SPECTRUM ANALYZER: PROCESSOR EXERCISE USING C LANGUAGE WITH C

INTRODUCTION

Matt Kleffner

Based on Spectrum Analyzer: Processor Exercise Using C Language† by
Matthew Berry

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License ‡

Abstract

This module describes a processor exercise in which students implement a spectrum analyzer using
mixed C and assembly code. Students are to acquire a block of 1024 samples, apply a Hamming window,
compute a length-1024 Discrete Fourier Transform using provided Fast Fourier Transform code, and
display the magnitude-squared spectrum on an oscilloscope.

1 Implementation

As this is your first experience with the C environment, you will have the option to add most of the required
code in C or assembly. A C skeleton will provide access to input samples, output samples, and interrupt
handling code. You will add code to transfer the inputs and outputs (in blocks at a time), apply a hamming
window, compute the magnitude-squared spectrum, and include a trigger pulse. After the hamming window
is created, either an assembly or C module that bit reverses the input and performs an FFT calculation is
called.

As your spectrum analyzer works on a block of samples at a time, you will need to use interrupts to
pause your processing while samples are transferred from/to the CODEC (A/D and D/A) buffer. Fortunately,
the interrupt handling routines have been written for you in a C shell program available at
v:\ece420\54x\dspclib\lab4main.c and the core code.

1.1 Interrupt Basics

Interrupts are an essential part of the operation of any microprocessor. They are particularly important
in embedded applications where DSPs are often used. Hardware interrupts provide a way for interacting
with external devices while the processor executes code. For example, in a key entry system, a key press

†http://cnx.org/content/m10658/2.8/
‡http://creativecommons.org/licenses/by/1.0
would generate a hardware interrupt. The system code would then jump to a specified location in program memory where a routine could process the key input. Interrupts provide an alternative to polling. Instead of checking for key presses at a predetermined rate (requires a clock), the system could be busy executing other code. On the TI-C54x DSP, interrupts provide a convenient way to transfer blocks of data to/from the CODEC in a timely fashion.

1.2 Interrupt Handling

The lab4main.c code and the core code are intended to make your interaction with the hardware much simpler. As there was a core file for working in the assembly environment (Labs 0-3), there is a core file for the C environment (V:\ece420\54x\dsplib\core.asm) which handles the interrupts from the CODEC (A/D and D/A) and the serial port. Here, we will describe the important aspects of the core code necessary to complete the assignment.

At the heart of the hardware interaction is the auto-buffering serial port. In the auto-buffering serial mode, the TI-C54x processor is able to do processing uninterrupted while samples are transferred to/from a buffer of length BlockLen = 64 samples. However, the spectrum analyzer to be implemented in this lab works over a block of N = 1024 samples. If it were possible to compute a 1024-point FFT in the sample time of one BlockLen, then no additional interrupt handling routines would be necessary. Samples could be collected in a 1024-length buffer and a 1024-point FFT could be computed uninterrupted while the auto-buffering buffer fills. Unfortunately, the DSP is not fast enough to accomplish this task.

We now provide an explanation of the shell C program lab4main.c listed in Appendix A (Section 3: Appendix A). The lab4main.c file contains the function interrupt void irq and a main program. The main program is an infinite loop over blocks of N = 1024 samples. Note that while the DSP is executing instructions in this loop, interrupts occur every BlockLen samples. Inside the infinite loop, you will insert code to do the operations which follow. Although each of these operations may be performed in C or assembly, we suggest you follow the guidelines suggested.

