Real-Time Digital Signal Processing using the Texas Instruments TMS320C6416DSK

Collected Notes from EE442/EE592

Phillip L. De Leon
New Mexico State University
Klipsch School of Electrical and Computer Engineering
Box 30001, Dept. 3-O
Las Cruces, New Mexico 88003-8001
(575) 646-DSP1
pdeleon@nmsu.edu

Contents

Preface

This book is a compilation of course notes written during development of a course in Real-Time Digital Signal Processing (DSP) at New Mexico State University.

Chapter 1

TI C6416DSK Pass Code

1.1 Introduction

In the installation CD that comes with the C6416DSK there appears to be four example pass code or loopback applications:

DSK_APP This example digitally processes audio data from the line input on the AIC23 codec and plays the result on the line output. It uses the McBSP and EDMA to efficiently handle the data transfer without intervention from the DSP.

SWI_AUDIO This example demonstrates how an application can use a codec mini-driver via the SIO module in SWI threads. This is a loopback application. Audio is read from an input SIO, then send back out on an output SIO. This application is configured to use the DIO adapter. This adapter library can be found in the CCS lib directory.

TSK_AUDIO This example demonstrates how an application can use a codec mini-driver via the SIO module in TSK threads. This is a loopback application. Audio is read from an input SIO, then send back out on an output SIO. This application is configured to use the DIO adapter. This adapter library can be found in the CCS lib directory.

PIP_AUDIO This example demonstrates how an application can use a codec mini-driver via the PIP module in SWI threads. This is a loopback application. Audio is read from an input PIP, then send back out on an output PIP. This application is configured to use the PIO adapter. This adapter library can be found in this local audio directory.

In this course we will use the DSK_APP.C program which is a complex and extensive passcode providing the developer access to many features on the DSK. However, as will be discussed shortly, we have modified the program to allow more flexibility in this course.

1.2 File Descriptions for DSK_APP

In this section, we provide brief descriptions of files that comprise the DSK_APP pass code as taken directly from the accompanying CD-ROM.

aic23.c Codec driver implementation specific to the Spectrum Digital DSK6416 board.

aic23.h This file is the header file for the AIC23 codec driver implementation specific to the Spectrum Digital EVM5509 board. It contains control word bit-definition macros and declaration

of public functions. The AIC23 Control registers include 10 control registers (+the reset reg.) on AIC23, each 9 bits wide. The address of a control register is 7 bits wide: together, the address of a register and its content form a 16-bit control word, the address occupying the uppermost 7 bits and the content lowermost 9.

build.bat This file builds the dsk_app project.

cc_build_Debug.log Generated by CCS. Gives build information such as errors, warnings, etc.

Debug Folder containing the downloadable object code as well as other files.

dsk_app.c Main code for dsk_app.

dsk_app.cdb Initialization for individual DSP/BIOS modules—requires the DSP/BIOS configuration tool.

dsk_app.pjt Project file for CCS which lists project settings, source files, generated files, and other settings. Do not edit.

dsk_appcfg.h At compile time, CCS will auto-generate DSP/BIOS related files based on DSP/BIOS module settings. This header file contains results of the autogeneration and must be included for proper operation. The name of the file is taken from dsk_app.cdb and adding cfg.h.

The following files are generated by the Configuration tool: dsk_appcfg_c.c, dsk_appcfg.cmd, dsk_appcfg.h62, dsk_appcfg.s62

The function of following files is unknown: build-dsk_app.tcf, Debug.lkf, dsk_app.paf, dsk_app.udc

1.3 Description for DSK_APP

The following description of DSK_APP is taken from the header of dsk_app.c. This example digitally processes audio data from the line input on the AIC23 codec and plays the result on the line output. It uses the McBSP and EDMA to efficiently handle the data transfer without intervention from the DSP.

1.3.1 Data transfer

Audio data is transferred back and forth from the codec through McBSP2, a bidirectional serial port. The EDMA is configured to take every 16-bit signed audio sample arriving on McBSP2 and store it in a buffer in memory until it can be processed. Once it has been processed, the EDMA controller sends the data back to McBSP2 for transmission.

A second serial port, McBSP1 is used to control/configure the AIC23. The codec receives serial commands through McBSP1 that set configuration parameters such as volume, sample rate and data format.

In addition to basic EDMA transfers, this example uses 2 special techniques to make audio processing more convenient and efficient:

1. Ping-pong data buffering in memory (see below)
2. Linked EDMA transfers

Applications with single buffers for receive and transmit data are very tricky and timing dependent because new data constantly overwrites the data being transmitted. Ping-pong buffering is a technique where two buffers (referred to as the PING buffer and the PONG buffer) are used for a data transfer instead of only one. The EDMA is configured to fill the PING buffer first, then the PONG buffer. While the PONG buffer is being filled, the PING buffer can be processed with the knowledge

that the current EDMA transfer won't overwrite it. This example uses ping-pong buffers on both transmit and receive ends for a total of four buffers.

The EDMA controller must be configured slightly differently for each buffer. When a buffer is filled, the EDMA controller generates an interrupt. The interrupt handler must reload the configuration for the next buffer before the next audio sample arrives. An EDMA feature called linked transfers is used to make this event less time critical. Each configuration is created in advance and the EDMA controller automatically links to the next configuration when the current configuration is finished. An interrupt is still generated, but it serves only to signal the DSP that it can process the data. The only time constraint is that all the audio data must be processed before the active buffer fills up, which is much longer than the time between audio samples. It is much easier to satisfy real-time constraints with this implementation.

1.3.2 Program flow

When the program is run, the individual DSP/BIOS modules are initialized as configured in dsk_app1.cdb with the DSP/BIOS configuration tool. The main() function is then called as the main user thread. In this example main() performs application initialization and starts the EDMA data transfers. When main exits, control passes back entirely to DSP/BIOS which services any interrupts or threads on an as-needed basis.

The edmaHwi() interrupt service routine is called when a buffer has been filled. It contains a state variable named pingOrPong that indicates whether the buffer is a PING or PONG buffer. dmaHwi switches the buffer state to the opposite buffer and calls the SWI thread processBuffer to process the audio data.

1.3.3 Other Functions

The example includes a few other functions that are executed in the background as examples of the multitasking that DSP/BIOS is capable of:

1. blinkLED() toggles LED #0 every 500ms if DIP switch #0 is depressed. It is a periodic thread with a period of 500 ticks

2. load() simulates a 20-25% dummy load if DIP switch #1 is depressed. It represents other computation that may need to be done. It is a periodic thread with a period of 10ms.

Please see the 6416 DSK help file under Software/Examples for more detailed information on this example.

DSP/BIOS is configured using the DSP/BIOS configuration tool. Settings for this example are stored in a configuration file called dsk_app.cdb. At compile time, Code Composer will auto-generate DSP/BIOS related files based on these settings. A header file called dsk_appcfg.h contains the results of the autogeneration and must be included for proper operation. The name of the file is taken from dsk_app.cdb and adding cfg.h.

1.4 Ping-Pong Buffering

Instead of sending a sample pair to a "process stereo" function and then receiving a sample pair back, the pass code instead processes an *input block* of samples and produces an *output block* of samples. While this processing is taking place, a separate input buffer receives samples from the codec and a separate output buffer transmits samples to the codec. When the input buffer is filled with new samples and the output buffer is emptied of samples, the system swaps the buffers/blocks and continues.

If the block size is set to two samples long (right and left samples), this function is identical to the familiar sample-by-sample processing. On the other hand, for block size greater than two, the

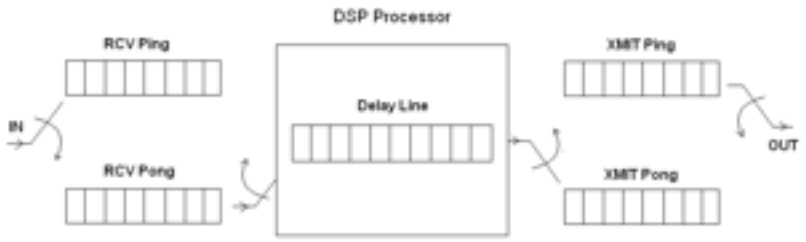

Figure 1.1: Ping pong buffering.

pass code will support applications requiring block processing such as those that require an FFT. For block sizes greater than two, the right and left samples are interleaved in the block, i.e. right, left, right, left, ...

