REAL-TIME DSP LABORATORY1:
An Introduction to the TMS320C6713 DSK

January 22, 2014

Contents

Note: Starred sections contain assigned tasks to be done or assignments to be written up in the report.

1 Introduction

Digital signal processing, or DSP, is a rapidly growing industry within Electrical and Computer Engineering. With processing power doubling every 18 months (according to Moore's law), the number of applications suitable for DSP is increasing at a comparable rate. In this course, our aim is to show how mathematical algorithms for digital signal processing may be encoded for implementation on programmable hardware.

In this first lab, you will become familiar with a development system for programming DSP hardware. You will study:

- Code Composer Studio
- TMS320C6713 DSP chip and supporting chip set (DSK) architecture
- The C programming language

2 Hardware and Software

2.1 DSP Chip Manufacturers

Many companies produce DSP chips. Some of the more well known include Agere Systems, Analog Devices, Motorola, Lucent Technologies, NEC, SGS-Thompson, Conexant, and Texas Instruments [1], [2], [3]. For information on these companies, see [1] and [2]. In this course, we will use DSP chips designed and manufactured by Texas Instruments (TI). These DSP chips will be interfaced through Code Composer Studio (CCS) software developed by TI.

2.2 Code Composer Studio (CCS)

CCS is a powerful integrated development environment that provides a useful transition between a high-level (C or assembly) DSP program and an on-board machine language program. CCS consists of a set of software tools and libraries for developing DSP programs, compiling and linking them into machine code, and writing them into memory on the DSP chip and on-board external memory. It also contains diagnostic tools for analyzing and tracing algorithms as they are being implemented on-board. In this class, we will always use CCS to develop, compile, and link programs that will be downloaded from a PC to DSP hardware.

2.3 TMS320 DSP Chips

In 1983, Texas Instruments released their first generation of DSP chips, the TMS320 single-chip DSP series. The first generation chips (C1x family) could execute an instruction in a single 200-nanosecond (ns) instruction cycle. The current generation of TI DSPs includes the C2000, C5000, and C6000 series, which can run up to 8 32-bit parallel instructions in one 4.44ns instruction cycle, for an instruction rate of $1.8 \cdot 10^9$ instructions per second. The C2000, and C5000 series are fixed-point processors. The C6000 series contains both fixed-point and floating-point processors. The distinction between fixed-point and floating-point processors is important, and will be discussed in more detail later. For this lab, we will be using the C6713 processor, a member of C67x family of floating-point processors[2].

The different families in the TMS320 seriea are targetted at different applications. The C2000 and C5000 series of chips are primarily used for digital control. They consume very little power and are used in many portable devices including 3G cell phones, GPS (Global Positioning System) receivers, portable medical equipment, and digital music players. Due to their low power consumption (40mW to 160mW of active power), they are very attractive for power sensitive portable systems. The C6000 series of chips provides both fixed and floating-point processors that are used in systems that require high performance. Since these chips are not as power efficient as the C5000 series of chips (.5W to 1.4W of active power), they are generally not

used in portable devices. Instead, the C6000 series of chips is used in high quality digital audio applications, broadband infrastructure, and digital video/imaging, the latter being associated almost exclusively with the fixed-point C64x family of processors. When designing a product, the issues of power consumption, processing power, size, reliability, efficiency, etc. will be a concern.

Learning how to implement basic DSP algorithms on the C6713 will provide you with the tools to execute complex designs under various constraints in future projects. At one time, assembly language was preferred for DSP programming. Today, C is the preferred way to code algorithms, and we shall use it for fixed- and floating-point processing.

2.4 DSP Starter Kit (DSK)

The TMS320C6713 DSP chip is very powerful by itself, but for development of programs, a supporting architecture is required to store programs and data, and bring signals on and off the board. In order to use this DSP chip in a lab or development environment, a circuit board containing appropriate components, designed and manufactured by Spectrum Digital, Inc, is provided. Together, CCS, the DSP chip, and supporting hardware make up the DSP Starter Kit, or DSK.