1. Transfer inputs and outputs (C)
2. Apply a Hamming Window (C/assembly)
3. Bit-reverse the input (C and assembly)
4. Apply an N-point FFT (C and assembly)
5. Compute the magnitude-squared spectrum (C/assembly)
6. Include a trigger pulse (C/assembly)

The function WaitAudio is an assembly function in the core code which handles the CODEC interrupts. An interrupt from the CODEC occurs every BlockLen samples. The SetAudioInterrupt(irq) call in the main program tells the core code to jump to the irq function when an interrupt occurs. In the irq function, BlockLen samples of the A/D input in Rcvptr (channel 1) are written to a length Ninputs buffer, and BlockLen of the output samples in the outputs buffer are written to the D/A output via Xmitptr on channel 2. In C, pointers may be used as array names so that Xmitptr[0] is the first word pointed to by Xmitptr. As in the assembly core, the input samples are not in consecutive order. The right and left inputs are offset from Rcvptr respectively by 4i and 4i + 2, i = 0, ..., BlockLen - 1. The six output channels are accessed consecutively as offsets from Xmitptr. On channel 1 of the output, the input is echoed out. You are to fill the buffer outputs with the windowed magnitude-squared FFT values by performing the operations listed above.

In the main code, the while(!input_full); loop waits for N samples to collect in the inputs buffer. Next, the N inputs and outputs must be transferred. You are to write this portion of code. This portion of code is to be done first, within BlockLen sample times; otherwise the first BlockLen of samples of output would not be available on time. Once this loop is finished, the lengthy processing of the FFT can continue. During this processing, the DSP is interrupted every BlockLen samples to transfer samples. Once this processing is over, the infinite loop returns to while(!input_full); to wait for N samples to finish collecting.
The flow diagram in Figure 1 summarizes the operation of the interrupt handling routine.

![Flow diagram](image)

**Figure 1**: Overall program flow of the main function and the interrupt handling function. (a) main function (b) interrupt handler

### 1.3 Assembly FFT Routine

As the list of operations indicates, bit-reversal and FFT computation are to be done in both C and assembly. For the assembly version, make sure that the line defining \texttt{C\_FFT} is commented in \texttt{lab4main.c}. We are providing you with a shell assembly file, available at \texttt{v:\\ece420\\54x\\dspclib\\c\_fft\_given.asm} and shown in Appendix B (Section 4: Appendix B.), containing many useful declarations and some code. The code for performing bit-reversal and other declarations needed for the FFT routine are also provided in this section. However, we would like you to enter this code manually, as you will be expected to understand its operation.

The assembly file \texttt{c\_fft\_given.asm} contains two main parts, the data section starting with \texttt{.sect ".data"} and the program section starting with \texttt{.sect ".text"}. Every function and variable accessed in C must be preceded by a single underscore \_ in assembly and a \texttt{.global _name} must be placed in the assembly file for linking. In this example, \texttt{bit\_rev\_fft} is an assembly function called from the C program with a label \texttt{.bit\_rev\_fft} in the text portion of the assembly file and a \texttt{.global \_bit\_rev\_fft} declaration. In each assembly function, the macro \texttt{ENTER\_ASM} is called upon entering and \texttt{LEAVE\_ASM} is called upon exiting. These macros are defined in \texttt{v:\\ece420\\54x\\dspclib\\core.inc}. The \texttt{ENTER\_ASM} macro saves the status registers and AR1, AR6, and AR7 when entering a function as required by the register use conventions. The \texttt{ENTER\_ASM} macro also sets the status registers to the assembly conventions we have been using (i.e., FRCT = 1

http://cnx.org/content/m11827/1.5/
for fractional arithmetic and CPL=0 for DP referenced addressing). The LEAVE_ASM macro just restores the saved registers.

### 1.3.1 Parameter Passing

The parameter passing convention between assembly and C is simple for single input, single output assembly functions. From a C program, the input to an assembly program is in the low part of accumulator A with the output returned in the same place. When more than one parameter is passed to an assembly function, the parameters are passed on the stack (see the core file description for more information). We suggest that you avoid passing or returning more than one parameter. Instead, use global memory addresses to pass in or return more than one parameter. Another alternative is to pass a pointer to the start of a buffer intended for passing and returning parameters.

### 1.3.2 Registers Modified

When entering and leaving an assembly function, the ENTER_ASM and LEAVE_ASM macros ensure that certain registers are saved and restored. Since the C program may use any and all registers, the state of a register cannot be expected to remain the same between calls to assembly function(s). Therefore, any information that needs to be preserved across calls to an assembly function must be saved to memory!

