E11,
0x30FB, 0x33DE, 0x36BA, 0x398C, 0x3C56, 0x3F17, 0x41CE, 0x447A,
0x471C, 0x49B4, 0x4C3F, 0x4EBF, 0x5133, 0x539B, 0x55F5, 0x5842,
0x5A82, 0x5CB4, 0x5ED7, 0x60EC, 0x62F2, 0x64E8, 0x66CF, 0x68A6,
0x6A6D, 0x6C24, 0x6DCA, 0x6F5F, 0x70E2, 0x7255, 0x73B5, 0x7504,
0x7641, 0x776C, 0x7884, 0x798A, 0x7A7D, 0x7B5D, 0x7C29, 0x7CE3,
0x7D8A, 0x7E1D, 0x7E9D, 0x7F09, 0x7F62, 0x7FA7, 0x7FD8, 0x7FF6
};

unsigned char code BRTable[128] =
{
0, 64, 32, 96, 16, 80, 48, 112,
8, 72, 40, 104, 24, 88, 56, 120,
4, 68, 36, 100, 20, 84, 52, 116,
12, 76, 44, 108, 28, 92, 60, 124,
```

```
2, 66, 34, 98, 18, 82, 50, 114,
10, 74, 42, 106, 26, 90, 58, 122,
6, 70, 38, 102, 22, 86, 54, 118,
14, 78, 46, 110, 30, 94, 62, 126,
1, 65, 33, 97, 17, 81, 49, 113,
9, 73, 41, 105, 25, 89, 57, 121,
5, 69, 37, 101, 21, 85, 53, 117,
13, 77, 45, 109, 29, 93, 61, 125,
3, 67, 35, 99, 19, 83, 51, 115,
11, 75, 43, 107, 27, 91, 59, 123,
7, 71, 39, 103, 23, 87, 55, 119,
15, 79, 47, 111, 31, 95, 63, 127
};
#endif

//-----------------------------------------------------------------------------
// SIN and BR Tables for NUM_FFT = 128
//-----------------------------------------------------------------------------
#if (NUM_FFT == 128)
int code SinTable[32] =
{
0x0000, 0x0647, 0x0C8B, 0x12C8, 0x18F8, 0x1F19, 0x2528, 0x2B1F,
0x30FB, 0x36BA, 0x3C56, 0x41CE, 0x471C, 0x4C3F, 0x5133, 0x55F5,
0x5A82, 0x5ED7, 0x62F2, 0x66CF, 0x6A6D, 0x6DCA, 0x70E2, 0x73B5,
0x7641, 0x7884, 0x7A7D, 0x7C29, 0x7D8A, 0x7E9D, 0x7F62, 0x7FD8
};

unsigned char code BRTable[64] =
{
0, 32, 16, 48, 8, 40, 24, 56,
4, 36, 20, 52, 12, 44, 28, 60,
2, 34, 18, 50, 10, 42, 26, 58,
6, 38, 22, 54, 14, 46, 30, 62,
1, 33, 17, 49, 9, 41, 25, 57,
5, 37, 21, 53, 13, 45, 29, 61,
3, 35, 19, 51, 11, 43, 27, 59,
7, 39, 23, 55, 15, 47, 31, 63
};
#endif

//-----------------------------------------------------------------------------
// SIN and BR Tables for NUM_FFT = 64
//-----------------------------------------------------------------------------
#if (NUM_FFT == 64)
int code SinTable[16] =
{
0x0000, 0x0C8B, 0x18F8, 0x2528, 0x30FB, 0x3C56, 0x471C, 0x5133,
0x5A82, 0x62F2, 0x6A6D, 0x70E2, 0x7641, 0x7A7D, 0x7D8A, 0x7F62
};

unsigned char code BRTable[32] =
{
0, 16, 8, 24, 4, 20, 12, 28,
2, 18, 10, 26, 6, 22, 14, 30,
1, 17, 9, 25, 5, 21, 13, 29,
3, 19, 11, 27, 7, 23, 15, 31
};
#endif
```

```
//-----------------------------------------------------------------------------
// SIN and BR Tables for NUM_FFT = 32
//-----------------------------------------------------------------------------
#if (NUM_FFT == 32)
int code SinTable[8] =
{
0x0000, 0x18F8, 0x30FB, 0x471C, 0x5A82, 0x6A6D, 0x7641, 0x7D8A
};

unsigned char code BRTable[16] =
{
0, 8, 4, 12, 2, 10, 6, 14,
1, 9, 5, 13, 3, 11, 7, 15
};
#endif

//-----------------------------------------------------------------------------
// SIN and BR Tables for NUM_FFT = 16
//-----------------------------------------------------------------------------
#if (NUM_FFT == 16)
int code SinTable[4] =
{
0x0000, 0x30FB, 0x5A82, 0x7641
};

unsigned char code BRTable[8] =
{
0, 4, 2, 6, 1, 5, 3, 7
};
#endif

//-----------------------------------------------------------------------------
// SIN and BR Tables for NUM_FFT = 8
//-----------------------------------------------------------------------------
#if (NUM_FFT == 8)
int code SinTable[2] =
{
0x0000, 0x5A82
};

unsigned char code BRTable[4] =
{
0, 2, 1, 3
};
#endif

//-----------------------------------------------------------------------------
// SIN and BR Table for NUM_FFT = 4
//-----------------------------------------------------------------------------
#if (NUM_FFT == 4)

int code SinTable[1] =
{
0x0000
};

unsigned char code BRTable[2] =
{
0, 1
```

```
};
#endif


//-----------------------------------------------------------------------------
// Window Functions for NUM_FFT = 1024
//-----------------------------------------------------------------------------
#if (NUM_FFT == 1024)

#if (WINDOW_TYPE == 1)
// Triangle Window
unsigned int code WindowFunc[512] =
{
0x0000, 0x0080, 0x0100, 0x0180, 0x0200, 0x0280, 0x0300, 0x0380,
0x0400, 0x0480, 0x0500, 0x0580, 0x0600, 0x0680, 0x0700, 0x0780,
0x0800, 0x0880, 0x0900, 0x0980, 0x0A00, 0x0A80, 0x0B00, 0x0B80,
0x0C00, 0x0C80, 0x0D00, 0x0D80, 0x0E00, 0x0E80, 0x0F00, 0x0F80,
0x1000, 0x1080, 0x1100, 0x1180, 0x1200, 0x1280, 0x1300, 0x1380,
0x1400, 0x1480, 0x1500, 0x1580, 0x1600, 0x1680, 0x1700, 0x1780,
0x1800, 0x1880, 0x1900, 0x1980, 0x1A00, 0x1A80, 0x1B00, 0x1B80,
0x1C00, 0x1C80, 0x1D00, 0x1D80, 0x1E00, 0x1E80, 0x1F00, 0x1F80,
0x2000, 0x2080, 0x2100, 0x2180, 0x2200, 0x2280, 0x2300, 0x2380,
0x2400, 0x2480, 0x2500, 0x2580, 0x2600, 0x2680, 0x2700, 0x2780,
0x2800, 0x2880, 0x2900, 0x2980, 0x2A00, 0x2A80, 0x2B00, 0x2B80,
0x2C00, 0x2C80, 0x2D00, 0x2D80, 0x2E00, 0x2E80, 0x2F00, 0x2F80,
0x3000, 0x3080, 0x3100, 0x3180, 0x3200, 0x3280, 0x3300, 0x3380,
0x3400, 0x3480, 0x3500, 0x3580, 0x3600, 0x3680, 0x3700, 0x3780,
0x3800, 0x3880, 0x3900, 0x3980, 0x3A00, 0x3A80, 0x3B00, 0x3B80,
0x3C00, 0x3C80, 0x3D00, 0x3D80, 0x3E00, 0x3E80, 0x3F00, 0x3F80,
0x4000, 0x4080, 0x4100, 0x4180, 0x4200, 0x4280, 0x4300, 0x4380,
0x4400, 0x4480, 0x4500, 0x4580, 0x4600, 0x4680, 0x4700, 0x4780,
0x4800, 0x4880, 0x4900, 0x4980, 0x4A00, 0x4A80, 0x4B00, 0x4B80,
0x4C00, 0x4C80, 0x4D00, 0x4D80, 0x4E00, 0x4E80, 0x4F00, 0x4F80,
0x5000, 0x5080, 0x5100, 0x5180, 0x5200, 0x5280, 0x5300, 0x5380,
0x5400, 0x5480, 0x5500, 0x5580, 0x5600, 0x5680, 0x5700, 0x5780,
0x5800, 0x5880, 0x5900, 0x5980, 0x5A00, 0x5A80, 0x5B00, 0x5B80,
0x5C00, 0x5C80, 0x5D00, 0x5D80, 0x5E00, 0x5E80, 0x5F00, 0x5F80,
0x6000, 0x6080, 0x6100, 0x6180, 0x6200, 0x6280, 0x6300, 0x6380,
0x6400, 0x6480, 0x6500, 0x6580, 0x6600, 0x6680, 0x6700, 0x6780,
0x6800, 0x6880, 0x6900, 0x6980, 0x6A00, 0x6A80, 0x6B00, 0x6B80,
0x6C00, 0x6C80, 0x6D00, 0x6D80, 0x6E00, 0x6E80, 0x6F00, 0x6F80,
0x7000, 0x7080, 0x7100, 0x7180, 0x7200, 0x7280, 0x7300, 0x7380,
0x7400, 0x7480, 0x7500, 0x7580, 0x7600, 0x7680, 0x7700, 0x7780,
0x7800, 0x7880, 0x7900, 0x7980, 0x7A00, 0x7A80, 0x7B00, 0x7B80,
0x7C00, 0x7C80, 0x7D00, 0x7D80, 0x7E00, 0x7E80, 0x7F00, 0x7F80,
0x8000, 0x8080, 0x8100, 0x8180, 0x8200, 0x8280, 0x8300, 0x8380,
0x8400, 0x8480, 0x8500, 0x8580, 0x8600, 0x8680, 0x8700, 0x8780,
0x8800, 0x8880, 0x8900, 0x8980, 0x8A00, 0x8A80, 0x8B00, 0x8B80,
0x8C00, 0x8C80, 0x8D00, 0x8D80, 0x8E00, 0x8E80, 0x8F00, 0x8F80,
0x9000, 0x9080, 0x9100, 0x9180, 0x9200, 0x9280, 0x9300, 0x9380,
0x9400, 0x9480, 0x9500, 0x9580, 0x9600, 0x9680, 0x9700, 0x9780,
0x9800, 0x9880, 0x9900, 0x9980, 0x9A00, 0x9A80, 0x9B00, 0x9B80,
0x9C00, 0x9C80, 0x9D00, 0x9D80, 0x9E00, 0x9E80, 0x9F00, 0x9F80,
0xA000, 0xA080, 0xA100, 0xA180, 0xA200, 0xA280, 0xA300, 0xA380,
0xA400, 0xA480, 0xA500, 0xA580, 0xA600, 0xA680, 0xA700, 0xA780,
0xA800, 0xA880, 0xA900, 0xA980, 0xAA00, 0xAA80, 0xAB00, 0xAB80,
0xAC00, 0xAC80, 0xAD00, 0xAD80, 0xAE00, 0xAE80, 0xAF00, 0xAF80,
0xB000, 0xB080, 0xB100, 0xB180, 0xB200, 0xB280, 0xB300, 0xB380,
0xB400, 0xB480, 0xB500, 0xB580, 0xB600, 0xB680, 0xB700, 0xB780,
```