In this lab, we will go over the basic components of the DSK and show how software may be downloaded onto the DSK. For more information, see [4].

3 Programming Languages

Assembly language was once the most commonly used programming language for DSP chips (such as TI's TMS320 series) and microprocessors (such as Motorola's 68MC11 series). Coding in assembly forces the programmer to manage CPU core registers (located on the DSP chip) and to schedule events in the CPU core. It is the most time consuming way to program, but it is the only way to fully optimize a program. Assembly language is specific to a given architecture and is primarily used to schedule time critical and memory critical parts of algorithms. In this course, we will use assembly code to gain intuition into the structure of digital filtering algorithms.

The preferred way to code algorithms is to code them in C. Coding in C requires a compiler that will convert C code to the assembly code of a given DSP instruction set. C compilers are very common, so this is not a limitation. In fact, it is an advantage, since C coded algorithms may be implemented on a variety platforms (provided there is a C compiler for a given architecture and instruction set). Most of the programs created in this course will be coded in C. In CCS, the C compiler has four optimization levels. The highest level of optimization will not achieve the same level of optimization that programmer-optimized assembly programs will, but TI has done a good job in making the optimized C compiler produce code that is comparable to programmer-optimized assembly code.

Lastly, a cross between assembly language and C exists within CCS. It is called linear assembly code. Linear assembly looks much like assembly language code, but it allows for symbolic names and does not require the programmer schedule events and manage CPU core registers on the DSP. Its advantage over C code is that it uses the DSP more efficiently, and its advantage over assembly code is that it requires less time to program with. This will be apparent in future labs when assembly and linear assembly code are given.

4 Codec

The TI AIC23 codec (coder/decoder) is a chip located on-board the DSK which interfaces the DSP chip to the analog world, specifically signal generator(s) and either an oscilloscope or stereo headphones. The codec contains a coder, or analog-to-digital converter(ADC), and a decoder or digital-to-analog converter (DAC). Both coder and decoder have two channels wihch run at sample rates which can be set from 8KHz to 96KHz and support data word lengths of 16b, 20b, 24b, and 32b at the digital interfaces. In this course, we will generally use 16 bit data word length. In Lab02, we will explore the codec in more depth.

5 C6713 DSP Chip

The C6713 DSP chip is a floating point processor which contains a CPU (Central Processing Unit), internal memory, enhanced direct memory access (EDMA) controller, and on-chip peripheral interfaces. These interface include a 32-bit external memory interface (EMIF), four Multi-channel Buffered Serial Ports (McASP and McBSP) used to communicate with the codec, two 32-bit timers, a host port interface (HPI) for high-speed communication between chips in a multi-DSP system, an interrupt selector, and a phase lock loop (PLL), along with hardware for 'Boot Configurations' and 'Power Down Logic'. See Figure 1.

6 Timing

The DSP chip must be able to establish communication links between the CPU (DSP core), the codecs, and memory. The two McBSPs, serial port 0 (SP0) and serial port 1 (SP1), are used to establish bidirectional asynchronous links between the CPU and the codec or alternately an external daughter card (not used in this course). SP0 is used to send control data between the codec and CPU; SP1 plays a similar role for digital audio data. The McBSPs use *frame synchronization* to communicate with external devices [6]. Each McBSP has seven pins. Five of them are used for timing and the other two are connected to the data receive and data transmit pins on the on-board codec or daughter card. Also included in each McBSP is a 32-bit Serial Port Control Register (SPCR). This register is updated when the on-board codec (or daughter card) is ready to send data to or receive data from the CPU. The status of the SPCR will only be a concern to us when polling methods are implemented.

In this lab, we will be exploring two possible ways of establishing a *real-time* communication link between the CPU and the codec. The idea of real-time communication is that we want a continuous stream of samples to be sent to or from the codec. For example, at an 8KHz sample rate, one sample will be sent every .125ms. This is controlled by the on-board codec, which will signal the CPU over serial port 0 (SP0), every 0.125ms.