Now, we explain how to use the FFT routine provided by TI for the C54x. The FFT routine fft.asm located in v:\ece420\54x\dsplib\ computes an in-place, complex FFT. The length of the FFT is defined as a label K_FFT_SIZE and the algorithm assumes that the input starts at data memory location _fft_data.

To have your code assemble for an $N$-point FFT, you will have to include the following label definitions in your assembly code.

```
N .set 1024
K_FFT_SIZE .set N ; size of FFT
K_LOGN .set 10 ; number of stages (log_2(N))
```

In addition to defining these constants, you will have to include twiddle-factor tables for the FFT. These tables (twiddle1\(^1\) and twiddle2\(^2\) ) are available in the shared directory v:\ece420\54x\dsplib\. Note that the tables are each $N$ points long representing values from 0 to just shy of 180 degrees and must be accessed using a circular pointer. To include these tables at the proper location in memory with the appropriate labels referenced by the FFT, use the following

```
.sect ".data"
.align 1024
sine .copy "v:\ece420\54x\dsplib\twiddle1"
.align 1024
cosine .copy "v:\ece420\54x\dsplib\twiddle2"
```

The FFT provided requires that the input be in bit-reversed order, with alternating real and imaginary components. Bit-reversed addressing is a convenient way to order input $x[n]$ into a FFT so that the output

---

\(^1\)http://cnx.org/content/m11827/latest/TWIDDLE1

\(^2\)http://cnx.org/content/m11827/latest/TWIDDLE2

http://cnx.org/content/m11827/1.5/
$X(k)$ is in sequential order (i.e., $X(0)$, $X(1)$, $\ldots$, $X(N-1)$ for an $N$-point FFT). The following table illustrates the bit-reversed order for an eight-point sequence.

<table>
<thead>
<tr>
<th>Input Order</th>
<th>Binary Representation</th>
<th>Bit-Reversed Representation</th>
<th>Output Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 1

The following routine performs the bit-reversed reordering of the input data. The routine assumes that the input is stored in data memory starting at the location labeled `_bit_rev_data`, which must be aligned to the least power of two greater than the input buffer length, and consists of alternating real and imaginary parts. Because our input data is going to be purely real in this lab, you will have to make sure that you set the imaginary parts to zero by zeroing out every other memory location.

```assembly
1 bit_rev:
2 STM #_bit_rev_data,AR3 ; AR3 -> original input
3 STM #_fft_data,AR7 ; AR7 -> data processing buffer
4 MVMM AR7,AR2 ; AR2 -> bit-reversed data
5 STM #K_FFT_SIZE-1,BRC
6 RPTBD bit_rev_end-1
7 STM #K_FFT_SIZE,AR0 ; AR0 = 1/2 size of circ buffer
8 MVDD *AR3+,*AR2+  
9 MVDD *AR3-,*AR2+
10 MAR *AR3+0B
11 bit_rev_end:
12 NOP
13 RET
```

As mentioned, in the above code `_bit_rev_data` is a label indicating the start of the input data and `_fft_data` is a label indicating the start of a circular buffer where the bit-reversed data will be written. Note that although AR7 is not used by the bit-reversed routine directly, it is used extensively in the FFT routine to keep track of the start of the FFT data space.

In general, to have a pointer index memory in bit-reversed order, the AR0 register needs to be set to one-half the length of the circular buffer; a statement such as ARx+0B is used to move the ARx pointer to the next location. For more information regarding the bit-reversed addressing mode, refer to page 5-18 in the TI-54x CPU and Peripherals manual[9]. Is it possible to bit-reverse a buffer in place? For a diagram of
the ordering of the data expected by the FFT routine, see Figure 4-10 in the TI-54x Applications Guide\[?\]. Note that the FFT code uses all the pointers available and does not restore the pointers to their original values.