The method of filling a block with new samples (input) and unloading a block of processed samples (output) while simultaneously processing a block of previous samples is accomplished through a *ping-pong* buffering mechanism. We require a total of four buffers: two input buffers and two output buffers as follows:

Input Buffers: receive (RCV) PING buffer and RCV PONG buffer
Output Buffers: transmit (XMIT) PING buffer, XMIT PONG buffer

1.4.1 Operation

As in the figure, the RCV PING buffer receives R/L input samples from the codec while the XMIT PING buffer transmits R/L output samples to the codec once per sample period. Simultaneously, we process prior input samples in the RCV PONG buffer and move the output samples to the XMIT PONG buffer. Once the RCV PING buffer is filled with new samples (and the XMIT PING buffer is emptied), the RCV PONG buffer will have been processed and output copied to the XMIT PONG buffer for output. We flip back and forth between the two pairs of buffers or "Ping Pong" between the two pairs. At any given time, only the RCV (input) and XMIT (output) buffers can be accessed or processed.

In the above figure, our processing amounts to feeding a delay line with the samples from the RCV PONG buffer while outputting samples from delay line to XMIT PONG buffer. Of course, it is critical that the amount of time required to process a block of samples is less than the time required to fill a new block of samples.

1.5 Sample I/O in DSK_APP

Since we assume separate right and left channel processing, the first step in passing samples is to de-interleave the stereo samples from the input block (see figure below) and store as separate right and left input blocks.

Once the input samples have been de-interleaved, we must process them. As will be seen later, we have written a process_signal() function for sample-by-sample processing (BUFFSIZE = 2, i.e. right and left samples) and a process_block() function for block processing (BUFFSIZE ¿ 2). For a pass code or loopback application, (no signal processing), we need only copy the samples (or blocks) from input to output. Finally, we must interleave right and left output samples before transmission to the codec. In the original dsk_app.c, these functions were more or less contained in a copydata()

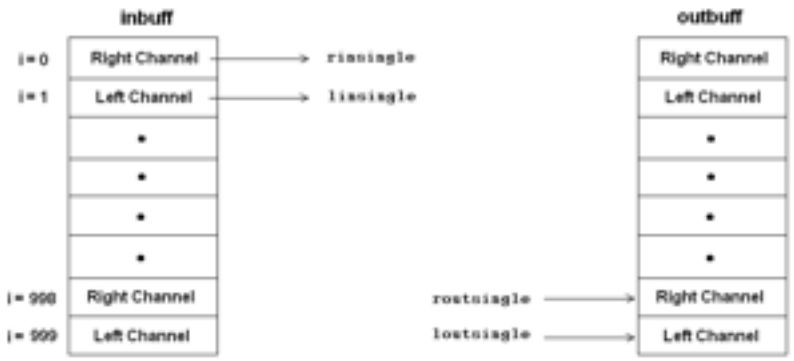

Figure 1.2: Ping pong buffering.

function. This function has been replaced by a processBuffer() function for this course. Normally, we will use the process_signal() function unless block processing is required.

1.6 Modifications to DSK_APP

We make several modifications to the DSK_APP code in order to provide a more user-friendly pass code, a smooth transition from the Freescale Modified Pass Pack to the TI DSP_APP, and provide a professional-grade pass code for further development. These modifications are made through changes in dsk_app.c as well as in the addition of four files: initialize_program.c, process_block.c, process_signal.c, user_data.h and one additional set of DSP C routines contained in the directory DSPFunctionsFixedPoint: ccan.c, cdelay.c, cfir.c, cround.c, impulse.c, lookup_wave.c, lookup_waveIntDelta.c, tap.c, tdl.c, util.h, wrap.c. We describe the changes in the order in which they appear in dsp_app.c

1.6.1 USER_DATA.H

We include a user_data.h file which partially serves to replicate the pass.dat file in the Freescale Pass Code. This file declares the initialize_program() and process_signal()/process_block() routines and all global variables. It also includes the DSP utility file util.h (described below).

```
1  #ifndef USER_DATA
2  #define USER_DATA
3
4  void initialize_program();
5  void process_signal(short inputRight, short inputLeft, short *outputRight, short *
       outputLeft);
6  void process_block(short *inputRight, short *inputLeft, short *outputRight, short *
       outputLeft);
7
8  /* BUFSIZE number of samples in buffer. For sample-by-sample processing set to 2 (R/L) */
9  /* For block processing, set to 2N for N interleaved pairs R, L, R, L, ... */
10 #define BUFFSIZE 2
11
12 /* Global variables here as extern, initializations in initialize_program.c */
13
14 #include "DSPFunctionsFixedPoint/util.h"
15
16 #endif
```

1.6.2 UTIL.H

This file declares all given *fixed-point* DSP routines needed in EE442/EE592 as well additional routines the student develops. It is assumed these files reside in the folder DSPFunctionsFixedPoint and will be described in detail in the next lecture. In addition, sampling rate definitions are defined here for use in the dsk_app.c code. These hex equivalents are found in "aic23.h" file on line 115 and are given in 9-bit format.

```
1  #ifndef UTIL_H
2  #define UTIL_H
3
4  /* Sampling rate definitions in tlv320aic23.pdf (and in aic23.h) */
5  /* See "Codec configuration settings" in dsk_app.c */
6  #define DSKAPP_AIC23_8KHZ   0x000d
7  #define DSKAPP_AIC23_16KHZ  0       /* not supported */
8  #define DSKAPP_AIC23_24KHZ  0       /* not supported */
9  #define DSKAPP_AIC23_32KHZ  0x0019
10 #define DSKAPP_AIC23_44KHZ  0x0023
11 #define DSKAPP_AIC23_48KHZ  0x0001
12 #define DSKAPP_AIC23_96KHZ  0x001d
13
14 short ccan(short M, short *a, short *b, short *w, short **p, short x);
15 void cdelay(short D, short *w, short **p);
16 short cfir(short M, short *h, short *w, short **p, short x);
17 int cround(int a);
18 short impulse();
19 short lookup_wave(short L, const short *table, int delta, short *intOffset, unsigned short
       *UfracOffset);
20 short lookup_waveIntDelta(short L, const short *table, short delta, short *offset);
21 short tap(short M, short *w, short *p, short i);
22 short tdl(short M, short *w, short **p, short N, short *gain, short *delta, short x);
23 void wrap(short M, short *w, short **p);
24
25 #endif
```

1.6.3 Sample-by-Sample Processing

For the projects developed in this course which do not require block processing, we

```
#define BUFFSIZE 2
```

(one left and one right sample) so that we process sample-by-sample as in the Freescale passcode. In this case, all processing code will reside in process_signal.c.

1.6.4 Sampling Frequency

With the sampling rate definitions defined in the util.h, the developer can use one of the sampling rate definitions in line 313 in the dsk_app.c file:

```
1 /* Codec configuration settings */
2 DSK6416_AIC23_Config config = { \
3
4 ...
5 DSKAPP_AIC23_48KHZ,   /* 8 DSK6416_AIC23_SAMPLERATE Sample rate control  */
6 ...
```

1.6.5 INITIALIZE_PROGRAM.C

Within the dsp_app.c code, we place a call to the initialize_program() routine in the appropriate place. This routine can be found in the initialize_program.c file and is to be appropriately coded by the developer. This routines serves to mimic the proginit.asm routine in the Freescale code

```
1 #include "user_data.h"
2
3 /*************************************/
4 /* Global variable initializations here */
```

```
5  /*************************************/
6
7  void initialize_program()
8  {
9
10 }
```

1.6.6 processBuffer()

The processBuffer() routine in dsk_app.c has been created to more closely resemble the Freescale main event loop which does sample-by-sample processing. In addition, the modification also gives us the option of block processing.

If BUFFSIZE = 2, we have:

```
1  if (pingPong == PING) {
2      /* Toggle LED #3 as a visual cue */
3      /* DSK6416_LED_toggle(3); */
4
5      /* Process signal sample-by-sample */
6      process_signal(gBufferRcvPing[0], gBufferRcvPing[1], &gBufferXmtPing[0], &
           gBufferXmtPing[1]);
7
8  } else {
9      /* Toggle LED #2 as a visual cue */
10     /* DSK6416_LED_toggle(2); */
11
12     /* Process signal sample-by-sample */
13     process_signal(gBufferRcvPong[0], gBufferRcvPong[1], &gBufferXmtPong[0], &
           gBufferXmtPong[1]);
14 }
```

In this case, we call the process_signal() routine passing the R/L input samples and pointers for the R/L output samples. This routine can be found in the process_signal.c file and is to be appropriately coded by the developer. This routines serves to mimic the process_stereo routine (found in the procster.asm file) in the Freescale code.