6.1 Polling

The first method for establishing a real-time communication link between the CPU and the on-board codec is polling. When the on-board codec is ready to receive a sample from the CPU, it sets bit 17 of the SPCR register in the McBSP on the C6713 DSP to true[1]. Bit 17 of the

[1]By convention, a binary 0 will be is used for false and a binary 1 is used for true.

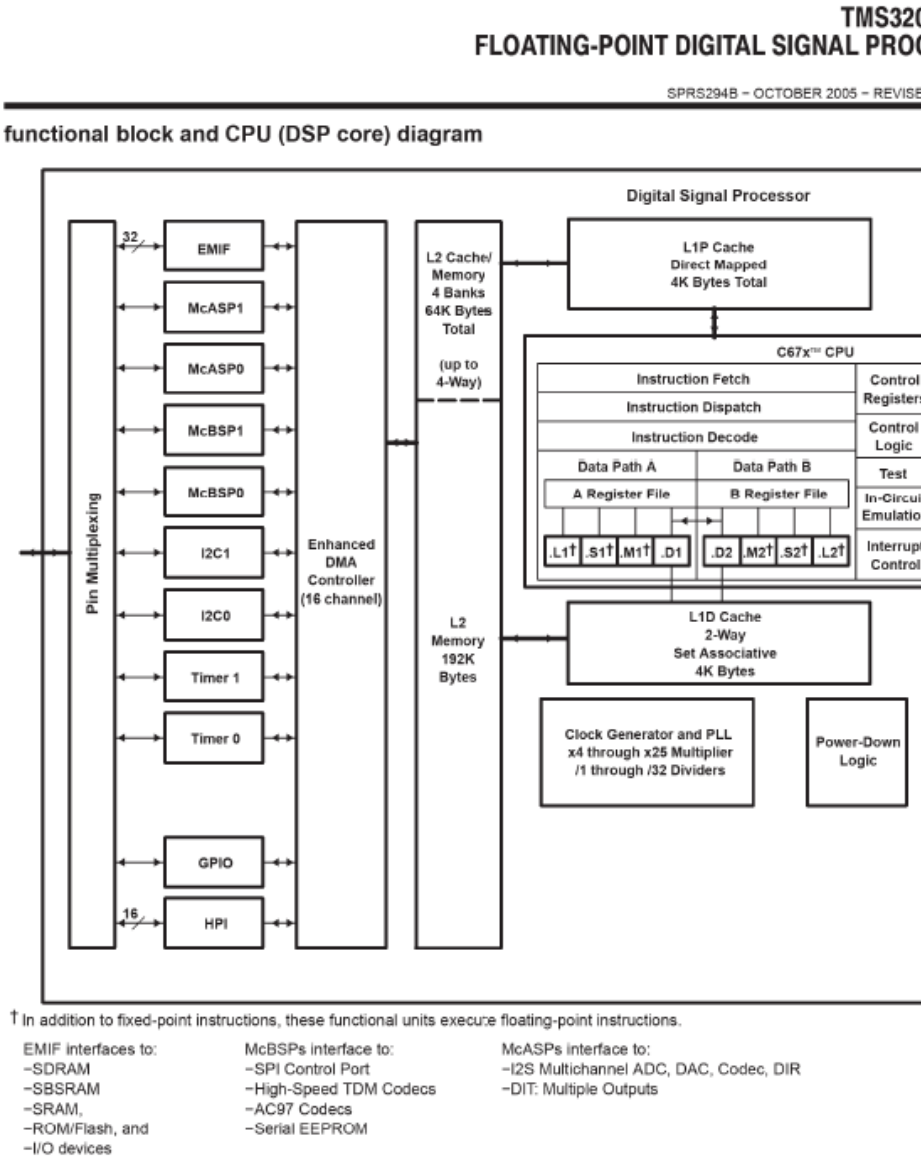

Figure 1: C6713 DSP chip layout. Adapted from [5].