1.4 C FFT Routine

A bit-reversing and FFT routine have also been provided in lab4fft.c, listed in Appendix C (Section 5: Appendix C). Again, make sure you understand how the bit reversal is taking place. In lab4main.c, the line defining C_FFT must not be commented for use of the C FFT routine. The sine tables (twiddle factors) are located in sinetables.h\[3\]. This fft requires its inputs in two buffers, the real buffer real and the imaginary buffer imag, and the output is placed in the same buffers. The length of the FFT, N, and logN are defined in lab4.h, which is also listed in Appendix C (Section 5: Appendix C). When experimenting with the C FFT make sure you modify these length values instead of the ones in the assembly code and lab4main.c!

1.5 Creating the Window

As mentioned, you will be using the FFT to compute the spectrum of a windowed input. For your implementation you will need to create a 1024-point Hamming window. First, create a Hamming window in matlab using the function hamming. For the assembly FFT, use save_coef to save the window to a file that can then be included in your code with the .copy directive. For the C FFT, use the matlab function write_intvector_headerfile\[4\] with name set to ‘window’ and elemperline set to 8 to create the header file that is included in lab4main.c.

1.6 Displaying the Spectrum

Once the DFT has been computed, you must calculate the squared magnitude of the spectrum for display.

\[
(|X(k)|)^2 = (\Re(X(k)))^2 + (\Im(X(k)))^2
\]  

(1)

You may find the assembly instructions squar and squra useful in implementing (1).

Because the squared magnitude is always nonnegative, you can replace one of the magnitude values with a -1.0 as a trigger pulse for display on the oscilloscope. This is easily performed by replacing the DC term \(k = 0\) with a -1.0 when copying the magnitude values to the output buffer. The trigger pulse is necessary for the oscilloscope to lock to a specific point in the spectrum and keep the spectrum fixed on the scope.

1.7 Intrinsics

If you are planning on writing some of the code in C, then you may be forced to use intrinsics. Intrinsic instructions provide a way to use assembly instructions directly in C. An example of an intrinsic instruction is bit_rev_data[0]=_smpyr(bit_rev_data[0],window[0]) which performs the assembly signed multiply round instruction. You may also find the _lsmpy instruction useful. For more information on intrinsics, see page 6-22 of the TI-C54x Optimizing C/C++ Compiler User’s Guide\[?\].

The following lines of code were borrowed from the C FFT to serve as an example of arithmetic operations in C. Save this code in a file called mathex.c and compile this file. Look at the resulting assembly file and investigate the differences between each block. Be sure to reference the compiler user’s guide to find out what the state of the FRCT and OVM bits are. Does each block work properly for all possible values?

```c
void main(void)
```

\[3\]http://cnx.org/content/m11827/latest/sinetables.h

\[4\]http://cnx.org/content/m11827/latest/write_intvector_headerfile.m

http://cnx.org/content/m11827/1.5/
\{
    int s1, s2;
    int t1, t2;
    int i1, i2;
    int n1 = 16383, n2 = 16382, n3 = 16381, n4 = 16380;

    /* Code for standard 32-bit hardware, */
    /* with x,y limited to 16 bits */
    s1 = (n1*n2 + n3*n4) >> 15;
    s2 = (n1 + n2) >> 1;

    /* Code for TI TMS5000 series */
    t1 = ((long int)(n1*n2) + (long int)(n3*n4)) >> 15;
    t2 = ((long int)n1 + (long int)n2) >> 1;

    /* Intrinsic code for TMS320C54X series */
    i1 = _sadd(_smpy(n1, n2), _smpy(n3, n4));
    i2 = _sshl(_sadd(n1, n2),-1);
\}

1.8 Compiling and Linking

A working program can be produced by compiling the C code and linking assembly modules and the core module. The compiler translates C code to a relocatable assembly form. The linker assigns physical addresses on the DSP to the relocatable data and code segments, resolves .global references and links runtime libraries.

The procedure for compiling C code and linking assembly modules has been automated for you in the batch file \texttt{v:\ece420\54x\dsptools\c_asm.bat}. The name of the first file becomes the name of the executable. Once you have completed \texttt{lab4main.c} and \texttt{c_fft_given.asm}, type \texttt{c_asm lab4main.c c_fft_given.asm} to produce a \texttt{lab4main.out} file to be loaded onto the DSP. For the C FFT type \texttt{c_asm lab4main.c lab4fft.c} to produce \texttt{lab4main.out}. Load the output file onto the DSP as usual and confirm that valid FFTs are calculated. Once valid output is obtained, measure how many clock cycles it takes to compute both the assembly and C FFT.