If BUFFSIZE > 2, we have:

```
1  if (pingPong == PING) {
2      /* Toggle LED #3 as a visual cue */
3      /* DSK6416_LED_toggle(3); */
4
5      /*********************************/
6      /* De-interleave into L/R blocks */
7      /*********************************/
8      for(i=0; i<BUFFSIZE/2; i++) {
9          j = 2*i;
10         inputRight[i] = gBufferRcvPing[j];
11         inputLeft[i] = gBufferRcvPing[j+1];
12         }
13
14     /* Process signal block-by-block */
15     process_block(inputRight, inputLeft, outputRight, outputLeft);
16
17     /************************/
18     /* Interleave L/R blocks */
19     /************************/
20     for(i=0; i<BUFFSIZE/2; i++) {
21         j = 2*i;
22         gBufferXmtPing[j] = outputRight[i];
23         gBufferXmtPing[j+1] = outputLeft[i];
24     }
25 } else {
26     /* Toggle LED #2 as a visual cue */
27     /* DSK6416_LED_toggle(2); */
28
29     /*********************************/
30     /* De-interleave into L/R blocks */
31     /*********************************/
32     for(i=0; i<BUFFSIZE/2; i++) {
33         j = 2*i;
34         inputRight[i] = gBufferRcvPong[j];
```

```
35          inputLeft[i] = gBufferRcvPong[j+1];
36      }
37
38      /* Process signal block-by-block */
39      process_block(inputRight, inputLeft, outputRight, outputLeft);
40
41      /************************/
42      /* Interleave L/R blocks */
43      /************************/
44      for(i=0; i<BUFFSIZE/2; i++) {
45          j = 2*i;
46          gBufferXmtPong[j] = outputRight[i];
47          gBufferXmtPong[j+1] = outputLeft[i];
48      }
49  }
```

1.6.7 PROCESS_SIGNAL.C

Within the processBuffer() routine, if BUFFSIZE = 2, we place a call to the process_signal() routine. The routine can be found in the process_signal.c file and is to be appropriately coded by the developer. The process_signal()/process_signal.c routine/file serves to mimic the process_sterero/procster.asm in the Freescale code.

```
1   #include "user_data.h"
2
3   void process_signal(short inputRight, short inputLeft, short *outputRight, short *
        outputLeft)
4   {
5       /*******************************/
6       /* Process right channel sample */
7       /*******************************/
8       *outputRight = inputRight;
9
10      /*******************************/
11      /* Process left channel sample */
12      /*******************************/
13      *outputLeft = inputLeft;
14
15  }
```

1.6.8 PROCESS_BLOCK.C

Within the processBuffer() routine, if BUFFSIZE > 2, we place a call to the process_block() routine. The routine can be found in the process_block.c file and is to be appropriately coded by the developer only if block processing is required.

```
1   #include "user_data.h"
2
3   void process_block(short *inputRightBlock, short *inputLeftBlock, short *outputRightBlock,
        short *outputLeftBlock)
4   {
5   short i;
6
7   /*******************************/
8   /* Process right channel block */
9   /*******************************/
10  for(i=0; i<BUFFSIZE/2; i++)
11      outputRightBlock[i] = inputRightBlock[i]; /* simple I/O copy */
12
13  /*******************************/
14  /* Process left channel block */
15  /*******************************/
16  for(i=0; i<BUFFSIZE/2; i++)
17      outputLeftBlock[i] = inputLeftBlock[i]; /* simple I/O copy */
18  }
```

1.6.9 Modified DSK_APP Code

A .zip file containing the DSK_APP pass code with modifications is available on the course webpage under TI 6416 Code.

Chapter 2

C Programming for Fixed-Point DSP

2.1 Introduction

The TMS320C6416 is a fixed-point signal processor therefore all arithmetic is based on short (16 bits) variables. Multiplications of two shorts leads to an int (32 bits) which is to be converted back to short.

The numerical mapping between fractional values and short is

Table 2.1: Numerical mapping

Fractional value	-1	$\rightarrow$	0	$\rightarrow$	$1-2^{-15}$
Short value	-32768 (-2^{15})	$\rightarrow$	0	$\rightarrow$	$+32767$ $(2^{15}-1)$

where we note $-32768 = -2^{15}$ and $32767 = 2^{15}-1$. Thus we need only multiply the proper fractional value by 2^{15} to get the short value. The mapping in the other direction is achieved through a division by 2^{15} (right shifting 15 times).

Note: Although it is possible to typecast sample values to float, perform all calculations in float, and typecast back to short, the overhead involved in performing floating-point calculations using a fixed-point processor can be prohibitive. Therefore,

C codes for EE442/EE592 are not to use floats for any calculations.

2.2 Examples of Non-Real Time Fixed-Point Codes

We demonstrate some implications of using shorts in calculations.

Program 1

```
1  /* Addition of two shorts */
2  short a, b, c;
3  int c2;
4
5  int main (void)
6  {
7      a = 30000;
8      b = 10000;
9
10     c = a + b;  /* c = -25536 = 40000   65536 */
11     c2 = a + b; /* c2 = 40000 */
```

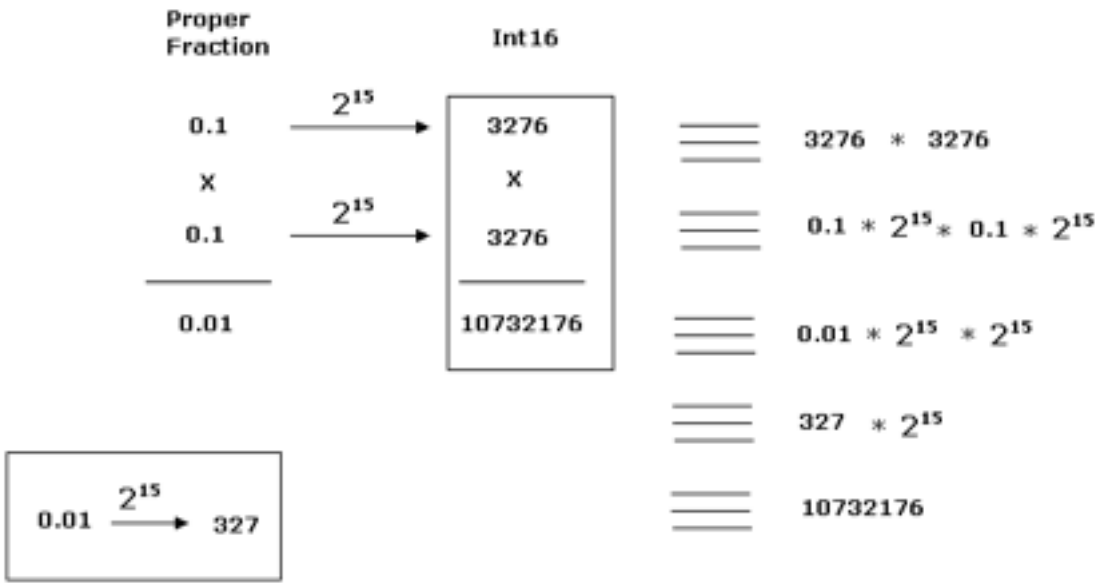

Figure 2.1: Program 2.

```
12
13      return 0;
14  }
```

Program 2

```
1   /* Multiplication of two shorts */
2   short a, b, c;
3   int c2;
4
5   int main (void)
6   {
7       a = 3276;    /* a = 0.1 */
8       b = 3276;    /* b = 0.1 */
9
10      c = a * b;   /* c = -15728 */
11
12      c2 = a * b;      /* c2 = 10732176 */
13      c2 = c2>>15;     /* c2 = 327 */
14      c = (short)c2;   /* c = 327 */
15
16      return 0;
17  }
```

Hence, we need to right shift the int value by 15 to have correct answer. Actually, before dividing by 2^{15} (right shifting by 15) we should convergently round:

Program 2b

```
1   /* Demonstration of convergent rounding */
2   int b;
3
4   int main (void)
5   {
6       b = 0x12347FFF; /* 16 LSBs < 0.5 */
7       b = cround(b);  /* round down */
8
```

```
9       b = 0x12348001; /* 16 LSBs > 0.5 */
10      b = cround(b);  /* round up */
11
12      b = 0x12348000; /* 16 LSBs = 0.5 */
13      b = cround(b);  /* convergently round down */
14
15      b = 0x12358000; /* 16 LSBs = 0.5 */
16      b = cround(b);  /* convergently round up */
17
18      return 0;
19  }
20
21  int cround(int a)
22  {
23      if (a & 0x7fff)             // If lower 16 bits != 0x8000
24          a += 0x8000;            //   normal rounding, i.e. add MSB to lower 16 bits
25      else                        // else lower 16 bits = 0x8000
26          a += (a>>1) & 0x8000;   //   convergent rounding: add LSB to upper 16 bits
27
28      return a & 0xffff0000;      // zero out lower 16 bits
29  }
```

Multiplication with convergent rounding:

Program 2c

```
1   /* Multiplication of two shorts with convergent rounding */
2   short a, b, c;
3   int c2;
4
5   int main (void)
6   {
7       a = 3276;   /* a = 0.1 */
8       b = 3276;   /* b = 0.1 */
9
10      c = a * b;  /* c = -15728 */
11
12      c2 = a * b;         /* c2 = 10732176 */
13      c2 = cround(c2);    /* convergently round */
14      c2 = c2>>15;        /* c2 = 328 */
15      c = (short)c2;      /* c = 328 */
16
17      return 0;
18  }
19
20  int cround(int a)
21  {
22      if (a & 0x7fff)             // If lower 16 bits != 0x8000
23          a += 0x8000;            //   normal rounding, i.e. add MSB to lower 16 bits
24      else                        // else lower 16 bits = 0x8000
25          a += (a>>1) & 0x8000;   //   convergent rounding: add LSB to upper 16 bits
26
27      return a & 0xffff0000;      // zero out lower 16 bits
28  }
```

Program 3

```
1   /* Division by short */
2
3   short a, c;
4   int c2;
5
6   int main (void)
7   {
8       a = 2;
9       c = 80000 / a;  /* c = -25536 = 40000   65536 */
10
11      c2 = 80000 / a; /* c2 = 40000 */
12
13      return 0;
14  }
```

Chapter 3

C Functions for Embedded DSP

3.1 Introduction

Material for this lecture and C codes are based on Chapters 4 and 7 in *Introduction to Signal Processing* by S. Orphanidis. All codes have been modified for fixed-point computation. In addition, we will show how to properly initialize and call these functions from within the DSK_APP (TI 6416DSK passcode) or the generic fixed-point stereo passcode available at

`http://www.ece.nmsu.edu/~pdeleon/EE592/TI_6416_Code.html`

3.2 Implementation of a Circular Queue

Implementation of a length N FIR filter requires storage of the most recent N samples, i.e. FIFO. Likewise implementation of a length N direct form II (canonical form) requires storage of the most recent N filter states, i.e. FIFO [note a length N IIR filter is interpreted to mean $N = \max(L, M)$ where L is the length of the feedback stage and M is the length of the feedforward stage]. Implementation of any FIFO requires a circular buffer or circular queue. For either case, we must allocate memory for the data, maintain a pointer which points to the oldest state, and wrap the pointer from top to bottom or bottom to top when necessary to ensure the pointer never exceeds the bounds of the array.

3.2.1 wrap()

The first function, wrap() (p. 172 Orphanidis) ensures that a pointer to the circular queue never goes beyond the address bounds of the array.

wrap()

```
1  void wrap(short M, short *w, short **p)
2  {
3      if (*p < w || *p > (w + M))
4          *p = w + (*p - w) % (M + 1);
5
6      if (*p - w < 0)
7          *p += M+1; // stupid C modulus operator
8  }
```

This functions operates as follows. Assume M is the order of the array, i.e. length of array is $M + 1$, *w is a pointer to the base address of the array, and *p is a pointer into the array. Since *p

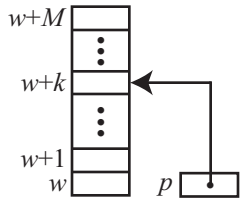

Figure 3.1: Circular queues in C.

is modified by wrap() it must be passed by reference, i.e. a pointer to a pointer **p. This routine circularly wraps pointer p relative to the array w by determining if *p has gone past the end of the array (or before the beginning of the array) and wraps it modulo M. Since the C modulo operator can't wrap for negative values we take this account into account with the second IF statement.

3.2.2 cdelay()

The second function, cdelay() (p. 177 Orphanidis) will point to the new, oldest state.

cdelay()

```
1  void cdelay(short D, short *w, short **p)
2  {
3      (*p)--;          /* decrement pointer and... */
4      wrap(D, w, p);   /* ...wrap modulo-(D+1) */
5  }
```

This functions operates as follows. Assume D is the order of the array, i.e. length of array is $D+1$, *w is a pointer to the base address of the array, and *p is a pointer into the array (p is an address). Since *p is modified by cdelay() it must be passed by reference, i.e. a pointer to a pointer **p. Since we want *p to point to the oldest sample or state, this routine simply decrements the pointer to the new, oldest state and wraps the pointer back around for circular addressing.

3.3 FIR Program

The FIR program computes the convolution (or inner product) between a vector of filter coefficients and an input vector or regressor containing the most recent N samples. The function cfir() implements the convolution assuming the input samples are stored in a circular queue.

cfir()

```
1  #include "util.h"
2
3  short cfir(short M, short *h, short *w, short **p, short x)
4  {
5      short i;
6      int y=0;
7
8      **p = x;     /* read input sample */
9
10     /* compute filter output, y = \sum_{i=0}^{M} h_k * w_k */
11     for(i=0; i<=M; i++) {
12         y += (*h++) * (*(*p)++);
13         wrap(M, w, p);
14     }
15
16     /* update filter states, i.e. wk(n+1) = w{k-1}(n) */
17     cdelay(M, w, p);     /* p -> um */
18
19     return cround(y)>>15;   /* round output and back to 16 bits */
20 }
```

This functions operates as follows. Assume M is the filter order, i.e. length of filter is $M+1$, *h is a pointer to the base address of the filter coefficients, *w is a pointer to the base address of the input queue, *p is a pointer to the oldest sample in the input queue passed by reference, i.e. **p, and x is the newest sample. The routine first replaces away the oldest sample in the queue with the newest, i.e. **p = x so that *p now points to the newest sample. Next we accumulate a sum of products between filter coefficients and the appropriate input sample. Each time through the loop we advance the pointer to the coefficients and the pointer to the input samples. Of course each time the input sample pointer is incremented we must wrap the pointer. Note that the accumulation of products is stored as int y. Next, with cdelay() we point to the new, oldest sample. Finally, we round y and divide by 2^{15} so that it contains the proper output value and return this output sample.

3.4 Real-Time FIR

(XXX See example FIR program for different version (simpler) of this program) In our passcode we must edit the files user_data.h, initialize_program.c, and process_signal.c for real-time operation. For this example, we use a simple 2-point moving average filter with filter coefficients 0.5, 0.5.

user_data.h (like pass.dat)

```
1  #ifndef USER_DATA
2  #define USER_DATA
3
4  void initialize_program();
5  void process_signal(short inputRight, short inputLeft, short *outputRight, short *
       outputLeft);
6  void process_block(short *inputRight, short *inputLeft, short *outputRight, short *
       outputLeft);
7
8  #define BUFFSIZE 2
9  #define FILTERORDER 1
10
11 /* Global variables here as extern, initializations in initialize_program.c */
12
13 extern short coeffs[];          /* FIR filter coefficients */
14 extern short states[];          /* filter states */
15 extern short *oldestStatePtr;   /* oldest state pointer */
16
17 #include "DSPFunctionsFixedPoint/util.h"
18
19 #endif
```

Note that extern is a storage class used to transmit variable information across functions. The storage is permanently assigned (global).

initialize_program.c (like proginit.asm)

```
1  #include "user_data.h"
2
3  /*************************************/
4  /* Global variable initializations here */
5  /*************************************/
6  short coeffs[FILTERORDER+1]={16384,16384};  /* FIR filter coefficients */
7  short states[FILTERORDER+1];                /* filter states */
8  short *oldestStatePtr;                      /* oldest state pointer */
9
10 void initialize_program()
11 {
12     short i;
13
14     oldestStatePtr = states;     /* setup pointer to states and clear it out */
15     for(i=0;i<=FILTERORDER;i++)
16         states[i] = 0;
17 }
```

process_signal.c (like progster.asm)