```
0xB800, 0xB880, 0xB900, 0xB980, 0xBA00, 0xBA80, 0xBB00, 0xBB80,
0xBC00, 0xBC80, 0xBD00, 0xBD80, 0xBE00, 0xBE80, 0xBF00, 0xBF80,
0xC000, 0xC080, 0xC100, 0xC180, 0xC200, 0xC280, 0xC300, 0xC380,
0xC400, 0xC480, 0xC500, 0xC580, 0xC600, 0xC680, 0xC700, 0xC780,
0xC800, 0xC880, 0xC900, 0xC980, 0xCA00, 0xCA80, 0xCB00, 0xCB80,
0xCC00, 0xCC80, 0xCD00, 0xCD80, 0xCE00, 0xCE80, 0xCF00, 0xCF80,
0xD000, 0xD080, 0xD100, 0xD180, 0xD200, 0xD280, 0xD300, 0xD380,
0xD400, 0xD480, 0xD500, 0xD580, 0xD600, 0xD680, 0xD700, 0xD780,
0xD800, 0xD880, 0xD900, 0xD980, 0xDA00, 0xDA80, 0xDB00, 0xDB80,
0xDC00, 0xDC80, 0xDD00, 0xDD80, 0xDE00, 0xDE80, 0xDF00, 0xDF80,
0xE000, 0xE080, 0xE100, 0xE180, 0xE200, 0xE280, 0xE300, 0xE380,
0xE400, 0xE480, 0xE500, 0xE580, 0xE600, 0xE680, 0xE700, 0xE780,
0xE800, 0xE880, 0xE900, 0xE980, 0xEA00, 0xEA80, 0xEB00, 0xEB80,
0xEC00, 0xEC80, 0xED00, 0xED80, 0xEE00, 0xEE80, 0xEF00, 0xEF80,
0xF000, 0xF080, 0xF100, 0xF180, 0xF200, 0xF280, 0xF300, 0xF380,
0xF400, 0xF480, 0xF500, 0xF580, 0xF600, 0xF680, 0xF700, 0xF780,
0xF800, 0xF880, 0xF900, 0xF980, 0xFA00, 0xFA80, 0xFB00, 0xFB80,
0xFC00, 0xFC80, 0xFD00, 0xFD80, 0xFE00, 0xFE80, 0xFF00, 0xFF80
};
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[512] =
{
0x0000, 0x0000, 0x0002, 0x0005, 0x0009, 0x000F, 0x0016, 0x001E,
0x0027, 0x0031, 0x003D, 0x004A, 0x0058, 0x0068, 0x0078, 0x008A,
0x009D, 0x00B2, 0x00C7, 0x00DE, 0x00F6, 0x010F, 0x012A, 0x0145,
0x0162, 0x0180, 0x01A0, 0x01C0, 0x01E2, 0x0205, 0x0229, 0x024F,
0x0275, 0x029D, 0x02C6, 0x02F0, 0x031C, 0x0348, 0x0376, 0x03A5,
0x03D6, 0x0407, 0x043A, 0x046D, 0x04A2, 0x04D9, 0x0510, 0x0549,
0x0582, 0x05BD, 0x05FA, 0x0637, 0x0675, 0x06B5, 0x06F6, 0x0738,
0x077B, 0x07BF, 0x0805, 0x084B, 0x0893, 0x08DC, 0x0926, 0x0971,
0x09BE, 0x0A0B, 0x0A5A, 0x0AAA, 0x0AFB, 0x0B4D, 0x0BA0, 0x0BF4,
0x0C4A, 0x0CA0, 0x0CF8, 0x0D50, 0x0DAA, 0x0E05, 0x0E61, 0x0EBE,
0x0F1D, 0x0F7C, 0x0FDC, 0x103E, 0x10A0, 0x1104, 0x1169, 0x11CF,
0x1235, 0x129D, 0x1306, 0x1370, 0x13DB, 0x1447, 0x14B5, 0x1523,
0x1592, 0x1602, 0x1673, 0x16E6, 0x1759, 0x17CD, 0x1842, 0x18B9,
0x1930, 0x19A8, 0x1A22, 0x1A9C, 0x1B17, 0x1B93, 0x1C10, 0x1C8E,
0x1D0D, 0x1D8E, 0x1E0E, 0x1E90, 0x1F13, 0x1F97, 0x201C, 0x20A1,
0x2128, 0x21AF, 0x2238, 0x22C1, 0x234B, 0x23D6, 0x2462, 0x24EF,
0x257D, 0x260C, 0x269B, 0x272B, 0x27BD, 0x284F, 0x28E2, 0x2975,
0x2A0A, 0x2A9F, 0x2B35, 0x2BCC, 0x2C64, 0x2CFD, 0x2D96, 0x2E31,
0x2ECC, 0x2F68, 0x3004, 0x30A1, 0x3140, 0x31DE, 0x327E, 0x331E,
0x33C0, 0x3461, 0x3504, 0x35A7, 0x364B, 0x36F0, 0x3796, 0x383C,
0x38E3, 0x398A, 0x3A32, 0x3ADB, 0x3B85, 0x3C2F, 0x3CDA, 0x3D85,
0x3E31, 0x3EDE, 0x3F8C, 0x403A, 0x40E8, 0x4197, 0x4247, 0x42F8,
0x43A9, 0x445A, 0x450D, 0x45BF, 0x4673, 0x4727, 0x47DB, 0x4890,
0x4945, 0x49FB, 0x4AB2, 0x4B69, 0x4C21, 0x4CD9, 0x4D91, 0x4E4A,
0x4F04, 0x4FBE, 0x5078, 0x5133, 0x51EE, 0x52AA, 0x5367, 0x5423,
0x54E0, 0x559E, 0x565C, 0x571A, 0x57D9, 0x5898, 0x5957, 0x5A17,
0x5AD7, 0x5B98, 0x5C59, 0x5D1A, 0x5DDC, 0x5E9E, 0x5F60, 0x6023,
0x60E6, 0x61A9, 0x626C, 0x6330, 0x63F4, 0x64B8, 0x657D, 0x6642,
0x6707, 0x67CC, 0x6892, 0x6957, 0x6A1D, 0x6AE4, 0x6BAA, 0x6C71,
0x6D37, 0x6DFE, 0x6EC6, 0x6F8D, 0x7054, 0x711C, 0x71E4, 0x72AC,
0x7374, 0x743C, 0x7504, 0x75CC, 0x7695, 0x775D, 0x7826, 0x78EF,
0x79B8, 0x7A80, 0x7B49, 0x7C12, 0x7CDB, 0x7DA4, 0x7E6D, 0x7F36,
0x7FFF, 0x80C9, 0x8192, 0x825B, 0x8324, 0x83ED, 0x84B6, 0x857F,
0x8647, 0x8710, 0x87D9, 0x88A2, 0x896A, 0x8A33, 0x8AFB, 0x8BC3,
```