2 Quiz Information

From your prelab experiments, you should be able to describe the effect of windowing and zero-padding on FFT spectral analysis. In your DSP system, experiment with different inputs, changing $N$ and the type of window. Can you explain what happens as the input frequency is increased beyond the Nyquist rate? Does the $\left| X (k) \right|^2$ coincide with what you expect from Matlab? What is the relationship between the observed spectrum and the DTFT? What would happen if the FFT calculation takes longer than it takes to fill inputs with $N$ samples? How long does it take to compute each FFT? What are the tradeoffs between writing code in C versus assembly?

3 Appendix A:

\texttt{lab4main.c}^{5}

---

\texttt{^5http://cnx.org/content/m11827/1.5/}

\texttt{http://cnx.org/content/m11827/1.5/}
#include "v:/ece420/54x/dspclib/core.h"

/* comment the next line to use assembly fft */
#define C_FFT

#include "window.h"
#include "lab4.h" /* Number of C FFT points defined here */

/* function defined in lab4fft.c */
#include "v:/ece420/54x/dspclib/window.h"
#include "lab4.h" /* Number of C FFT points defined here */

/* function defined in lab4fft.c */
void fft(void);

/* FFT data buffers */
int real[N]; /* Real part of data */
int imag[N]; /* Imaginary part of data */

#define C_FFT /* Use C FFT */
#include "window.h"
#include "lab4.h" /* Number of C FFT points defined here */

/* function defined in lab4fft.c */
void fft(void);

/* FFT data buffers */
int real[N]; /* Real part of data */
int imag[N]; /* Imaginary part of data */

#else /* Use assembly FFT */
#define N 1024 /* Number of assembly FFT points */

/* Function defined by c_fft_given.asm */
void bit_rev_fft(void);

/* FFT data buffers (declared in c_fft_given.asm) */
extern int bit_rev_data[N*2]; /* Data input for bit-reverse function */
extern int fft_data[N*2]; /* In-place FFT & Output array */
extern int window[N]; /* The Hamming window */

#endif /* C_FFT */

/* Our input/output buffers */
int inputs[N];
int outputs[N];

volatile int input_full = 0; /* volatile means interrupt changes it */
int count = 0;

interrupt void irq(void)
{
    int *Xmitptr,*Rcvptr; /* pointers to Xmit & Rcv Bufs */
    int i;

http://cnx.org/content/m11827/1.5/
static int in_irq = 0;  /* Flag to prevent reentrance */

/* Make sure we're not in the interrupt (should never happen) */
if( in_irq )
  return;

/* Mark we're processing, and enable interrupts */
in_irq = 1;
enable_irq();

/* The following waitaudio call is guaranteed not to
   actually wait; it will simply return the pointers. */
WaitAudio(&Rcvptr,&Xmitptr);

/* input_full should never be true... */
if( !input_full )
{
  for (i=0; i<BlockLen; i++)
  {
    /* Save input, and echo to channel 1 */
    inputs[count] = Xmitptr[6*i] = Rcvptr[4*i];
    /* Send FFT output to channel 2 */
    Xmitptr[6*i+1] = outputs[count];
    count++;
  }
}

/* Have we collected enough data yet? */
if( count >= N )
  input_full = 1;
/* We're not in the interrupt anymore... */
disable_irq();
in_irq = 0;

main()
{
  /* Initialize IRQ stuff */
count = 0;
  input_full = 0;
  SetAudioInterrupt(irq);  /* Set up interrupts */
  while (1)
  {
    while( !input_full );  /* Wait for a data buffer to collect */
    /* From here until we clear input_full can only take */
* BlockLen sample times, so don’t do too much here. */