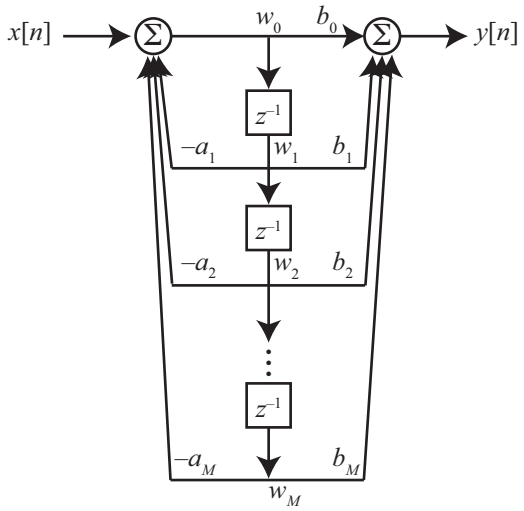

Figure 3.2: Direct form II or canonical form.

```
1  #include "user_data.h"
2
3  void process_signal(short inputRight, short inputLeft, short *outputRight, short *
       outputLeft)
4  {
5      /*******************************/
6      /* Process right channel sample */
7      /*******************************/
8      *outputRight = cfir(FILTERORDER, coeffs, states, &oldestStatePtr, inputRight);
9
10     /*******************************/
11     /* Process left channel sample */
12     /*******************************/
13     *outputLeft = inputLeft;
14 }
```

3.5 IIR Program

The standard difference equation for an Mth order IIR filter is given by

$$y(n) \quad = \quad \sum_{k=0}^{M} b_k x(n-k) - \sum_{k=1}^{M} a_k y(n-k). \tag{3.1}$$

Direct form I and II filter realizations are give below

With a direct form II realization of an IIR filter, the state equations are:

Compute new filter state, $w_0(n) = x(n) - a_1 w_1(n) - a_2 w_2(n) - \cdots - a_M w_M(n)$

Compute filter output, $y(n) = b_0 w_0(n) + b_1 w_1(n) + b_2 w_2(n) + \cdots + b_M w_M(n)$

Update filter states, $w_k(n+1) = w_{k-1}(n)$

The function ccan() implements the DFII (canonical form) assuming the filter states are stored in a circular queue.

ccan()

```
#include "util.h"

short ccan(short M, short *a, short *b, short *w, short **p, short x)
{
    short i;
    int y=0, w0;

    **p = x;            /* read input sample in (temporarily stored as **p) */

    w0 = *(*p)++ <<15;  /* begin calculation of w0 by initializing with x */
    wrap(M, w, p);      /* p -> w1 */

    /* compute new filter state, w0 = -\sum_{i=1}^{M} a_k * w_k */
    for(a++, i=1; i<=M; i++) {
        w0 -= (*a++) * (*(*p)++);
        wrap(M, w, p);
    }
    **p = cround(w0)>>15;   /* round and put w0 into state queue, p -> w0 */

    /* compute filter output, y = \sum_{i=0}^{M} b_k * w_k */
    for(i=0; i<=M; i++) {
        y += (*b++) * (*(*p)++);
        wrap(M, w, p);
    }

    /* update filter states, i.e. wk(n+1) = w{k-1}(n) */
    cdelay(M, w, p);    /* p -> wm */

    return cround(y)>>15;   /* round output and back to 16 bits */
}
```

This functions operates as follows. Assume M is the filter order, i.e. length of filter state vector is $M+1$, *a is a pointer to the base address of the feedback coefficients, *b is a pointer to the base address of the feedforward coefficients, *w is a pointer to the base address of the filter states, *p is a pointer to the oldest state in the filter states queue passed by reference, i.e. **p, and x is the newest sample.

First, we create an int variable, w0 which will store the new filter state. w0 is initialized with the input sample (multiplied by 2^{15} for proper storage as int) and we point to the first filter state (wrapping if necessary). The first for loop then computes the negative sum of products between feedback coefficients and filter states (wrapping if necessary). After the sum of products is completed, we divide by 2^{15} for storage of the new filter state as a short. Next, we accumulate a sum of products between feedforward coefficients and filter states (wrapping if necessary). Next, we back the pointer up so as to point to the new, oldest state and wrap with cdelay(). Finally, we round y and divide by 2^{15} so that it contains the proper output value and return this output sample.

3.6 Real-Time IIR

In our passcode we must edit the files user_data.h, initialize_program.c, and process_stereo.c for real-time operation. For this example, we use a simple 2nd order Butterworth halfband LPF (designed in MATLAB):

```
    [b,a] = butter(2,0.5);
    a_Coeffs = round(a * 32768) % convert to short
    b_Coeffs = round(b * 32768) % convert to short
```

user_data.h (like pass.dat)

```
#ifndef USER_DATA
#define USER_DATA

void initialize_program();
void process_signal(short inputRight, short inputLeft, short *outputRight, short *
    outputLeft);

#define FILTERORDER 2

```

```
9  extern short a_Coeffs[];          /* feedback coeffs */
10 extern short b_Coeffs[];          /* feedforward coeffs */
11 extern short states[];             /* filter states */
12 extern short *oldestStatePtr;     /* oldest state pointer */
13
14 #include "util.h"
15
16 #endif
```

initialize_program.c (like proginit.asm)

```
1  #include "user_data.h"
2
3  /*************************************/
4  /* Global variable initializations here */
5  /*************************************/
6  short a_Coeffs[FILTERORDER+1]={32767, 0, 5622};      /* feedback coeffs */
7  short b_Coeffs[FILTERORDER+1]={9598, 19195, 9598};   /* feedforward coeffs */
8  short states[FILTERORDER+1];                         /* filter states */
9  short *oldestStatePtr;                               /* oldest state pointer */
10
11 void initialize_program()
12 {
13     short i;
14
15     oldestStatePtr = states;     /* setup pointer to states and clear it out */
16     for(i=0;i<=FILTERORDER;i++)
17         states[i] = 0;
18 }
```

process_signal.c (like procster.asm)

```
1  #include "user_data.h"
2
3  void process_signal(short inputRight, short inputLeft, short *outputRight, short *
       outputLeft)
4  {
5      /*******************************/
6      /* Process right channel sample */
7      /*******************************/
8      *outputRight = ccan(FILTERORDER, a_Coeffs, b_Coeffs, states, &oldestStatePtr,
           inputRight);
9
10     /******************************/
11     /* Process left channel sample */
12     /******************************/
13     *outputLeft = inputLeft;
14 }
```

3.7 Wavetable Synthesis

The algorithms for wavetable synthesis can be coded in C for fixed-point operation. The first step in synthesizing sinusoids is to store the sinusoid values in a lookup table or array:

$$\text{waveTable}[l] = \sin(2\pi l/L),\ 0 \le l \le L-1 \tag{3.2}$$

where L is the table length. A MATLAB program, WaveTableMaker.m, has been written that will generate the table values. These values can then be pasted into the initialize_program.c file. Software codes then use this array to synthesis the sinewave. We begin with the code for wavetable synthesis using an integer-valued delta.

3.7.1 Integer-Valued Delta

Synthesis with an integer-valued delta is straightforward. We use an offset into the lookup table to get the value and increment the offset by delta each sample period. The offset is always taken modulo L (table length) so that we do not address beyond the array.

lookup_waveIntDelta()

```
lookup_waveIntDelta.c
#include "util.h"

short lookup_waveIntDelta(short L, const short *table, short delta, short *Offset)
{
    short y;

    y = table[*Offset];      /* get table entry */
    *Offset += delta;        /* accumulate delta */
    *Offset = *Offset % L;   /* Offset always mod L */
    return y;
}
```

The following lines of code in the appropriate files will then implement real-time wavetable synthesis with an integer table increment or delta.

user_data.h (like pass.dat)

```
#ifndef USER_DATA
#define USER_DATA

void initialize_program();
void process_signal(short inputRight, short inputLeft, short *outputRight, short *
    outputLeft);

#define TABLELENGTH 256

extern const short waveTable[];
extern short delta;      /* lookup table increment */
extern short offset;     /* accumulated delta */

#include "util.h"

#endif
```

initialize_program.c (like proginit.asm)

```
#include "user_data.h"

/*************************************/
/* Global variable initializations here */
/*************************************/
const short waveTable[TABLELENGTH] = {
   0, 804, 1608, 2411, 3212, 4011, 4808, 5602, 6393, 7180,
...
   -4808, -4011, -3212, -2411, -1608, -804};

short delta = 2;     /* lookup table increment */
short offset = 0;    /* accumulated delta */

void initialize_program()
{
}
```

process_signal.c (like procster.asm)

```
#include "user_data.h"

void process_signal(short inputRight, short inputLeft, short *outputRight, short *
    outputLeft)
{
    /*******************************/
    /* Process right channel sample */
    /*******************************/
    *outputRight = lookup_waveIntDelta(TABLELENGTH, waveTable, delta, &offset);

    /*******************************/
    /* Process left channel sample */
    /*******************************/
    *outputLeft = inputLeft;
}
```

3.7.2 Real-Valued Delta

The basic idea for wavetable synthesis with a real-valued delta is to use linear interpolation to estimate the value between table entries. To do this we begin by accumulating the real-valued delta as an offset into the table. The integer part of the offset will point us to the first of the two successive values (l) while the fractional part of the offset will determine the distance between the successive values (x). Since the DSP is fixed-point, the integer parts of delta and offset will be stored as the upper 16 bits in an int while the unsigned fractional part of the delta and offset will be stored as the lower 16 bits of the int. The accumulation of delta into the offset using fixed-point is then

$$\text{intOffset.UfracOffset} = \text{intOffset.UfracOffset} + \text{intDelta.UfracDelta} \tag{3.3}$$

The algorithm is summarized as follows.