```
0x8C8B, 0x8D53, 0x8E1B, 0x8EE3, 0x8FAB, 0x9072, 0x9139, 0x9201,
0x92C8, 0x938E, 0x9455, 0x951B, 0x95E2, 0x96A8, 0x976D, 0x9833,
0x98F8, 0x99BD, 0x9A82, 0x9B47, 0x9C0B, 0x9CCF, 0x9D93, 0x9E56,
0x9F19, 0x9FDC, 0xA09F, 0xA161, 0xA223, 0xA2E5, 0xA3A6, 0xA467,
0xA528, 0xA5E8, 0xA6A8, 0xA767, 0xA826, 0xA8E5, 0xA9A3, 0xAA61,
0xAB1F, 0xABDC, 0xAC98, 0xAD55, 0xAE11, 0xAECC, 0xAF87, 0xB041,
0xB0FB, 0xB1B5, 0xB26E, 0xB326, 0xB3DE, 0xB496, 0xB54D, 0xB604,
0xB6BA, 0xB76F, 0xB824, 0xB8D8, 0xB98C, 0xBA40, 0xBAF2, 0xBBA5,
0xBC56, 0xBD07, 0xBDB8, 0xBE68, 0xBF17, 0xBFC5, 0xC073, 0xC121,
0xC1CE, 0xC27A, 0xC325, 0xC3D0, 0xC47A, 0xC524, 0xC5CD, 0xC675,
0xC71C, 0xC7C3, 0xC869, 0xC90F, 0xC9B4, 0xCA58, 0xCAFB, 0xCB9E,
0xCC3F, 0xCCE1, 0xCD81, 0xCE21, 0xCEBF, 0xCF5E, 0xCFFB, 0xD097,
0xD133, 0xD1CE, 0xD269, 0xD302, 0xD39B, 0xD433, 0xD4CA, 0xD560,
0xD5F5, 0xD68A, 0xD71D, 0xD7B0, 0xD842, 0xD8D4, 0xD964, 0xD9F3,
0xDA82, 0xDB10, 0xDB9D, 0xDC29, 0xDCB4, 0xDD3E, 0xDDC7, 0xDE50,
0xDED7, 0xDF5E, 0xDFE3, 0xE068, 0xE0EC, 0xE16F, 0xE1F1, 0xE271,
0xE2F2, 0xE371, 0xE3EF, 0xE46C, 0xE4E8, 0xE563, 0xE5DD, 0xE657,
0xE6CF, 0xE746, 0xE7BD, 0xE832, 0xE8A6, 0xE919, 0xE98C, 0xE9FD,
0xEA6D, 0xEADC, 0xEB4A, 0xEBB8, 0xEC24, 0xEC8F, 0xECF9, 0xED62,
0xEDCA, 0xEE30, 0xEE96, 0xEEFB, 0xEF5F, 0xEFC1, 0xF023, 0xF083,
0xF0E2, 0xF141, 0xF19E, 0xF1FA, 0xF255, 0xF2AF, 0xF307, 0xF35F,
0xF3B5, 0xF40B, 0xF45F, 0xF4B2, 0xF504, 0xF555, 0xF5A5, 0xF5F4,
0xF641, 0xF68E, 0xF6D9, 0xF723, 0xF76C, 0xF7B4, 0xF7FA, 0xF840,
0xF884, 0xF8C7, 0xF909, 0xF94A, 0xF98A, 0xF9C8, 0xFA05, 0xFA42,
0xFA7D, 0xFAB6, 0xFAEF, 0xFB26, 0xFB5D, 0xFB92, 0xFBC5, 0xFBF8,
0xFC29, 0xFC5A, 0xFC89, 0xFCB7, 0xFCE3, 0xFD0F, 0xFD39, 0xFD62,
0xFD8A, 0xFDB0, 0xFDD6, 0xFDFA, 0xFE1D, 0xFE3F, 0xFE5F, 0xFE7F,
0xFE9D, 0xFEBA, 0xFED5, 0xFEF0, 0xFF09, 0xFF21, 0xFF38, 0xFF4D,
0xFF62, 0xFF75, 0xFF87, 0xFF97, 0xFFA7, 0xFFB5, 0xFFC2, 0xFFCE,
0xFFD8, 0xFFE1, 0xFFE9, 0xFFF0, 0xFFF6, 0xFFFA, 0xFFFD, 0xFFFF
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[512] =
{
0x147A, 0x147B, 0x147D, 0x147F, 0x1483, 0x1489, 0x148F, 0x1496,
0x149F, 0x14A8, 0x14B3, 0x14BF, 0x14CC, 0x14DA, 0x14EA, 0x14FA,
0x150C, 0x151E, 0x1532, 0x1547, 0x155D, 0x1574, 0x158D, 0x15A6,
0x15C1, 0x15DC, 0x15F9, 0x1617, 0x1636, 0x1656, 0x1678, 0x169A,
0x16BE, 0x16E2, 0x1708, 0x172F, 0x1757, 0x1780, 0x17AA, 0x17D5,
0x1802, 0x182F, 0x185E, 0x188E, 0x18BE, 0x18F0, 0x1923, 0x1957,
0x198C, 0x19C3, 0x19FA, 0x1A32, 0x1A6C, 0x1AA6, 0x1AE2, 0x1B1F,
0x1B5D, 0x1B9B, 0x1BDB, 0x1C1C, 0x1C5E, 0x1CA2, 0x1CE6, 0x1D2B,
0x1D71, 0x1DB8, 0x1E01, 0x1E4A, 0x1E95, 0x1EE0, 0x1F2D, 0x1F7A,
0x1FC9, 0x2018, 0x2069, 0x20BB, 0x210D, 0x2161, 0x21B6, 0x220B,
0x2262, 0x22BA, 0x2312, 0x236C, 0x23C7, 0x2422, 0x247F, 0x24DD,
0x253B, 0x259B, 0x25FC, 0x265D, 0x26C0, 0x2723, 0x2787, 0x27ED,
0x2853, 0x28BA, 0x2922, 0x298C, 0x29F6, 0x2A61, 0x2ACC, 0x2B39,
0x2BA7, 0x2C16, 0x2C85, 0x2CF6, 0x2D67, 0x2DD9, 0x2E4C, 0x2EC0,
0x2F35, 0x2FAB, 0x3022, 0x3099, 0x3112, 0x318B, 0x3205, 0x3280,
0x32FC, 0x3378, 0x33F6, 0x3474, 0x34F3, 0x3573, 0x35F4, 0x3676,
0x36F8, 0x377B, 0x37FF, 0x3884, 0x390A, 0x3990, 0x3A17, 0x3A9F,
0x3B28, 0x3BB1, 0x3C3B, 0x3CC6, 0x3D52, 0x3DDE, 0x3E6C, 0x3EFA,
0x3F88, 0x4018, 0x40A8, 0x4138, 0x41CA, 0x425C, 0x42EF, 0x4382,
0x4417, 0x44AC, 0x4541, 0x45D7, 0x466E, 0x4706, 0x479E, 0x4837,
0x48D0, 0x496B, 0x4A05, 0x4AA1, 0x4B3D, 0x4BD9, 0x4C76, 0x4D14,
```

```
0x4DB3, 0x4E51, 0x4EF1, 0x4F91, 0x5032, 0x50D3, 0x5175, 0x5217,
0x52BA, 0x535D, 0x5401, 0x54A6, 0x554B, 0x55F0, 0x5696, 0x573D,
0x57E4, 0x588B, 0x5933, 0x59DB, 0x5A84, 0x5B2E, 0x5BD7, 0x5C82,
0x5D2C, 0x5DD7, 0x5E83, 0x5F2F, 0x5FDB, 0x6088, 0x6135, 0x61E3,
0x6291, 0x633F, 0x63EE, 0x649D, 0x654D, 0x65FC, 0x66AC, 0x675D,
0x680E, 0x68BF, 0x6971, 0x6A22, 0x6AD4, 0x6B87, 0x6C3A, 0x6CED,
0x6DA0, 0x6E53, 0x6F07, 0x6FBB, 0x7070, 0x7124, 0x71D9, 0x728E,
0x7344, 0x73F9, 0x74AF, 0x7565, 0x761B, 0x76D1, 0x7788, 0x783F,
0x78F6, 0x79AD, 0x7A64, 0x7B1B, 0x7BD3, 0x7C8A, 0x7D42, 0x7DFA,
0x7EB2, 0x7F6A, 0x8022, 0x80DB, 0x8193, 0x824C, 0x8304, 0x83BD,
0x8476, 0x852E, 0x85E7, 0x86A0, 0x8759, 0x8812, 0x88CB, 0x8984,
0x8A3D, 0x8AF6, 0x8BAF, 0x8C68, 0x8D21, 0x8DDA, 0x8E93, 0x8F4B,
0x9004, 0x90BD, 0x9176, 0x922E, 0x92E7, 0x939F, 0x9457, 0x9510,
0x95C8, 0x9680, 0x9738, 0x97F0, 0x98A7, 0x995F, 0x9A16, 0x9ACD,
0x9B84, 0x9C3B, 0x9CF2, 0x9DA9, 0x9E5F, 0x9F15, 0x9FCB, 0xA081,
0xA136, 0xA1EC, 0xA2A1, 0xA355, 0xA40A, 0xA4BE, 0xA573, 0xA626,
0xA6DA, 0xA78D, 0xA840, 0xA8F3, 0xA9A5, 0xAA58, 0xAB09, 0xABBB,
0xAC6C, 0xAD1D, 0xADCD, 0xAE7E, 0xAF2D, 0xAFDD, 0xB08C, 0xB13B,
0xB1E9, 0xB297, 0xB345, 0xB3F2, 0xB49F, 0xB54B, 0xB5F7, 0xB6A2,
0xB74E, 0xB7F8, 0xB8A2, 0xB94C, 0xB9F6, 0xBA9E, 0xBB47, 0xBBEF,
0xBC96, 0xBD3D, 0xBDE4, 0xBE8A, 0xBF2F, 0xBFD4, 0xC079, 0xC11D,
0xC1C0, 0xC263, 0xC305, 0xC3A7, 0xC448, 0xC4E9, 0xC589, 0xC628,
0xC6C7, 0xC766, 0xC804, 0xC8A1, 0xC93D, 0xC9D9, 0xCA75, 0xCB0F,
0xCBA9, 0xCC43, 0xCCDC, 0xCD74, 0xCE0C, 0xCEA2, 0xCF39, 0xCFCE,
0xD063, 0xD0F7, 0xD18B, 0xD21E, 0xD2B0, 0xD342, 0xD3D2, 0xD462,
0xD4F2, 0xD580, 0xD60E, 0xD69B, 0xD728, 0xD7B4, 0xD83E, 0xD8C9,
0xD952, 0xD9DB, 0xDA63, 0xDAEA, 0xDB70, 0xDBF6, 0xDC7B, 0xDCFF,
0xDD82, 0xDE04, 0xDE86, 0xDF07, 0xDF86, 0xE006, 0xE084, 0xE101,
0xE17E, 0xE1FA, 0xE275, 0xE2EF, 0xE368, 0xE3E1, 0xE458, 0xE4CF,
0xE545, 0xE5B9, 0xE62D, 0xE6A1, 0xE713, 0xE784, 0xE7F5, 0xE864,
0xE8D3, 0xE941, 0xE9AD, 0xEA19, 0xEA84, 0xEAEE, 0xEB57, 0xEBC0,
0xEC27, 0xEC8D, 0xECF3, 0xED57, 0xEDBA, 0xEE1D, 0xEE7E, 0xEEDF,
0xEF3F, 0xEF9D, 0xEFFB, 0xF057, 0xF0B3, 0xF10E, 0xF167, 0xF1C0,
0xF218, 0xF26F, 0xF2C4, 0xF319, 0xF36D, 0xF3BF, 0xF411, 0xF462,
0xF4B1, 0xF500, 0xF54D, 0xF59A, 0xF5E5, 0xF630, 0xF679, 0xF6C1,
0xF709, 0xF74F, 0xF794, 0xF7D8, 0xF81B, 0xF85E, 0xF89E, 0xF8DE,
0xF91D, 0xF95B, 0xF998, 0xF9D3, 0xFA0E, 0xFA47, 0xFA80, 0xFAB7,
0xFAED, 0xFB23, 0xFB57, 0xFB8A, 0xFBBB, 0xFBEC, 0xFC1C, 0xFC4B,
0xFC78, 0xFCA4, 0xFCD0, 0xFCFA, 0xFD23, 0xFD4B, 0xFD72, 0xFD98,
0xFDBC, 0xFDE0, 0xFE02, 0xFE23, 0xFE44, 0xFE63, 0xFE81, 0xFE9E,
0xFEB9, 0xFED4, 0xFEED, 0xFF06, 0xFF1D, 0xFF33, 0xFF48, 0xFF5C,
0xFF6E, 0xFF80, 0xFF90, 0xFFA0, 0xFFAE, 0xFFBB, 0xFFC7, 0xFFD2,
0xFFDB, 0xFFE4, 0xFFEB, 0xFFF1, 0xFFF6, 0xFFFA, 0xFFFD, 0xFFFF
};
#endif