/* First, transfer inputs and outputs */

#ifdef C_FFT /* Use C FFT */

/* Insert code to fill */

/* C FFT buffers */

#else /* Use assembly FFT */

/* Insert code to fill */

/* assembly FFT buffers */

#endif /* C_FFT */

/* Done with that... ready for new data collection */
count = 0; /* Need to reset the count */
input_full = 0; /* Mark we’re ready to collect more data */

/**********************************************************/

/* Now that we’ve gotten the data moved, we can do the */
/* more lengthy processing. */

#ifdef C_FFT /* Use C FFT */

/* Multiply the input signal by the Hamming window. */
/* . . . insert C / asm code . . . */

/* Bit-reverse and compute FFT in C */
fft();

/* Now, take absolute value squared of FFT */
/* . . . insert C / asm code . . . */

/* Last, set the DC coefficient to -1 for a trigger pulse */
/* . . . insert C / asm code . . . */

/* done, wait for next time around! */

#else /* Use assembly FFT */

/* Multiply the input signal by the Hamming window. */
/* . . . insert C / asm code . . . */

/* Bit-reverse and compute FFT in assembly */
bit_rev_fft();

/* Now, take absolute value squared of FFT */
/* . . . insert C / asm code . . . */

/* Last, set the DC coefficient to -1 for a trigger pulse */

http://cnx.org/content/m11827/1.5/
/* ... insert C /asm code ... */

/* done, wait for next time around! */

#endif /* C_FFT */

}

}

4 Appendix B:

c_fft_given.asm

1 ; v:\ece420\54x\dspclib\c_fft_given.asm
2 ; dgs - 9/14/2001
3 .copy "v:\ece420\54x\dspclib\core.inc"
4 .global _bit_rev_data
5 .global _fft_data
6 .global _window
7 .global _bit_rev_fft
8 .sect ".data"
9 .align 4*N
10 _bit_rev_data .space 16*2*N ; Input to _bit_rev_fft
11 .align 4*N
12 _fft_data .space 16*2*N ; FFT output buffer
13 .sect ".text"
14 _bit_rev_fft
15 ENTER_ASM
16 call bit_rev ; Do the bit-reversal.
17 call fft ; Do the FFT

http://cnx.org/content/m11827/latest/c_fft_given.asm
32
33        LEAVE_ASM
34        RET
35
36       bit_rev:
37       STM #_bit_rev_data,AR3    ; AR3 -> original input
38       STM #_fft_data,AR7       ; AR7 -> data processing buffer
39       MVMM AR7,AR2             ; AR2 -> bit-reversed data
40       RPTBD bit_rev_end-1
41       STM #K_FFT_SIZE-1,BRC
42       MVDD *AR3+,*AR2+         ; AR0 = 1/2 size of circ buffer
43       MVDD *AR3-,*AR2+         
44       MAR  *AR3+0B
45
46      bit_rev_end:
47        NOP
48        RET
49
50 ; Copy the actual FFT subroutine.
51 fft_data .set _fft_data ; FFT code needs this.
52 .copy "v:\ece420\54x\dsplib\fft.asm"
53
54 ; If you need any more assembly subroutines, make sure you name them
55 ; _name, and include a "._global _name" directive at the top. Also,
56 ; don't forget to use ENTER_ASM at the beginning, and LEAVE_ASM
57 ; and RET at the end!

5 Appendix C:

lab4.h⁷

1 #define N 1024    /* Number of FFT points */
2 #define logN 10

lab4fft.c⁸

1 /**************************************************************************************/
2 /* lab4fft.c                                                             */
3 /* Douglas L. Jones                                                        */
4 /* University of Illinois at Urbana-Champaign                         */