SPCR is the CPU transmit ready (XRDY) bit, which the on-board codec uses to let the CPU know when it can transmit data[2]. In a polling application, the CPU continuously checks the status of the SPCR and transmits a data sample to the codec as soon as bit 17 of the SPCR is set true. Upon transmission, the McBSP will reset bit 17 of the SPCR to false. The polling

[2]The bits in the SPCR are labelled from the point-of-view of the DSP chip and not the peripheral that it is communicating with.

program will then wait until the on-board codec sets bit 17 to true to indicate that it is ready to receive the next data sample. In this manner, a polling algorithm will maintain a constant stream of data flowing to the on-board codec. Bit 1 (RRDY) of the SPCR is used in a similar manner to control the flow of sample data in the opposite direction – from the codec to the CPU.

On the DSP hardware, polling is implemented mostly in software. The on-board codec will continuously set XRDY and/or RRDY of the SPCR and the McBSP on the DSP chip will always reset it. However, it is up to the programmer to write a program that will continuously check the status of the SPCR. Fortunately, a program (support file) has already been written to manage this. The details of this program will be explained later when a polling example is implemented.

6.2 Interrupts

Interrupts are another way of handling asynchronous events on the DSP chip and may be generated either internally through software or externally by other components on the DSK. Handling (or servicing) of interrupts requires extra hardware that operates autonomously from the CPU. The C6713 chip is equipped with this hardware (timers, McBSP, etc.) and is the preferred way to time events on the DSP chip. In this lab, we will use interrupts to establish a real-time communication link between the on-board codec and the CPU via SP0. When interrupts are used to establish the real-time link between the on-board codec and the CPU, the interrupt registers in SP0 are configured to handle interrupts. Now, when the codec sets the transmit ready bit in the SPCR, the McBSP will generate an interrupt. When an interrupt occurs, the following events happen:

- the current program execution is halted;
- the current execution state is saved (in a CPU register);
- the program branches to and processes the interrupt; and
- upon completion of the interrupt processing, the execution state is restored and the program continues executing.

We will gain more insight into this process when we study assembly and linear assembly code in Lab 3.

On the C6713 DSP chip, there are thirty-two possible interrupt sources, but only twelve may be assigned by the programmer, namely INT4 through INT15 [6]. These twelve interrupts are prioritized by the 'Interrupt Selector' (see Figure 1)[3]. In this class, we will use the *SP0 transmit interrupt*, which is labelled by the interrupt acronym XINT0 in the TI literature [6]. Arbitrarily, we choose INT11 to handle this interrupt [4]. Using interrupts requires that each interrupt be mapped to an interrupt service routine (ISR). This is done by a *Vectors* file that, in our case, maps INT11 to the C coded ISR `c_int11()`. In addition, INT11 must be selected to handle interrupts from XINT0, and the DSP chip must be set up to accept programmer assigned interrupts[4]. These tasks will be done by the C coded function `comm_intr()`, which will be provided for you on the class webpage. This process will be explained again during the first programming example.

[3] In the assignable interrupts, INT4 has the highest priority and INT15 has the lowest.

[4] In polling programs, programmer assigned interrupts are disabled. This is done when the DSK is initialized.