#if (WINDOW_TYPE == 4)
// Blackman Window
unsigned int code WindowFunc[512] =
{
0x0000, 0x0000, 0x0000, 0x0001, 0x0003, 0x0005, 0x0007, 0x000A,
0x000E, 0x0012, 0x0016, 0x001A, 0x0020, 0x0025, 0x002B, 0x0032,
0x0039, 0x0040, 0x0048, 0x0050, 0x0059, 0x0062, 0x006C, 0x0076,
0x0080, 0x008B, 0x0097, 0x00A3, 0x00AF, 0x00BC, 0x00CA, 0x00D8,
0x00E6, 0x00F5, 0x0104, 0x0114, 0x0124, 0x0135, 0x0146, 0x0158,
0x016A, 0x017D, 0x0190, 0x01A4, 0x01B9, 0x01CD, 0x01E3, 0x01F9,
0x020F, 0x0226, 0x023D, 0x0255, 0x026E, 0x0287, 0x02A0, 0x02BA,
0x02D5, 0x02F0, 0x030C, 0x0328, 0x0345, 0x0362, 0x0380, 0x039F,
```

```
0x03BE, 0x03DE, 0x03FE, 0x041F, 0x0441, 0x0463, 0x0486, 0x04A9,
0x04CD, 0x04F1, 0x0516, 0x053C, 0x0563, 0x058A, 0x05B1, 0x05DA,
0x0603, 0x062C, 0x0656, 0x0681, 0x06AD, 0x06D9, 0x0706, 0x0734,
0x0762, 0x0791, 0x07C1, 0x07F1, 0x0822, 0x0854, 0x0886, 0x08B9,
0x08ED, 0x0922, 0x0957, 0x098D, 0x09C4, 0x09FC, 0x0A34, 0x0A6D,
0x0AA7, 0x0AE2, 0x0B1D, 0x0B59, 0x0B96, 0x0BD4, 0x0C12, 0x0C51,
0x0C91, 0x0CD2, 0x0D14, 0x0D56, 0x0D9A, 0x0DDE, 0x0E23, 0x0E68,
0x0EAF, 0x0EF6, 0x0F3F, 0x0F88, 0x0FD2, 0x101D, 0x1068, 0x10B5,
0x1102, 0x1150, 0x11A0, 0x11F0, 0x1240, 0x1292, 0x12E5, 0x1339,
0x138D, 0x13E2, 0x1439, 0x1490, 0x14E8, 0x1541, 0x159B, 0x15F6,
0x1652, 0x16AF, 0x170D, 0x176B, 0x17CB, 0x182B, 0x188D, 0x18EF,
0x1953, 0x19B7, 0x1A1D, 0x1A83, 0x1AEA, 0x1B53, 0x1BBC, 0x1C26,
0x1C91, 0x1CFE, 0x1D6B, 0x1DD9, 0x1E48, 0x1EB8, 0x1F2A, 0x1F9C,
0x200F, 0x2083, 0x20F8, 0x216F, 0x21E6, 0x225E, 0x22D7, 0x2352,
0x23CD, 0x2449, 0x24C7, 0x2545, 0x25C5, 0x2645, 0x26C6, 0x2749,
0x27CC, 0x2851, 0x28D7, 0x295D, 0x29E5, 0x2A6D, 0x2AF7, 0x2B82,
0x2C0E, 0x2C9A, 0x2D28, 0x2DB7, 0x2E47, 0x2ED8, 0x2F6A, 0x2FFD,
0x3091, 0x3126, 0x31BC, 0x3253, 0x32EB, 0x3384, 0x341E, 0x34B9,
0x3555, 0x35F2, 0x3691, 0x3730, 0x37D0, 0x3871, 0x3913, 0x39B7,
0x3A5B, 0x3B00, 0x3BA6, 0x3C4D, 0x3CF6, 0x3D9F, 0x3E49, 0x3EF4,
0x3FA0, 0x404D, 0x40FB, 0x41AA, 0x425A, 0x430B, 0x43BD, 0x4470,
0x4523, 0x45D8, 0x468E, 0x4744, 0x47FC, 0x48B4, 0x496D, 0x4A28,
0x4AE3, 0x4B9F, 0x4C5C, 0x4D19, 0x4DD8, 0x4E97, 0x4F58, 0x5019,
0x50DB, 0x519E, 0x5262, 0x5326, 0x53EC, 0x54B2, 0x5579, 0x5641,
0x570A, 0x57D3, 0x589D, 0x5968, 0x5A34, 0x5B01, 0x5BCE, 0x5C9C,
0x5D6B, 0x5E3A, 0x5F0B, 0x5FDB, 0x60AD, 0x617F, 0x6252, 0x6326,
0x63FA, 0x64CF, 0x65A5, 0x667B, 0x6752, 0x682A, 0x6902, 0x69DA,
0x6AB4, 0x6B8D, 0x6C68, 0x6D43, 0x6E1E, 0x6EFA, 0x6FD7, 0x70B4,
0x7192, 0x7270, 0x734E, 0x742D, 0x750D, 0x75ED, 0x76CD, 0x77AE,
0x788F, 0x7970, 0x7A52, 0x7B35, 0x7C17, 0x7CFA, 0x7DDE, 0x7EC1,
0x7FA5, 0x808A, 0x816E, 0x8253, 0x8338, 0x841E, 0x8503, 0x85E9,
0x86CF, 0x87B5, 0x889C, 0x8982, 0x8A69, 0x8B50, 0x8C37, 0x8D1E,
0x8E05, 0x8EEC, 0x8FD4, 0x90BB, 0x91A3, 0x928A, 0x9372, 0x9459,
0x9541, 0x9628, 0x9710, 0x97F7, 0x98DE, 0x99C6, 0x9AAD, 0x9B94,
0x9C7B, 0x9D61, 0x9E48, 0x9F2E, 0xA015, 0xA0FB, 0xA1E0, 0xA2C6,
0xA3AB, 0xA490, 0xA575, 0xA65A, 0xA73E, 0xA822, 0xA905, 0xA9E8,
0xAACB, 0xABAE, 0xAC90, 0xAD71, 0xAE52, 0xAF33, 0xB013, 0xB0F3,
0xB1D3, 0xB2B1, 0xB390, 0xB46D, 0xB54B, 0xB627, 0xB703, 0xB7DF,
0xB8BA, 0xB994, 0xBA6D, 0xBB46, 0xBC1E, 0xBCF6, 0xBDCD, 0xBEA3,
0xBF78, 0xC04D, 0xC121, 0xC1F4, 0xC2C6, 0xC398, 0xC468, 0xC538,
0xC607, 0xC6D5, 0xC7A2, 0xC86F, 0xC93A, 0xCA04, 0xCACE, 0xCB96,
0xCC5E, 0xCD25, 0xCDEA, 0xCEAF, 0xCF72, 0xD035, 0xD0F6, 0xD1B6,
0xD275, 0xD334, 0xD3F1, 0xD4AC, 0xD567, 0xD621, 0xD6D9, 0xD790,
0xD846, 0xD8FB, 0xD9AE, 0xDA61, 0xDB12, 0xDBC1, 0xDC70, 0xDD1D,
0xDDC9, 0xDE73, 0xDF1C, 0xDFC4, 0xE06A, 0xE10F, 0xE1B3, 0xE255,
0xE2F6, 0xE396, 0xE433, 0xE4D0, 0xE56B, 0xE605, 0xE69D, 0xE733,
0xE7C8, 0xE85C, 0xE8EE, 0xE97E, 0xEA0D, 0xEA9A, 0xEB26, 0xEBB0,
0xEC39, 0xECBF, 0xED45, 0xEDC8, 0xEE4A, 0xEECB, 0xEF49, 0xEFC6,
0xF042, 0xF0BB, 0xF133, 0xF1A9, 0xF21E, 0xF290, 0xF301, 0xF370,
0xF3DE, 0xF44A, 0xF4B3, 0xF51C, 0xF582, 0xF5E6, 0xF649, 0xF6AA,
0xF709, 0xF766, 0xF7C1, 0xF81B, 0xF873, 0xF8C8, 0xF91C, 0xF96E,
0xF9BE, 0xFA0D, 0xFA59, 0xFAA3, 0xFAEC, 0xFB33, 0xFB77, 0xFBBA,
0xFBFB, 0xFC3A, 0xFC77, 0xFCB2, 0xFCEB, 0xFD22, 0xFD57, 0xFD8A,
0xFDBB, 0xFDEA, 0xFE17, 0xFE43, 0xFE6C, 0xFE93, 0xFEB8, 0xFEDC,
0xFEFD, 0xFF1C, 0xFF39, 0xFF55, 0xFF6E, 0xFF85, 0xFF9A, 0xFFAE,
0xFFBF, 0xFFCE, 0xFFDB, 0xFFE6, 0xFFEF, 0xFFF6, 0xFFFB, 0xFFFE
};
#endif
```

```
#endif


//-----------------------------------------------------------------------------
// Window Functions for NUM_FFT = 512
//-----------------------------------------------------------------------------
#if (NUM_FFT == 512)