CALCULATE_SLOPE Let intOffset denote the integer part of the offset and x1, x2 denote waveTable[intOffset], waveTable[intOffset+1] respectively. x1 and x2 are the two successive table values between which we must interpolate. The slope of the line across the two points is simply the "rise over the run" where the "rise" is x2 x1 and the "run" is one.

CALCULATE_INTERPOLATED_VALUE Let fracOffset denote the signed fractional part of the offset. The interpolated value, y is simply (x2 x1)*fracOffset + x1.

UPDATE_OFFSET The new offset is incremented by delta as above and intOffset is taken modulo L.

lookup_wave()

```
1  lookup_wave.c
2  #include "util.h"
3
4  short lookup_wave(short L, const short *table, int delta, short *intOffset, unsigned short
       *UfracOffset)
5  {
6      short fracOffset, x1, x2;
7      int offset, y;
8
9      x1 = table[*intOffset];  /* get consecutive table entries */
10     x2 = table[(*intOffset+1) % L];
11     fracOffset = (*UfracOffset)>>1;  /* restore sign bit */
12     y = (x2-x1)*(fracOffset) + (int)(x1<<15);  /* y = mx + b */
13
14     /* build offset = intOffset.UfracOffset */
15     offset = (((int)(*intOffset))<<16) | ((int)(*UfracOffset));
16     offset += delta;     /* increment offset by delta */
17
18     /* mask off fractional part */
19     *UfracOffset = (short)(offset & 0x0000FFFF);
20
21     /* mask off integer part */
22     *intOffset = (short)((offset & 0xFFFF0000)>>16);
23     *intOffset = *intOffset % L;  /* intOffset always mod L */
24
25     return cround(y)>>15;   /* round output and back to 16 bits */
26 }
```

The following lines of code in the appropriate files will then implement real-time wavetable synthesis with a real-valued table increment or delta.

user_data.h (like pass.dat)

```
1  #ifndef USER_DATA
2  #define USER_DATA
3
4  void initialize_program();
5  void process_signal(short inputRight, short inputLeft, short *outputRight, short *
       outputLeft);
```

```
6
7  #define TABLELENGTH 256
8
9  extern const short waveTable[];
10 extern int delta;                /* lookup table increment */
11 extern short intOffset;          /* integer part of accumulated delta */
12 extern unsigned short UfracOffset;  /* unsigned frac part of accumu delta */
13
14 #include "util.h"
15
16 #endif
```

initialize_program.c (like proginit.asm)

```
1  #include "user_data.h"
2
3  /***************************************/
4  /* Global variable initializations here */
5  /***************************************/
6  const short waveTable[TABLELENGTH] = {
7     0, 804, 1608, 2411, 3212, 4011, 4808, 5602, 6393, 7180,
8  ...
9     -4808, -4011, -3212, -2411, -1608, -804};
10
11 int delta;
12 short intOffset = 0;              /* integer part of accumulated delta */
13 unsigned short UfracOffset = 0;   /* fractional part of accumulated delta */
14
15 void initialize_program()
16 {
17 /* put integer, unsigned fractional part of delta in upper, lower 16 bits */
18     delta = (int)(2<<16) | (int)((16384 <<1));  /* delta = 2.5 */
19 }
```

process_signal.c (like procster.asm)

```
1  #include "user_data.h"
2
3  void process_signal(short inputRight, short inputLeft, short *outputRight, short *
       outputLeft)
4  {
5      /*******************************/
6      /* Process right channel sample */
7      /*******************************/
8      *outputRight = lookup_wave(TABLELENGTH, waveTable, delta, &intOffset, &UfracOffset);
9
10     /*******************************/
11     /* Process left channel sample */
12     /*******************************/
13     *outputLeft = inputLeft;
14 }
```

3.8 Other C Codes

Prof. DeLeon has written a collection of fixed- and floating-point pass codes (both mono and stereo) which call process_signal() in the same way DSK_APP does. These codes receive/send right and left input/output, from/to a file rather than a codec. These codes allow one to develop the signal processing routines on their platform/IDE of choice (for De Leon this would be Macintosh/Xcode) and paste the routines directly into a CCS project for compilation and real-time evaluation.

```
1  int main (void)
2  {
3      short inputRightSample, inputLeftSample, outputRightSample, outputLeftSample;
4      FILE *fopen(), *inputFilePtr, *outputFilePtr;
5
6      inputFilePtr = fopen("input.txt","r");       /* open input file for reading */
7      outputFilePtr = fopen("output.txt","w");     /* open output file for writing */
8
9      /*****************/
10     /* Initialization */
11     /*****************/
12     initialize_program();
13
14     /*************************************************************************/
15     /* Main loop - process right, left input samples; get right, left output samples */
16     /*************************************************************************/
17     while (fscanf(inputFilePtr, "%hd%hd", &inputRightSample, &inputLeftSample) != EOF)  {
18         process_signal(inputRightSample, inputLeftSample, &outputRightSample, &
               outputLeftSample);
19         fprintf(outputFilePtr, "%hd\n%hd\n", outputRightSample, outputLeftSample);
20     }
21
22     fclose(inputFilePtr);   /* close input file */
23     fclose(outputFilePtr);  /* close output file */
24
25     return 0;
26 }
```

If interested in developing code using this approach, see class website for more details.

Chapter 4

Real-Time Processing using the Fast Fourier Transform

4.1 The Discrete Fourier Transform

For a review of the Discrete Fourier Transform (DFT), please see Section 9.2 of *Real-Time Digital Signal Processing using the Motorola DSP5630xEVM*.

4.2 The Fast Fourier Transform

For a review of the Fast Fourier Transform (FFT), please see Section 9.3 of *Real-Time Digital Signal Processing using the Motorola DSP5630xEVM*.

4.3 FFTs on the TI TMS320C6416

4.3.1 DSPLib

TI provides a set of functions (DSPLib) which are C-callable, assembly-optimized, general-purpose signal processing routines. The reference manual describing DSPLib is named spru565b and can be downloaded from the TI or course website.

Many of the routines found in DSPLib can be used in computationally intensive real-time applications where optimal execution speed is critical. The advantages is using DSPLib are twofold. First, we can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. Second, by providing ready-to-use DSP functions, DSPLIB can significantly shorten DSP application development time.

Installation of DSPLib is as follows. Using Code Composer Studio, you add DSPLIB by choosing dsp64x.lib from the menu Project → Add Files to Project. The library dsp64x.lib can be found at

```
C:\ccstudio\c6400\include\lib\
```

Once DSPLib is installed, the linker command file (.cmd) is automatically modified with the following options in order to insure proper linking

```
-lrts6400.lib
-ldsp64x.lib
```

Finally, you must include the header file/function corresponding to the DSPLib function in your project.