7 Generating a Sinusoid in Real-Time

In many of the communication systems that we will design, we will want to be able to generate a sinusoid with arbitrary frequency f_o. In the first project, we generated the sinusoid

$$x(t) = \sin(2\pi f_o t), \tag{1}$$

where $f_o = 1$KHz. In real-time digital systems, this requires *samples* of the signal in eqn(1) to be sent to the codec at a fixed rate. In this example, samples will be sent to the on-board codec at a rate $f_s = 8$KHz ($t_s = 0.125$ms). In C code, we generate samples of eqn(1) ever t_s seconds. This results in

$$x[n] = x(nt_s) = \sin(2\pi f_o n t_s) = \sin(2\pi \frac{f_o}{f_s} n) = \sin(\theta_o n), \qquad \theta_o = 2\pi \frac{f_o}{f_s}, \tag{2}$$

which is only defined for integer values of n. Here, the argument of the sine function, $\theta_o n = 2\pi \frac{f_o}{f_s} n$, is a linear function that can be easily updated at each sample point. At the next time instance, time $n+1$, the argument becomes

$$\theta_o(n+1) = 2\pi \frac{f_o}{f_s}(n+1) = 2\pi \frac{f_o}{f_s} n + 2\pi \frac{f_o}{f_s} = \theta_o n + \theta_o, \tag{3}$$

which is the previous argument, namely $\theta_o n$, plus the phase offset $\theta_o = 2\pi \frac{f_o}{f_s}$. This means that the sinusoidal frequency is determined by the distance (in radians) between sample points within one 2π period of a sine wave. As long as $f_o < \frac{f_s}{2}$ ($\theta_o < \pi$), the output will have frequency f_o. Using $f_o > \frac{f_s}{2}$ ($\theta_o > \pi$) will result in aliasing, which will be explored in the first assignment. This way of generating a sine wave may seem counterintuitive, since we generally fix f_o and vary the sampling rate. Here, the sampling rate is fixed, so we control the frequency of the reconstructed signal by specifying the amount of radians to increment (within a 2π period) at each sample point. The key idea here is that the sample rate f_s is fixed and that f_o is generated by carefully choosing the sample spacing (θ_o radians).

Using the on-board codec running at 8MHz, it is theoretically possible to generate any sinusoid whose frequency is $f_o < 4$KHz in eqn(2), although limitations in the smoothing filter on the analog output reduce the maximum frequency to less than $\frac{f_s}{2}$.

8 Setting up the Equipment and Starting Code Composer Studio

8.1 Hardware Setup

For every lab in this course (with a few minor variations), the following equipment will be needed at every lab station:

- A pentium based computer with CCS version 5.5 installed on it.
- A C6713 DSK including power supply and USB cable.

- Two 3 foot cables with a 1/8th inch stereo headphone male jack on one end and two BNC male connectors (RF connectors) on the other end.
- A set of speakers or headphones (provided by student).
- A signal generator.
- An oscilloscope.

To set up the DSK, connect the USB cable between the port on the the DSK board (J201) and the USB2.0 port on the computer, then connect the 5V power supply to the power connector next to the USB port on the DSK board. You should see the four LEDs next to some dip switches blink, then finally stay all lit.

8.2 Starting CCS and Creating The First Project *

The information in this section assumes you have read the document "Getting Started with CCS" [GSCCS] and have organized your workspaces and projects as recommended. This Lab Note uses the project names and so forth as given in GSCCS - substitute the names you have chosen if necessary.

Start with one of the team members logged into the windows network.

First, connect the DSK to the power supply - wall plug first and then cable to the board. The group of LEDs should flash several time, finally stopping with all four LEDs lit. This process takes about 15 seconds and is the board performing a self-test. Note that the board is NOT ready for use until all four LEDs are lit

Then connect the board to the USB port on the computer using the supplied USB cable (important: USE the computer USB port marked 2.0, located near the top of the computer). You should hear a beep as the board driver loads.

You can (and should) run a quick test of the board and USB link by clicking on

```
C:\ti\DSK6713\drivers\6713DSKDiag.exe
```

Once the program is fully loaded, click on the `General` tab at upper left and then the `Start` button. A pop-up window titled `DSK Startup` will appear in the lower-right with the text "Waiting for USB Enumeration". This is just the diagnostic program scanning all the USB ports on the computer to find where the board is connected. The enumeration may hang because of incompatibility with USB3.0; when that happens, simply cancel the enumeration - the link has been established. The diagnostics will then start and should complete in about 30 seconds with a status of `PASS`. The window labelled `DSK` should show the following:

```
Utility Revision 1.12
  Board Version:  2
   CPLD Version:2
```

A couple of notes on this test:

- The diagnostic program will only run correctly in the Windows-XP compatability mode. If it fails to establish connection, check the compatability mode (right-click on exe, click properties, click compatibility tab).