#if (WINDOW_TYPE == 1)
// Triangle Window
unsigned int code WindowFunc[256] =
{
0x0000, 0x0100, 0x0200, 0x0300, 0x0400, 0x0500, 0x0600, 0x0700,
0x0800, 0x0900, 0x0A00, 0x0B00, 0x0C00, 0x0D00, 0x0E00, 0x0F00,
0x1000, 0x1100, 0x1200, 0x1300, 0x1400, 0x1500, 0x1600, 0x1700,
0x1800, 0x1900, 0x1A00, 0x1B00, 0x1C00, 0x1D00, 0x1E00, 0x1F00,
0x2000, 0x2100, 0x2200, 0x2300, 0x2400, 0x2500, 0x2600, 0x2700,
0x2800, 0x2900, 0x2A00, 0x2B00, 0x2C00, 0x2D00, 0x2E00, 0x2F00,
0x3000, 0x3100, 0x3200, 0x3300, 0x3400, 0x3500, 0x3600, 0x3700,
0x3800, 0x3900, 0x3A00, 0x3B00, 0x3C00, 0x3D00, 0x3E00, 0x3F00,
0x4000, 0x4100, 0x4200, 0x4300, 0x4400, 0x4500, 0x4600, 0x4700,
0x4800, 0x4900, 0x4A00, 0x4B00, 0x4C00, 0x4D00, 0x4E00, 0x4F00,
0x5000, 0x5100, 0x5200, 0x5300, 0x5400, 0x5500, 0x5600, 0x5700,
0x5800, 0x5900, 0x5A00, 0x5B00, 0x5C00, 0x5D00, 0x5E00, 0x5F00,
0x6000, 0x6100, 0x6200, 0x6300, 0x6400, 0x6500, 0x6600, 0x6700,
0x6800, 0x6900, 0x6A00, 0x6B00, 0x6C00, 0x6D00, 0x6E00, 0x6F00,
0x7000, 0x7100, 0x7200, 0x7300, 0x7400, 0x7500, 0x7600, 0x7700,
0x7800, 0x7900, 0x7A00, 0x7B00, 0x7C00, 0x7D00, 0x7E00, 0x7F00,
0x8000, 0x8100, 0x8200, 0x8300, 0x8400, 0x8500, 0x8600, 0x8700,
0x8800, 0x8900, 0x8A00, 0x8B00, 0x8C00, 0x8D00, 0x8E00, 0x8F00,
0x9000, 0x9100, 0x9200, 0x9300, 0x9400, 0x9500, 0x9600, 0x9700,
0x9800, 0x9900, 0x9A00, 0x9B00, 0x9C00, 0x9D00, 0x9E00, 0x9F00,
0xA000, 0xA100, 0xA200, 0xA300, 0xA400, 0xA500, 0xA600, 0xA700,
0xA800, 0xA900, 0xAA00, 0xAB00, 0xAC00, 0xAD00, 0xAE00, 0xAF00,
0xB000, 0xB100, 0xB200, 0xB300, 0xB400, 0xB500, 0xB600, 0xB700,
0xB800, 0xB900, 0xBA00, 0xBB00, 0xBC00, 0xBD00, 0xBE00, 0xBF00,
0xC000, 0xC100, 0xC200, 0xC300, 0xC400, 0xC500, 0xC600, 0xC700,
0xC800, 0xC900, 0xCA00, 0xCB00, 0xCC00, 0xCD00, 0xCE00, 0xCF00,
0xD000, 0xD100, 0xD200, 0xD300, 0xD400, 0xD500, 0xD600, 0xD700,
0xD800, 0xD900, 0xDA00, 0xDB00, 0xDC00, 0xDD00, 0xDE00, 0xDF00,
0xE000, 0xE100, 0xE200, 0xE300, 0xE400, 0xE500, 0xE600, 0xE700,
0xE800, 0xE900, 0xEA00, 0xEB00, 0xEC00, 0xED00, 0xEE00, 0xEF00,
0xF000, 0xF100, 0xF200, 0xF300, 0xF400, 0xF500, 0xF600, 0xF700,
0xF800, 0xF900, 0xFA00, 0xFB00, 0xFC00, 0xFD00, 0xFE00, 0xFF00
};
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[256] =
{
0x0000, 0x0002, 0x0009, 0x0016, 0x0027, 0x003D, 0x0058, 0x0078,
0x009D, 0x00C7, 0x00F6, 0x012A, 0x0162, 0x01A0, 0x01E2, 0x0229,
0x0275, 0x02C6, 0x031C, 0x0376, 0x03D6, 0x043A, 0x04A2, 0x0510,
0x0582, 0x05FA, 0x0675, 0x06F6, 0x077B, 0x0805, 0x0893, 0x0926,
0x09BE, 0x0A5A, 0x0AFB, 0x0BA0, 0x0C4A, 0x0CF8, 0x0DAA, 0x0E61,
0x0F1D, 0x0FDC, 0x10A0, 0x1169, 0x1235, 0x1306, 0x13DB, 0x14B5,
0x1592, 0x1673, 0x1759, 0x1842, 0x1930, 0x1A22, 0x1B17, 0x1C10,
0x1D0D, 0x1E0E, 0x1F13, 0x201C, 0x2128, 0x2238, 0x234B, 0x2462,
```

```
0x257D, 0x269B, 0x27BD, 0x28E2, 0x2A0A, 0x2B35, 0x2C64, 0x2D96,
0x2ECC, 0x3004, 0x3140, 0x327E, 0x33C0, 0x3504, 0x364B, 0x3796,
0x38E3, 0x3A32, 0x3B85, 0x3CDA, 0x3E31, 0x3F8C, 0x40E8, 0x4247,
0x43A9, 0x450D, 0x4673, 0x47DB, 0x4945, 0x4AB2, 0x4C21, 0x4D91,
0x4F04, 0x5078, 0x51EE, 0x5367, 0x54E0, 0x565C, 0x57D9, 0x5957,
0x5AD7, 0x5C59, 0x5DDC, 0x5F60, 0x60E6, 0x626C, 0x63F4, 0x657D,
0x6707, 0x6892, 0x6A1D, 0x6BAA, 0x6D37, 0x6EC6, 0x7054, 0x71E4,
0x7374, 0x7504, 0x7695, 0x7826, 0x79B8, 0x7B49, 0x7CDB, 0x7E6D,
0x7FFF, 0x8192, 0x8324, 0x84B6, 0x8647, 0x87D9, 0x896A, 0x8AFB,
0x8C8B, 0x8E1B, 0x8FAB, 0x9139, 0x92C8, 0x9455, 0x95E2, 0x976D,
0x98F8, 0x9A82, 0x9C0B, 0x9D93, 0x9F19, 0xA09F, 0xA223, 0xA3A6,
0xA528, 0xA6A8, 0xA826, 0xA9A3, 0xAB1F, 0xAC98, 0xAE11, 0xAF87,
0xB0FB, 0xB26E, 0xB3DE, 0xB54D, 0xB6BA, 0xB824, 0xB98C, 0xBAF2,
0xBC56, 0xBDB8, 0xBF17, 0xC073, 0xC1CE, 0xC325, 0xC47A, 0xC5CD,
0xC71C, 0xC869, 0xC9B4, 0xCAFB, 0xCC3F, 0xCD81, 0xCEBF, 0xCFFB,
0xD133, 0xD269, 0xD39B, 0xD4CA, 0xD5F5, 0xD71D, 0xD842, 0xD964,
0xDA82, 0xDB9D, 0xDCB4, 0xDDC7, 0xDED7, 0xDFE3, 0xE0EC, 0xE1F1,
0xE2F2, 0xE3EF, 0xE4E8, 0xE5DD, 0xE6CF, 0xE7BD, 0xE8A6, 0xE98C,
0xEA6D, 0xEB4A, 0xEC24, 0xECF9, 0xEDCA, 0xEE96, 0xEF5F, 0xF023,
0xF0E2, 0xF19E, 0xF255, 0xF307, 0xF3B5, 0xF45F, 0xF504, 0xF5A5,
0xF641, 0xF6D9, 0xF76C, 0xF7FA, 0xF884, 0xF909, 0xF98A, 0xFA05,
0xFA7D, 0xFAEF, 0xFB5D, 0xFBC5, 0xFC29, 0xFC89, 0xFCE3, 0xFD39,
0xFD8A, 0xFDD6, 0xFE1D, 0xFE5F, 0xFE9D, 0xFED5, 0xFF09, 0xFF38,
0xFF62, 0xFF87, 0xFFA7, 0xFFC2, 0xFFD8, 0xFFE9, 0xFFF6, 0xFFFD
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[256] =
{
0x147A, 0x147D, 0x1483, 0x148F, 0x149F, 0x14B3, 0x14CC, 0x14EA,
0x150C, 0x1532, 0x155D, 0x158D, 0x15C1, 0x15F9, 0x1636, 0x1678,
0x16BE, 0x1708, 0x1757, 0x17AA, 0x1802, 0x185E, 0x18BE, 0x1923,
0x198C, 0x19FA, 0x1A6C, 0x1AE2, 0x1B5D, 0x1BDB, 0x1C5E, 0x1CE6,
0x1D71, 0x1E01, 0x1E95, 0x1F2D, 0x1FC9, 0x2069, 0x210D, 0x21B6,
0x2262, 0x2312, 0x23C7, 0x247F, 0x253B, 0x25FC, 0x26C0, 0x2787,
0x2853, 0x2922, 0x29F6, 0x2ACC, 0x2BA7, 0x2C85, 0x2D67, 0x2E4C,
0x2F35, 0x3022, 0x3112, 0x3205, 0x32FC, 0x33F6, 0x34F3, 0x35F4,
0x36F8, 0x37FF, 0x390A, 0x3A17, 0x3B28, 0x3C3B, 0x3D52, 0x3E6C,
0x3F88, 0x40A8, 0x41CA, 0x42EF, 0x4417, 0x4541, 0x466E, 0x479E,
0x48D0, 0x4A05, 0x4B3D, 0x4C76, 0x4DB3, 0x4EF1, 0x5032, 0x5175,
0x52BA, 0x5401, 0x554B, 0x5696, 0x57E4, 0x5933, 0x5A84, 0x5BD7,
0x5D2C, 0x5E83, 0x5FDB, 0x6135, 0x6291, 0x63EE, 0x654D, 0x66AC,
0x680E, 0x6971, 0x6AD4, 0x6C3A, 0x6DA0, 0x6F07, 0x7070, 0x71D9,
0x7344, 0x74AF, 0x761B, 0x7788, 0x78F6, 0x7A64, 0x7BD3, 0x7D42,
0x7EB2, 0x8022, 0x8193, 0x8304, 0x8476, 0x85E7, 0x8759, 0x88CB,
0x8A3D, 0x8BAF, 0x8D21, 0x8E93, 0x9004, 0x9176, 0x92E7, 0x9457,
0x95C8, 0x9738, 0x98A7, 0x9A16, 0x9B84, 0x9CF2, 0x9E5F, 0x9FCB,
0xA136, 0xA2A1, 0xA40A, 0xA573, 0xA6DA, 0xA840, 0xA9A5, 0xAB09,
0xAC6C, 0xADCD, 0xAF2D, 0xB08C, 0xB1E9, 0xB345, 0xB49F, 0xB5F7,
0xB74E, 0xB8A2, 0xB9F6, 0xBB47, 0xBC96, 0xBDE4, 0xBF2F, 0xC079,
0xC1C0, 0xC305, 0xC448, 0xC589, 0xC6C7, 0xC804, 0xC93D, 0xCA75,
0xCBA9, 0xCCDC, 0xCE0C, 0xCF39, 0xD063, 0xD18B, 0xD2B0, 0xD3D2,
0xD4F2, 0xD60E, 0xD728, 0xD83E, 0xD952, 0xDA63, 0xDB70, 0xDC7B,
0xDD82, 0xDE86, 0xDF86, 0xE084, 0xE17E, 0xE275, 0xE368, 0xE458,
0xE545, 0xE62D, 0xE713, 0xE7F5, 0xE8D3, 0xE9AD, 0xEA84, 0xEB57,
0xEC27, 0xECF3, 0xEDBA, 0xEE7E, 0xEF3F, 0xEFFB, 0xF0B3, 0xF167,
```