TI DSPLib functions typically operate over vector operands for maximum efficiency. Even though these routines can be used to process short arrays or even scalars (unless a minimum size requirement

Table 4.1: Arguments and conventions used in DSPLib

Argument(s)	Description
$x,\ y$	Argument reflecting input data vector
r	Argument reflecting output data vector
$nx,\ ny,\ nr$	Arguments reflecting the size of vectors $x,\ y,\ r$ respectively. For functions in the case where $nx = ny = nr$, only nx is used
h	Argument reflecting filter coefficient vector (filter routines only)
nh	Argument reflecting the size of vector h
w	Argument reflecting FFT coefficient vector (FFT routines only)

is noted), greater speed-up over standard C routines when using longer arrays. Vector operands are composed of elements held in consecutive memory locations. Complex elements are stored in consecutive memory locations with the real and imaginary parts interleaved. In-place computation is generally not allowed, unless specifically noted, i.e. source memory addresses cannot be the same as the destination memory addresses.

4.3.2 Using FFT Routines from DSPLib

There are several requirements that must be met for using the FFT Routines in DSPLib. First, scaling (to prevent overflow) can be done on the twiddle factors or on the input samples. Second, the FFT routines require bit-reversal table available by executing the bitrev_index.c function from

```
C:\CCStudio\c6400\dsplib\support\fft
```

or an alternate can be written by the programmer. The bit reverse table contains 64 entries from 0-63 in bit-reversal order and an example is provided in dsk_app_fft_example.

The DSPLib FFT routine which we will use in our example is called DSP_fft16x16r which performs a 16×16 bit complex-valued FFT with rounding. The call syntax is

```
void DSP_fft16x16r(int nx, short * restrict x, const short * restrict w,
        const unsigned char * restrict brev, short * restrict y, int radix,
        int offset, int nmax)
```

where

nx , Length of FFT in complex samples. Must be power of 2 or 4 and $\leq$ 16384

x[2*nx], Pointer to complex 16-bit data input

w[2*nx], Pointer to complex FFT coefficients

brev[64], Pointer to bit reverse table containing 64 entries

y[2*nx], Pointer to complex 16-bit data output

radix , Smallest FFT butterfly used in computation used for decomposing FFT into sub-FFTs

offset , Index in complex samples of sub-FFT from start of main FFT

nmax , Size of main FFT in complex samples

4.4 Using FFT Routines in DSK_APP

In the user_data.h, if we define BUFFSIZE to anything other then 2, DSK_APP operates in a block processing mode as described in Section 1.6.8 which is required for FFT processing. In the following example, we will compute a 2048-point FFT and then invert the FFT. The result is effectively a loopback code which simply plays out (with delay) the samples received in the input block. In our passcode, we must edit the following files user_data.h, initialize_program.c, and process_stereo.c.

user_data.h

```
#ifndef USER_DATA
#define USER_DATA

#include <math.h>
#include "FFT/cplx16.h"

void initialize_program();
void process_signal(short inputRight, short inputLeft, short *outputRight, short *
    outputLeft);
void process_block(short *inputRight, short *inputLeft, short *outputRight, short *
    outputLeft);

#define BUFFSIZE 2048
#define FS 48000

#define FFTLEN  BUFFSIZE/2
#define FFTLEN_LOG2 10
#define RADIX 2
#define PI 3.14159265358979
#define DELTA (2*PI)/FFTLEN

/* Twiddle Factor Array:
Go to "Install dir" \CCStudio\c6400\dsplib\support\fft\*.c and compile the corresponding
.c file (using any C compiler) to generate the twiddle factor array according to the
selected FFTLEN (follow instructions in .exe file). Twiddle factor generating programs
from above folder are precompiled and stored in "C:\CCStudio\c6400\dsplib\bin\*.exe"
To get the actual twiddle factors, tw_fft16x16.exe can be called as follows
    "tw_fft16x16 [-s scale] N > outputfile.c"
The argument 'N' specifies the size of the FFT.  This value must be a power of 2.
The switch '-s scale' allows selecting a different scale factor for the coefficients.
The default scale factor is 32767.5. (Similarly other *.exe files and be called)
As an example, to generate the twiddle factors array for 512 point fft:
(Windows) After Compiling, run the .exe file by typing the
    following at the command prompt "tw_16x16 512 > tw_16x16x512.h"
(Linux) Type "gcc -lm tw_16x16.c" after the compilation, "./a.out 512 > tw_16x16x512.h"
*/

extern const short w[];                 // twiddle constants generated by DSPLIB:
extern Cplx16 Wi[FFTLEN/RADIX];         // array for IFFT twiddle constants
extern Cplx16 fftbuf_left[FFTLEN];      // array of complex data, for in-place FFT calc
extern Cplx16 fftbuf2_left[FFTLEN];
extern Cplx16 fftbuf_right[FFTLEN];     // array of complex data, for in-place FFT calc
extern Cplx16 fftbuf2_right[FFTLEN];
extern unsigned char brev[64];          // pointer to bit reverse table containing 64
    entries (see dsp_fft16x16r.h)

/* Global variables here as extern, initializations in initialize_program.c */
#include "DSPFunctionsFixedPoint/util.h"
#endif
```

initialize_program.c

```
#include "user_data.h"
/* Twiddle Factor Array:
Go to "Install dir"\CCStudio\c6400\dsplib\support\fft\*.c and compile the corresponding
.c file (using any C compiler) and generate the twiddle factor array according to the
selected FFTLEN (follow instructions in .exe file). Include the generated file below
*/
#if     FFTLEN == 1024      /* Size of buffer == 1024? */
#include "FFT/tw_16x16x1024.h"
#else
#if     FFTLEN == 512       /* Size of buffer == 512? */
```

```
11 #include "FFT/tw_16x16x512.h"
12 #else
13 #if     FFTLEN == 256       /* Size of buffer == 256? */
14 #include "FFT/tw_16x16x256.h"
15 #else   // Do not declare W, to get a compilation error, and force the user
16         // to used either one of the created FFT lengths or generate another
17         // file with the desired size of FFTLEN
18 #endif      /* 256  */
19 #endif      /* 512  */
20 #endif      /* 1024 */
21
22
23 /***************************************/
24 /* Global variable initializations here */
25 /***************************************/
26 // Twiddle factor array defined in "FFT/tw_16x16xXXX.h" file
27 Cplx16 fftbuf_left[FFTLEN]; // two buffers: DSPLIB FFT is not in-place
28
29 #pragma DATA_ALIGN(fftbuf_left,8) // c6000 likes things aligned on 64-bit boundaries
30 #pragma DATA_ALIGN(fftbuf2_left,8);
31
32 Cplx16 fftbuf2_left[FFTLEN];
33 Cplx16 fftbuf_right[FFTLEN]; // two buffers needed, as DSPLIB FFT is not in-place
34
35 #pragma DATA_ALIGN(fftbuf_right,8) // c6000 likes variables aligned on 64-bit boundaries
36 #pragma DATA_ALIGN(fftbuf2_right,8);
37 Cplx16 fftbuf2_right[FFTLEN];
38
39 // pointer to bit reverse table containing 64 entries (see \dsp_fft16x16r.h)
40 #pragma DATA_ALIGN(brev,8);
41 unsigned char brev[64] =
42 {
43 0, 32, 16, 48, 8, 40, 24, 56,
44 4, 36, 20, 52, 12, 44, 28, 60,
45 2, 34, 18, 50, 10, 42, 26, 58,
46 6, 38, 22, 54, 14, 46, 30, 62,
47 1, 33, 17, 49, 9, 41, 25, 57,
48 5, 37, 21, 53, 13, 45, 29, 61,
49 3, 35, 19, 51, 11, 43, 27, 59,
50 7, 39, 23, 55, 15, 47, 31, 63
51 };
52
53 void initialize_program()
54 {
55
56 }
```