- Running these disgnostics tests will leave the USB link in a non-functioning state such that you will not be able to download programs to the DSK. This can be easily corrected by "rebooting" the DSK:

  1. unplug the USB cable,
  2. unplug the power cable to the board,
  3. wait 5-10 seconds
  4. plug the power cable back into the board
  5. wait for the power on self-test to complet (4 LEDs are lit)
  6. plug the USB cable back into the DSK.

  The USB link should now be functional.

8.3 Creating and Running the First Project *

For the first project, create and download to the DSK a program to generate a sine wave using interrupt, using the supplied file sine_gen_intr.c.

Follow the procedures for creating a project described in "Getting Started with CCS". You will then be able to use this project as a template for other projects by copying and modifying the project as described in "Getting Started with CCS".

For this lab, we will just be observing and listening to signals generated by the on-board codec, so only the oscilloscope, headphones or speakers, and one of the 3 foot headphone-to-RF connector cables will be used. Before before connecting the DSK to the lab instruments, it is a good idea to verify that the instruments are properly set up and in working order. It is a good idea to connect the signal generator directly to the scope input you will be using, set up the signal generator for a 1 KHz, 1 V peak-to-peak sine wave and verify correct setup.

8.4 Building and Running the Project *

Now you must build and run the project. To build the first project, left-click `Project->Build All`. The program should compile with no errors.

When CCS "built" your project, it compiled the C coded source files and header files[5] into assembly code, using a built-in compiler. Then it assembled the assembly code into a COFF (common object file format) file that contains the program instructions, organized into modules. Finally, the linker organized these modules and the run-time support library (`rts6700.lib`) into memory locations to create the executable `.out` file `sine_gen_int.out`. The executable file, `sine_gen_int.out`, may be downloaded onto the DSK. When `sine_gen_int.out` is loaded onto the DSK, the assembled program instructions, global variables, and run-time support libraries are loaded to their linker-specified memory locations. This process is illustrated in [3, pg. 32].

[5]Header files are C coded files that are included into C coded source files. This is done via pre-processor directives, which will be explained later in this lab in the section Code Analysis. The purpose of header files is to store C code that may be useful to many applications and to store function prototypes. The latter is used to "clean up" a source code file and will be explained in more depth in Lab 4.

9 Code Analysis

Now that you have successfully implemented your first project in hardware, it is time to analyze the source code in `sine_gen_intr.c` to see exactly how this 1KHz sine wave was generated. Note in C (or more precisely C++) that text following '`//`' or between '`/*`' and '`*/`' is regarded as comment and is ignored when the program is compiled. A listing of `sine_gen_intr.c` is given in Figure 2.

In order to efficiently analyze the code, we will break it up into three sections, namely section one (lines 1 through 10), section two (lines 12 through 37), and section three (lines 39 through 43). Note that the line numbers referred to here are the numbers enclosed in the comment delimiters '`/*`' and '`*/`'. When viewing in CCS, there will be a separate set of line numbers to the left - ignore these.

Generally, the section containing the `main()` function, section three in this case, will always come last. In C, the function `main()` is always the starting point of the program. The linker knows to look for this function to begin execution. Therefore, a C program without a `main()` function is meaningless.