```
0xF218, 0xF2C4, 0xF36D, 0xF411, 0xF4B1, 0xF54D, 0xF5E5, 0xF679,
0xF709, 0xF794, 0xF81B, 0xF89E, 0xF91D, 0xF998, 0xFA0E, 0xFA80,
0xFAED, 0xFB57, 0xFBBB, 0xFC1C, 0xFC78, 0xFCD0, 0xFD23, 0xFD72,
0xFDBC, 0xFE02, 0xFE44, 0xFE81, 0xFEB9, 0xFEED, 0xFF1D, 0xFF48,
0xFF6E, 0xFF90, 0xFFAE, 0xFFC7, 0xFFDB, 0xFFEB, 0xFFF6, 0xFFFD
};
#endif


#if (WINDOW_TYPE == 4)
// Blackman Window
unsigned int code WindowFunc[256] =
{
0x0000, 0x0000, 0x0003, 0x0007, 0x000E, 0x0016, 0x0020, 0x002B,
0x0039, 0x0048, 0x0059, 0x006C, 0x0080, 0x0097, 0x00AF, 0x00CA,
0x00E6, 0x0104, 0x0124, 0x0146, 0x016A, 0x0190, 0x01B9, 0x01E3,
0x020F, 0x023D, 0x026E, 0x02A0, 0x02D5, 0x030C, 0x0345, 0x0380,
0x03BE, 0x03FE, 0x0441, 0x0486, 0x04CD, 0x0516, 0x0563, 0x05B1,
0x0603, 0x0656, 0x06AD, 0x0706, 0x0762, 0x07C1, 0x0822, 0x0886,
0x08ED, 0x0957, 0x09C4, 0x0A34, 0x0AA7, 0x0B1D, 0x0B96, 0x0C12,
0x0C91, 0x0D14, 0x0D9A, 0x0E23, 0x0EAF, 0x0F3F, 0x0FD2, 0x1068,
0x1102, 0x11A0, 0x1240, 0x12E5, 0x138D, 0x1439, 0x14E8, 0x159B,
0x1652, 0x170D, 0x17CB, 0x188D, 0x1953, 0x1A1D, 0x1AEA, 0x1BBC,
0x1C91, 0x1D6B, 0x1E48, 0x1F2A, 0x200F, 0x20F8, 0x21E6, 0x22D7,
0x23CD, 0x24C7, 0x25C5, 0x26C6, 0x27CC, 0x28D7, 0x29E5, 0x2AF7,
0x2C0E, 0x2D28, 0x2E47, 0x2F6A, 0x3091, 0x31BC, 0x32EB, 0x341E,
0x3555, 0x3691, 0x37D0, 0x3913, 0x3A5B, 0x3BA6, 0x3CF6, 0x3E49,
0x3FA0, 0x40FB, 0x425A, 0x43BD, 0x4523, 0x468E, 0x47FC, 0x496D,
0x4AE3, 0x4C5C, 0x4DD8, 0x4F58, 0x50DB, 0x5262, 0x53EC, 0x5579,
0x570A, 0x589D, 0x5A34, 0x5BCE, 0x5D6B, 0x5F0B, 0x60AD, 0x6252,
0x63FA, 0x65A5, 0x6752, 0x6902, 0x6AB4, 0x6C68, 0x6E1E, 0x6FD7,
0x7192, 0x734E, 0x750D, 0x76CD, 0x788F, 0x7A52, 0x7C17, 0x7DDE,
0x7FA5, 0x816E, 0x8338, 0x8503, 0x86CF, 0x889C, 0x8A69, 0x8C37,
0x8E05, 0x8FD4, 0x91A3, 0x9372, 0x9541, 0x9710, 0x98DE, 0x9AAD,
0x9C7B, 0x9E48, 0xA015, 0xA1E0, 0xA3AB, 0xA575, 0xA73E, 0xA905,
0xAACB, 0xAC90, 0xAE52, 0xB013, 0xB1D3, 0xB390, 0xB54B, 0xB703,
0xB8BA, 0xBA6D, 0xBC1E, 0xBDCD, 0xBF78, 0xC121, 0xC2C6, 0xC468,
0xC607, 0xC7A2, 0xC93A, 0xCACE, 0xCC5E, 0xCDEA, 0xCF72, 0xD0F6,
0xD275, 0xD3F1, 0xD567, 0xD6D9, 0xD846, 0xD9AE, 0xDB12, 0xDC70,
0xDDC9, 0xDF1C, 0xE06A, 0xE1B3, 0xE2F6, 0xE433, 0xE56B, 0xE69D,
0xE7C8, 0xE8EE, 0xEA0D, 0xEB26, 0xEC39, 0xED45, 0xEE4A, 0xEF49,
0xF042, 0xF133, 0xF21E, 0xF301, 0xF3DE, 0xF4B3, 0xF582, 0xF649,
0xF709, 0xF7C1, 0xF873, 0xF91C, 0xF9BE, 0xFA59, 0xFAEC, 0xFB77,
0xFBFB, 0xFC77, 0xFCEB, 0xFD57, 0xFDBB, 0xFE17, 0xFE6C, 0xFEB8,
0xFEFD, 0xFF39, 0xFF6E, 0xFF9A, 0xFFBF, 0xFFDB, 0xFFEF, 0xFFFB
};
#endif

#endif


//-----------------------------------------------------------------------------
// Window Functions for NUM_FFT = 256
//-----------------------------------------------------------------------------
#if (NUM_FFT == 256)

#if (WINDOW_TYPE == 1)
// Triangle Window
unsigned int code WindowFunc[128] =
```

```
{
0x0000, 0x0200, 0x0400, 0x0600, 0x0800, 0x0A00, 0x0C00, 0x0E00,
0x1000, 0x1200, 0x1400, 0x1600, 0x1800, 0x1A00, 0x1C00, 0x1E00,
0x2000, 0x2200, 0x2400, 0x2600, 0x2800, 0x2A00, 0x2C00, 0x2E00,
0x3000, 0x3200, 0x3400, 0x3600, 0x3800, 0x3A00, 0x3C00, 0x3E00,
0x4000, 0x4200, 0x4400, 0x4600, 0x4800, 0x4A00, 0x4C00, 0x4E00,
0x5000, 0x5200, 0x5400, 0x5600, 0x5800, 0x5A00, 0x5C00, 0x5E00,
0x6000, 0x6200, 0x6400, 0x6600, 0x6800, 0x6A00, 0x6C00, 0x6E00,
0x7000, 0x7200, 0x7400, 0x7600, 0x7800, 0x7A00, 0x7C00, 0x7E00,
0x8000, 0x8200, 0x8400, 0x8600, 0x8800, 0x8A00, 0x8C00, 0x8E00,
0x9000, 0x9200, 0x9400, 0x9600, 0x9800, 0x9A00, 0x9C00, 0x9E00,
0xA000, 0xA200, 0xA400, 0xA600, 0xA800, 0xAA00, 0xAC00, 0xAE00,
0xB000, 0xB200, 0xB400, 0xB600, 0xB800, 0xBA00, 0xBC00, 0xBE00,
0xC000, 0xC200, 0xC400, 0xC600, 0xC800, 0xCA00, 0xCC00, 0xCE00,
0xD000, 0xD200, 0xD400, 0xD600, 0xD800, 0xDA00, 0xDC00, 0xDE00,
0xE000, 0xE200, 0xE400, 0xE600, 0xE800, 0xEA00, 0xEC00, 0xEE00,
0xF000, 0xF200, 0xF400, 0xF600, 0xF800, 0xFA00, 0xFC00, 0xFE00
};
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[128] =
{
0x0000, 0x0009, 0x0027, 0x0058, 0x009D, 0x00F6, 0x0162, 0x01E2,
0x0275, 0x031C, 0x03D6, 0x04A2, 0x0582, 0x0675, 0x077B, 0x0893,
0x09BE, 0x0AFB, 0x0C4A, 0x0DAA, 0x0F1D, 0x10A0, 0x1235, 0x13DB,
0x1592, 0x1759, 0x1930, 0x1B17, 0x1D0D, 0x1F13, 0x2128, 0x234B,
0x257D, 0x27BD, 0x2A0A, 0x2C64, 0x2ECC, 0x3140, 0x33C0, 0x364B,
0x38E3, 0x3B85, 0x3E31, 0x40E8, 0x43A9, 0x4673, 0x4945, 0x4C21,
0x4F04, 0x51EE, 0x54E0, 0x57D9, 0x5AD7, 0x5DDC, 0x60E6, 0x63F4,
0x6707, 0x6A1D, 0x6D37, 0x7054, 0x7374, 0x7695, 0x79B8, 0x7CDB,
0x7FFF, 0x8324, 0x8647, 0x896A, 0x8C8B, 0x8FAB, 0x92C8, 0x95E2,
0x98F8, 0x9C0B, 0x9F19, 0xA223, 0xA528, 0xA826, 0xAB1F, 0xAE11,
0xB0FB, 0xB3DE, 0xB6BA, 0xB98C, 0xBC56, 0xBF17, 0xC1CE, 0xC47A,
0xC71C, 0xC9B4, 0xCC3F, 0xCEBF, 0xD133, 0xD39B, 0xD5F5, 0xD842,
0xDA82, 0xDCB4, 0xDED7, 0xE0EC, 0xE2F2, 0xE4E8, 0xE6CF, 0xE8A6,
0xEA6D, 0xEC24, 0xEDCA, 0xEF5F, 0xF0E2, 0xF255, 0xF3B5, 0xF504,
0xF641, 0xF76C, 0xF884, 0xF98A, 0xFA7D, 0xFB5D, 0xFC29, 0xFCE3,
0xFD8A, 0xFE1D, 0xFE9D, 0xFF09, 0xFF62, 0xFFA7, 0xFFD8, 0xFFF6
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[128] =
{
0x147A, 0x1483, 0x149F, 0x14CC, 0x150C, 0x155D, 0x15C1, 0x1636,
0x16BE, 0x1757, 0x1802, 0x18BE, 0x198C, 0x1A6C, 0x1B5D, 0x1C5E,
0x1D71, 0x1E95, 0x1FC9, 0x210D, 0x2262, 0x23C7, 0x253B, 0x26C0,
0x2853, 0x29F6, 0x2BA7, 0x2D67, 0x2F35, 0x3112, 0x32FC, 0x34F3,
0x36F8, 0x390A, 0x3B28, 0x3D52, 0x3F88, 0x41CA, 0x4417, 0x466E,
0x48D0, 0x4B3D, 0x4DB3, 0x5032, 0x52BA, 0x554B, 0x57E4, 0x5A84,
0x5D2C, 0x5FDB, 0x6291, 0x654D, 0x680E, 0x6AD4, 0x6DA0, 0x7070,
0x7344, 0x761B, 0x78F6, 0x7BD3, 0x7EB2, 0x8193, 0x8476, 0x8759,
0x8A3D, 0x8D21, 0x9004, 0x92E7, 0x95C8, 0x98A7, 0x9B84, 0x9E5F,
0xA136, 0xA40A, 0xA6DA, 0xA9A5, 0xAC6C, 0xAF2D, 0xB1E9, 0xB49F,
0xB74E, 0xB9F6, 0xBC96, 0xBF2F, 0xC1C0, 0xC448, 0xC6C7, 0xC93D,
```