⁷http://cnx.org/content/m11827/latest/lab4.h
⁸http://cnx.org/content/m11827/latest/lab4fft.c
/* January 19, 1992 */
/* Changed for use w/ short integers and lookup table for ECE420 */
/* Matt Kleffner */
/* February 10, 2004 */
/* */
/* fft: in-place radix-2 DIT DFT of a complex input */
/* */
/* Permission to copy and use this program is granted */
/* */
/* */
/* WARNING: */
/* This file is intended for educational use only, since most */
/* manufacturers provide hand-tuned libraries which typically */
/* include the fastest fft routine for their DSP/processor */
/* architectures. High-quality, open-source fft routines */
/* written in C (and included in MATLAB) can be found at */
/* http://www.fftw.org */
/* */
/* #defines expected in lab4.h */
/* */
/* N: length of FFT: must be a power of two */
/* logN: N = 2**logN */
/* */
/* 16-bit-limited input/output (must be defined elsewhere) */
/* */
/* real: integer array of length N with real part of data */
/* */
/* imag: integer array of length N with imag part of data */
/* */
/* */
/* sinetables.h must */
/* */
/* 1) #define Nt to an equal or greater power of two than N */
/* 2) contain the following integer arrays with */
/* element magnitudes bounded by M = 2**15-1: */
/* costable: M*cos(-2*pi*n/Nt), n=0,1,...,Nt/2-1 */
/* sintable: M*sin(-2*pi*n/Nt), n=0,1,...,Nt/2-1 */
/* */
/***********************************************************/
#include "lab4.h"
#include "sinetables.h"
extern int real[N];
extern int imag[N];

void fft(void)
{
    int i,j,k,n1,n2,n3;
    int c,s,a,t,Wr,Wi;
    j = 0;    /* bit-reverse */
    n2 = N >> 1;
    for (i=1; i < N - 1; i++)
    {
        n1 = n2;
        ...
while ( j >= n1 )
{
    j = j - n1;
    n1 = n1 \gg 1;
}
j = j + n1;

if ( i < j )
{
    t = real[i];
    real[i] = real[j];
    real[j] = t;
    t = imag[i];
    imag[i] = imag[j];
    imag[j] = t;
}

/* FFT */
n2 = 1; n3 = Nt;

for (i=0; i < logN; i++)
{
    n1 = n2; /* n1 = 2**i */
    n2 = n2 + n2; /* n2 = 2**(i+1) */
    n3 = n3 \gg 1; /* cos/sin arg of -6.283185307179586/n2 */
    a = 0;

    for (j=0; j < n1; j++)
    {
        c = costable[a];
        s = sintable[a];
        a = a + n3;

        for (k=j; k < N; k=k+n2)
        {
/* Code for standard 32-bit hardware, */
/* with real,imag limited to 16 bits */
/*
    Wr = (c*real[k+n1] - s*imag[k+n1]) \gg 15;
    Wi = (s*real[k+n1] + c*imag[k+n1]) \gg 15;
    real[k+n1] = (real[k] - Wr) \gg 1;
    imag[k+n1] = (imag[k] - Wi) \gg 1;
    real[k] = (real[k] + Wr) \gg 1;
    imag[k] = (imag[k] + Wi) \gg 1;
*/
/* End standard 32-bit code */
/* Code for TI TMS320C54X series */
	W = ((long int)(c*real[k+n1]) - (long int)(s*imag[k+n1])) \gg 15;
}
Wi = ((long int)(s*real[k+n1]) + (long int)(c*imag[k+n1])) >> 15;
real[k+n1] = ((long int)real[k] - (long int)Wr) >> 1;
imag[k+n1] = ((long int)imag[k] - (long int)Wi) >> 1;
real[k] = ((long int)real[k] + (long int)Wr) >> 1;
imag[k] = ((long int)imag[k] + (long int)Wi) >> 1;

/* End code for TMS320C54X series */

/* Intrinsic code for TMS320C54X series */

Wr = _ssub(_smpy(c, real[k+n1]), _smpy(s, imag[k+n1]));
Wi = _sadd(_smpy(s, real[k+n1]), _smpy(c, imag[k+n1]));
real[k+n1] = _sshl(_ssub(real[k], Wr),-1);
imag[k+n1] = _sshl(_ssub(imag[k], Wi),-1);
real[k] = _sshl(_sadd(real[k], Wr),-1);
imag[k] = _sshl(_sadd(imag[k], Wi),-1);

/* End intrinsic code for TMS320C54X series */

return;