process_block.c

```
1  #include "user_data.h"
2
3  void process_block(short *inputRightBlock, short *inputLeftBlock, short *outputRightBlock,
       short *outputLeftBlock)
4  {
5      int i;
6      short tmp ;
7
8      /* Copy the samples from the input buffer to the fft buffer */
9      for ( i=0; i<FFTLEN; i++ )
10     {   // >> 2 => keep in data within +-1.0 to have output within (log2(FFTLEN)/2)+1
11         fftbuf_left[i].re = inputLeftBlock[i] >> 2; // real input signal; Q15 to float
12         fftbuf_left[i].im = 0x0000 ; // real signal --> zero imaginary part
13         fftbuf_right[i].re = inputRightBlock[i] >> 2; // real input signal; Q15 to float
14         fftbuf_right[i].im = 0x0000 ; // real signal --> zero imaginary part
15     }
16
17     /***************/
18     /* Compute FFT */
19     /***************/
20     // (intrinsic function of TI's DSPLIB, see file "\dsp_fft16x16r.h also \reference\
           SPRA884A.pdf)
21     DSP_fft16x16r( FFTLEN, (short *)fftbuf_left, w, brev, (short *)fftbuf2_left, 4, 0,
           FFTLEN );
22     DSP_fft16x16r( FFTLEN, (short *)fftbuf_right, w, brev, (short *)fftbuf2_right, 4, 0,
           FFTLEN );
```

```
23
24      /*******************************/
25      /* Spectral processing here... */
26      /*******************************/
27
28      /************************/
29      /* Compute Inverse FFT */
30      /************************/
31      // x = IFFT(X) = (1/N) * conj( FFT( conj( X ) ) )   // see TI AppNote: .\reference\
            SPRA884A.pdf pg 19
32      // First, Conjugate the input of FFT()
33      for ( i=0; i<(FFTLEN); i++ )
34      {
35          tmp = fftbuf2_left[i].re;
36          fftbuf2_left[i].re = tmp ; //<< 2;//FFTLEN_LOG2;
37          tmp = -fftbuf2_left[i].im;
38          fftbuf2_left[i].im = tmp ; //<< 2;//FFTLEN_LOG2;
39
40          tmp = fftbuf2_right[i].re;
41          fftbuf2_right[i].re = tmp ; //<< 2;//FFTLEN_LOG2;
42          tmp = -fftbuf2_right[i].im;
43          fftbuf2_right[i].im = tmp ; //<< 2;//FFTLEN_LOG2;
44      }
45      // Call FFT()
46      // (intrinsic function in DSPLIB, see "\dsp_fft16x16r.h also \reference\SPRA884A.pdf)
47      DSP_fft16x16r( FFTLEN, (short *)fftbuf2_left, w, brev, (short *)fftbuf_left, 4, 0,
            FFTLEN );
48      DSP_fft16x16r( FFTLEN, (short *)fftbuf2_right, w, brev, (short *)fftbuf_right, 4, 0,
            FFTLEN );
49
50
51      /* Copy the results from the fft buffer to the output buffer */
52      for ( i=0; i<(FFTLEN); i++ )
53      {
54          outputLeftBlock[i] = fftbuf_left[i].re ; //<< 3 ;   //FFTLEN_LOG2 just real input
                signal;  Q15 to float
55          outputRightBlock[i] = fftbuf_right[i].re ; //<< 3 ; //FFTLEN_LOG2 just real input
                signal;  Q15 to float
56      }
57  }
```

The complete FFT example can be found on the course website.

Chapter 5

Other DSK Functions

5.1 Introduction

5.2 General Extension Language

General Extension Language (GEL) is an interpretive language similar to C that lets you create functions to extend CCS's capabilities. Functions are created using GEL syntax and separately loaded into CCS. With GEL you can create a Graphical User Interface (GUI) that displays on the host PC a window containing

- sliders (example shown in Fig. 5.1
- dialog boxes
- menu actions
- windows containing output text

Actions to the GUI, such as those from sliders can then be used to control the DSK. More detailed information on GEL can be found in Chapter 12 of the Code Composer Studio User's Guide (SPRU328B.PDF) and in the CCS online help (Help → Contents → Creating Code and Building Your Project → Automating Tasks with the General Extension Language).

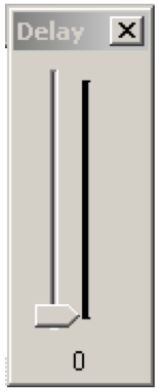

Figure 5.1: Example slider.

5.2.1 Example GEL Apllication

As a simple example on the use of GEL for creating a GUI, consider the following difference equation

$$y[n] \quad = \quad 0.5x[n] + 0.5x[n - D] \tag{5.1}$$

which implements an echo. We would like to make the echo delay, D a variable and control it (real-time) using a GUI slider similar that that shown in Fig. 5.1.

We also assume that $0 \leq D \leq \mathrm{MAX_DELAY}$. Implementation of the slider-controlled delay variable consists of modifying the following files.

user_data.h

```
<SNIP>

/* Global variables here as extern, initializations in initialize_program.c */
#define MAX_DELAY 32767          /* buffer length, max should be 32767 */
extern short buffer[];           /* buffer to store samples */
extern short *OldestSamplePtr;   /* pointer to oldest sample in buffer */
extern short delay;              /* desired echo delay */

#include "DSPFunctionsFixedPoint/util.h"

<SNIP>
```

initialize_program.c

```
short buffer[MAX_DELAY];     /* buffer to store samples */
short *OldestSamplePtr;      /* pointer to oldest sample in buffer */
short delay;                 /* echo delay */

void initialize_program()
{
    int i;

    delay = 0;               /* initial echo delay in samples*/
    OldestSamplePtr = buffer;    /* setup pointer to buffer and clear it out */
    for(i=0;i<MAX_DELAY;i++)
        buffer[i] = 0;
}
```

process_signal.c

```
void process_signal(short inputRight, short inputLeft, short *outputRight, short *
    outputLeft)
{
    short delay_sample;
    /*******************************/
    /* Process right channel sample */
    /*******************************/
    *OldestSamplePtr = inputRight;  // insert newest sample into buffer
    delay_sample = tap((MAX_DELAY-1),buffer,OldestSamplePtr,delay);
    *outputRight = inputRight/2 + delay_sample/2;

    cdelay((MAX_DELAY-1), buffer, &OldestSamplePtr);    // move pointer back

    /*******************************/
    /* Process left channel sample */
    /*******************************/
    *outputLeft = inputLeft;
}
```

echo.gel

```

menuitem "Echo";
// min, max, increment, skip, variable name
slider Delay(0, 32766, 100, 1000, d)
{
```

```
6       // delay is a global variable in process_signal.c
7       delay = d;
8   }
```

The steps to incorporate the gel file into the echo program are as follows.

1. Build the project as usual and load to target H/W
2. Load the GEL file (File → Load GEL...)
3. Display the slider (GEL → Echo → Delay) as in Figs. 5.2 and 5.3.
4. Run the program.

Notice that as you raise the slider, the delay value is increased. As the delay value is increased the echo delay becomes longer. At a 48 kHz sample rate, the maximum delay of 32766 samples yields a delay of 683 ms.

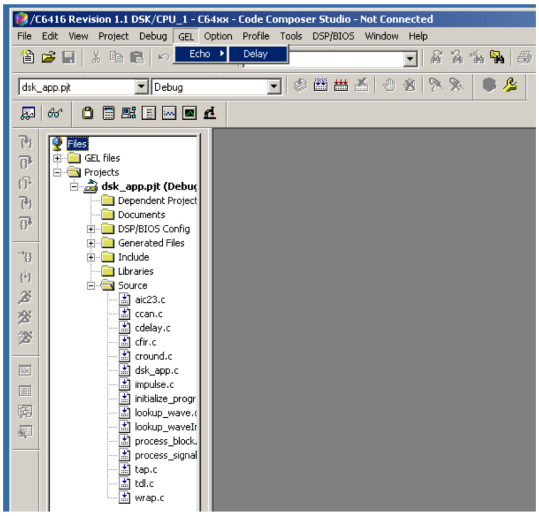

Figure 5.2: Displaying the slider (after build but before execution).

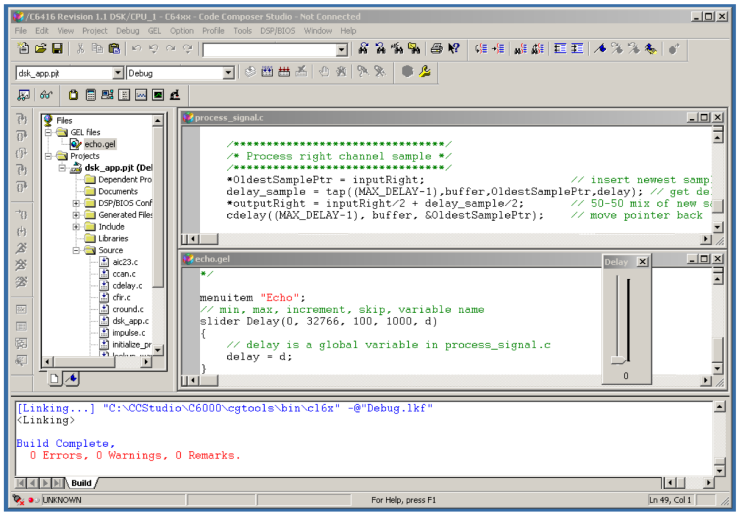

Figure 5.3: Delay slider visible in middle of window, right side.