```
0xCBA9, 0xCE0C, 0xD063, 0xD2B0, 0xD4F2, 0xD728, 0xD952, 0xDB70,
0xDD82, 0xDF86, 0xE17E, 0xE368, 0xE545, 0xE713, 0xE8D3, 0xEA84,
0xEC27, 0xEDBA, 0xEF3F, 0xF0B3, 0xF218, 0xF36D, 0xF4B1, 0xF5E5,
0xF709, 0xF81B, 0xF91D, 0xFA0E, 0xFAED, 0xFBBB, 0xFC78, 0xFD23,
0xFDBC, 0xFE44, 0xFEB9, 0xFF1D, 0xFF6E, 0xFFAE, 0xFFDB, 0xFFF6
};
#endif


#if (WINDOW_TYPE == 4)
// Blackman Window
unsigned int code WindowFunc[128] =
{
0x0000, 0x0003, 0x000E, 0x0020, 0x0039, 0x0059, 0x0080, 0x00AF,
0x00E6, 0x0124, 0x016A, 0x01B9, 0x020F, 0x026E, 0x02D5, 0x0345,
0x03BE, 0x0441, 0x04CD, 0x0563, 0x0603, 0x06AD, 0x0762, 0x0822,
0x08ED, 0x09C4, 0x0AA7, 0x0B96, 0x0C91, 0x0D9A, 0x0EAF, 0x0FD2,
0x1102, 0x1240, 0x138D, 0x14E8, 0x1652, 0x17CB, 0x1953, 0x1AEA,
0x1C91, 0x1E48, 0x200F, 0x21E6, 0x23CD, 0x25C5, 0x27CC, 0x29E5,
0x2C0E, 0x2E47, 0x3091, 0x32EB, 0x3555, 0x37D0, 0x3A5B, 0x3CF6,
0x3FA0, 0x425A, 0x4523, 0x47FC, 0x4AE3, 0x4DD8, 0x50DB, 0x53EC,
0x570A, 0x5A34, 0x5D6B, 0x60AD, 0x63FA, 0x6752, 0x6AB4, 0x6E1E,
0x7192, 0x750D, 0x788F, 0x7C17, 0x7FA5, 0x8338, 0x86CF, 0x8A69,
0x8E05, 0x91A3, 0x9541, 0x98DE, 0x9C7B, 0xA015, 0xA3AB, 0xA73E,
0xAACB, 0xAE52, 0xB1D3, 0xB54B, 0xB8BA, 0xBC1E, 0xBF78, 0xC2C6,
0xC607, 0xC93A, 0xCC5E, 0xCF72, 0xD275, 0xD567, 0xD846, 0xDB12,
0xDDC9, 0xE06A, 0xE2F6, 0xE56B, 0xE7C8, 0xEA0D, 0xEC39, 0xEE4A,
0xF042, 0xF21E, 0xF3DE, 0xF582, 0xF709, 0xF873, 0xF9BE, 0xFAEC,
0xFBFB, 0xFCEB, 0xFDBB, 0xFE6C, 0xFEFD, 0xFF6E, 0xFFBF, 0xFFEF
};
#endif

#endif

//-----------------------------------------------------------------------------
// Window Functions for NUM_FFT = 128
//-----------------------------------------------------------------------------
#if (NUM_FFT == 128)

#if (WINDOW_TYPE == 1)
// Triangle Window
unsigned int code WindowFunc[64] =
{
0x0000, 0x0400, 0x0800, 0x0C00, 0x1000, 0x1400, 0x1800, 0x1C00,
0x2000, 0x2400, 0x2800, 0x2C00, 0x3000, 0x3400, 0x3800, 0x3C00,
0x4000, 0x4400, 0x4800, 0x4C00, 0x5000, 0x5400, 0x5800, 0x5C00,
0x6000, 0x6400, 0x6800, 0x6C00, 0x7000, 0x7400, 0x7800, 0x7C00,
0x8000, 0x8400, 0x8800, 0x8C00, 0x9000, 0x9400, 0x9800, 0x9C00,
0xA000, 0xA400, 0xA800, 0xAC00, 0xB000, 0xB400, 0xB800, 0xBC00,
0xC000, 0xC400, 0xC800, 0xCC00, 0xD000, 0xD400, 0xD800, 0xDC00,
0xE000, 0xE400, 0xE800, 0xEC00, 0xF000, 0xF400, 0xF800, 0xFC00
};
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[64] =
{
0x0000, 0x0027, 0x009D, 0x0162, 0x0275, 0x03D6, 0x0582, 0x077B,
```

```
0x09BE, 0x0C4A, 0x0F1D, 0x1235, 0x1592, 0x1930, 0x1D0D, 0x2128,
0x257D, 0x2A0A, 0x2ECC, 0x33C0, 0x38E3, 0x3E31, 0x43A9, 0x4945,
0x4F04, 0x54E0, 0x5AD7, 0x60E6, 0x6707, 0x6D37, 0x7374, 0x79B8,
0x7FFF, 0x8647, 0x8C8B, 0x92C8, 0x98F8, 0x9F19, 0xA528, 0xAB1F,
0xB0FB, 0xB6BA, 0xBC56, 0xC1CE, 0xC71C, 0xCC3F, 0xD133, 0xD5F5,
0xDA82, 0xDED7, 0xE2F2, 0xE6CF, 0xEA6D, 0xEDCA, 0xF0E2, 0xF3B5,
0xF641, 0xF884, 0xFA7D, 0xFC29, 0xFD8A, 0xFE9D, 0xFF62, 0xFFD8
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[64] =
{
0x147A, 0x149F, 0x150C, 0x15C1, 0x16BE, 0x1802, 0x198C, 0x1B5D,
0x1D71, 0x1FC9, 0x2262, 0x253B, 0x2853, 0x2BA7, 0x2F35, 0x32FC,
0x36F8, 0x3B28, 0x3F88, 0x4417, 0x48D0, 0x4DB3, 0x52BA, 0x57E4,
0x5D2C, 0x6291, 0x680E, 0x6DA0, 0x7344, 0x78F6, 0x7EB2, 0x8476,
0x8A3D, 0x9004, 0x95C8, 0x9B84, 0xA136, 0xA6DA, 0xAC6C, 0xB1E9,
0xB74E, 0xBC96, 0xC1C0, 0xC6C7, 0xCBA9, 0xD063, 0xD4F2, 0xD952,
0xDD82, 0xE17E, 0xE545, 0xE8D3, 0xEC27, 0xEF3F, 0xF218, 0xF4B1,
0xF709, 0xF91D, 0xFAED, 0xFC78, 0xFDBC, 0xFEB9, 0xFF6E, 0xFFDB
};
#endif


#if (WINDOW_TYPE == 4)
// Blackman Window
unsigned int code WindowFunc[64] =
{
0x0000, 0x000E, 0x0039, 0x0080, 0x00E6, 0x016A, 0x020F, 0x02D5,
0x03BE, 0x04CD, 0x0603, 0x0762, 0x08ED, 0x0AA7, 0x0C91, 0x0EAF,
0x1102, 0x138D, 0x1652, 0x1953, 0x1C91, 0x200F, 0x23CD, 0x27CC,
0x2C0E, 0x3091, 0x3555, 0x3A5B, 0x3FA0, 0x4523, 0x4AE3, 0x50DB,
0x570A, 0x5D6B, 0x63FA, 0x6AB4, 0x7192, 0x788F, 0x7FA5, 0x86CF,
0x8E05, 0x9541, 0x9C7B, 0xA3AB, 0xAACB, 0xB1D3, 0xB8BA, 0xBF78,
0xC607, 0xCC5E, 0xD275, 0xD846, 0xDDC9, 0xE2F6, 0xE7C8, 0xEC39,
0xF042, 0xF3DE, 0xF709, 0xF9BE, 0xFBFB, 0xFDBB, 0xFEFD, 0xFFBF
};
#endif

#endif


//-----------------------------------------------------------------------------
// Window Functions for NUM_FFT = 64
//-----------------------------------------------------------------------------
#if (NUM_FFT == 64)

#if (WINDOW_TYPE == 1)
// Triangle Window
unsigned int code WindowFunc[32] =
{
0x0000, 0x0800, 0x1000, 0x1800, 0x2000, 0x2800, 0x3000, 0x3800,
0x4000, 0x4800, 0x5000, 0x5800, 0x6000, 0x6800, 0x7000, 0x7800,
0x8000, 0x8800, 0x9000, 0x9800, 0xA000, 0xA800, 0xB000, 0xB800,
0xC000, 0xC800, 0xD000, 0xD800, 0xE000, 0xE800, 0xF000, 0xF800
};
```

```
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[32] =
{
0x0000, 0x009D, 0x0275, 0x0582, 0x09BE, 0x0F1D, 0x1592, 0x1D0D,
0x257D, 0x2ECC, 0x38E3, 0x43A9, 0x4F04, 0x5AD7, 0x6707, 0x7374,
0x7FFF, 0x8C8B, 0x98F8, 0xA528, 0xB0FB, 0xBC56, 0xC71C, 0xD133,
0xDA82, 0xE2F2, 0xEA6D, 0xF0E2, 0xF641, 0xFA7D, 0xFD8A, 0xFF62
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[32] =
{
0x147A, 0x150C, 0x16BE, 0x198C, 0x1D71, 0x2262, 0x2853, 0x2F35,
0x36F8, 0x3F88, 0x48D0, 0x52BA, 0x5D2C, 0x680E, 0x7344, 0x7EB2,
0x8A3D, 0x95C8, 0xA136, 0xAC6C, 0xB74E, 0xC1C0, 0xCBA9, 0xD4F2,
0xDD82, 0xE545, 0xEC27, 0xF218, 0xF709, 0xFAED, 0xFDBC, 0xFF6E
};
#endif


#if (WINDOW_TYPE == 4)
// Blackman Window
unsigned int code WindowFunc[32] =
{
0x0000, 0x0039, 0x00E6, 0x020F, 0x03BE, 0x0603, 0x08ED, 0x0C91,
0x1102, 0x1652, 0x1C91, 0x23CD, 0x2C0E, 0x3555, 0x3FA0, 0x4AE3,
0x570A, 0x63FA, 0x7192, 0x7FA5, 0x8E05, 0x9C7B, 0xAACB, 0xB8BA,
0xC607, 0xD275, 0xDDC9, 0xE7C8, 0xF042, 0xF709, 0xFBFB, 0xFEFD
};
#endif

#endif


//-------------------------------------------------------------------------
// Window Functions for NUM_FFT = 32
//-------------------------------------------------------------------------
#if (NUM_FFT == 32)

#if (WINDOW_TYPE == 1)
// Triangle Window
unsigned int code WindowFunc[16] =
{
0x0000, 0x1000, 0x2000, 0x3000, 0x4000, 0x5000, 0x6000, 0x7000,
0x8000, 0x9000, 0xA000, 0xB000, 0xC000, 0xD000, 0xE000, 0xF000
};
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[16] =
{
0x0000, 0x0275, 0x09BE, 0x1592, 0x257D, 0x38E3, 0x4F04, 0x6707,
```

```
0x7FFF, 0x98F8, 0xB0FB, 0xC71C, 0xDA82, 0xEA6D, 0xF641, 0xFD8A
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[16] =
{
0x147A, 0x16BE, 0x1D71, 0x2853, 0x36F8, 0x48D0, 0x5D2C, 0x7344,
0x8A3D, 0xA136, 0xB74E, 0xCBA9, 0xDD82, 0xEC27, 0xF709, 0xFDBC
};
#endif


#if (WINDOW_TYPE == 4)
// Blackman Window
unsigned int code WindowFunc[16] =
{
0x0000, 0x00E6, 0x03BE, 0x08ED, 0x1102, 0x1C91, 0x2C0E, 0x3FA0,
0x570A, 0x7192, 0x8E05, 0xAACB, 0xC607, 0xDDC9, 0xF042, 0xFBFB
};
#endif

#endif


//-----------------------------------------------------------------------------
// Window Functions for NUM_FFT = 16
//-----------------------------------------------------------------------------
#if (NUM_FFT == 16)

#if (WINDOW_TYPE == 1)
// Triangle Window
unsigned int code WindowFunc[8] =
{
0x0000, 0x2000, 0x4000, 0x6000, 0x8000, 0xA000, 0xC000, 0xE000
};
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[8] =
{
0x0000, 0x09BE, 0x257D, 0x4F04, 0x7FFF, 0xB0FB, 0xDA82, 0xF641
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[8] =
{
0x147A, 0x1D71, 0x36F8, 0x5D2C, 0x8A3D, 0xB74E, 0xDD82, 0xF709
};
#endif


#if (WINDOW_TYPE == 4)
```

```
// Blackman Window
unsigned int code WindowFunc[8] =
{
0x0000, 0x03BE, 0x1102, 0x2C0E, 0x570A, 0x8E05, 0xC607, 0xF042
};
#endif

#endif


//------------------------------------------------------------------------
// Window Functions for NUM_FFT = 8
//------------------------------------------------------------------------
#if (NUM_FFT == 8)

#if (WINDOW_TYPE == 1)
// Triangle Window
unsigned int code WindowFunc[4] =
{
0x0000, 0x4000, 0x8000, 0xC000
};
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[4] =
{
0x0000, 0x257D, 0x7FFF, 0xDA82
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[4] =
{
0x147A, 0x36F8, 0x8A3D, 0xDD82
};
#endif


#if (WINDOW_TYPE == 4)
// Blackman Window
unsigned int code WindowFunc[4] =
{
0x0000, 0x1102, 0x570A, 0xC607
};
#endif

#endif


//------------------------------------------------------------------------
// Window Functions for NUM_FFT = 4
//------------------------------------------------------------------------
#if (NUM_FFT == 4)

#if (WINDOW_TYPE == 1)
// Triangle Window
```

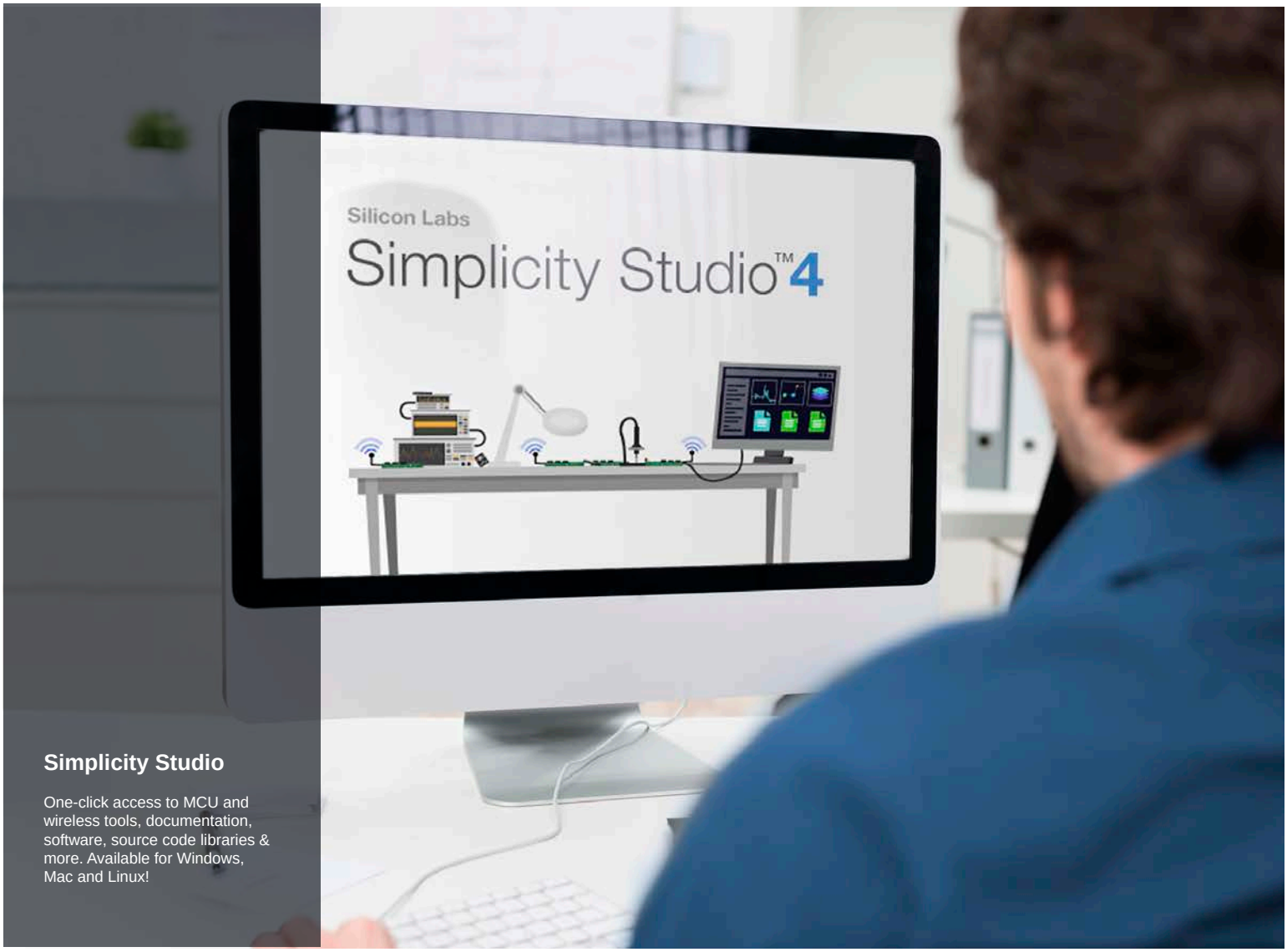
```
unsigned int code WindowFunc[2] =
{
0x0000, 0x8000
};
#endif

#if (WINDOW_TYPE == 2)
// Hanning Window
unsigned int code WindowFunc[2] =
{
0x0000, 0x7FFF
};
#endif


#if (WINDOW_TYPE == 3)
// Hamming Window
unsigned int code WindowFunc[2] =
{
0x147A, 0x8A3D
};
#endif


#if (WINDOW_TYPE == 4)
// Blackman Window
unsigned int code WindowFunc[2] =
{
0x0000, 0x570A
};
#endif

#endif
```



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com