The first section of code (lines 1 through 10) is used for preprocessor directives and the definition of global variables. In C, the `#` sign signifies a preprocessor directive. In this course, we will primarily use only two preprocessor directives, namely `#include` and `#define`. Line 1 includes a header file for the codec (`_aic23` refers to the codec). This file is needed so that the preprocessor can replace the text string `DSK6713_AIC23_FREQ_8KHZ` on line 4 with the appropriate numerical value (more on this below), as well as for other reasons. In line 2, the preprocessor directive, `#include <math.h>`, tells the preprocessor to insert the code stored in the header file `math.h` into the first lines of the code `sine_gen.c` before the compiler compiles the file. Including this header file allows us to call mathematical functions such as sin(·), cos(·), tan(·), etc. as well as functions for logarithms, exponentials, and hyperbolic functions. This header file is required for the sin(·) function line 34. To see a full list of functions available with `math.h`, open the `Includes` folder in the `Project Explorer` pane, expand the first entry, scroll down to `math.h` and expand it. This will list all the defines and functions. You can also click on `math.h` and it will open in the edit pane.

The next preprocessor directive defines the fixed point number `PI`, which approximates the irrational number π. Before compiling, the preprocessor will replace every occurrence of `PI` in `sine_gen.c` with the number specified[6].

The next seven lines (4 through 10) define the global variables: `fs`, `f0`, `fs_float`, `angle`, `offset`, `amp`, and `sine_value`.

Line 4 defines an unsigned 32b integer `fs` which is a special global variable recognized by the codec support files; it specifies the codec sampling frequency. You can see the relationship between `fs` and actual frequency on lines 44-50 of file `dsk6713_aic23.h`. To open this file, open the `Include` folder in the "File View" panel and double-click on `dsk6713_aic23.h`. Notice that all sampling frequencies are multiples of 8KHz, but not all multiples are supported. To change the sampling rate to 96KHz, we would replace the right hand side of line 4 with

[6]Other commonly used preprocessor directives are `#if` and `#endif`. These directives are used to debug logical errors in code. For example, if you do not want to execute a section of code, you would put `#if 0` before the section of code and `#endif` at the end of the section. The preprocessor would then remove this section of code before compiling. By breaking the section into smaller pieces, you can home in on logical errors.

```
// This project uses support files generated by Rulph Chassaing
// Comm routines included in C6xdskinit.c

/*1 */ #include "dsk6713_aic23.h"            // needed to access codec function
/*2 */ #include <math.h>                    // needed for sin(*/ function
/*3 */ #define PI 3.14159265359             // define the constant PI
/*4 */ Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;    // sampling frequency of codec
/*5 */ float f0=1000;                        // generated sinusoid frequency
/*6 */ float fs_float;                       // needed for calculating offset
/*7 */ float angle=0;                        // sin(*/ argument in radians
/*8 */ float offset;                         // sin(*/ argument change per sample period
/*9 */ short amp=20;                         // sine amplitude scaling factor
/*10*/ short sine_value;                     // value sent to codec
/*11*/
/*12*/ interrupt void c_int11()              // interrupt service routine
/*13*/ {
/*14*/  switch(fs)                           // get sampling freq in Hz from fs
/*15*/  {
/*16*/    case   DSK6713_AIC23_FREQ_8KHZ:
/*17*/    case   DSK6713_AIC23_FREQ_16KHZ:
/*18*/    case   DSK6713_AIC23_FREQ_24KHZ:
/*19*/    case   DSK6713_AIC23_FREQ_32KHZ:
/*20*/    case   DSK6713_AIC23_FREQ_48KHZ:
/*21*/           fs_float=8000*fs;
/*22*/           break;
/*23*/   case DSK6713_AIC23_FREQ_44KHZ:
/*24*/           fs_float=44000;
/*25*/           break;
/*26*/   case DSK6713_AIC23_FREQ_96KHZ:
/*27*/           fs_float=96000;
/*28*/           break;
/*29*/   }
/*30*/  offset=2*PI*f0/fs_float;             // set offset value
/*31*/  angle = angle + offset;              // previous angle plus offset
/*32*/  if (angle > 2*PI)                    // reset angle if > 2*PI
/*33*/    angle -= 2*PI;                     // angle = angle - 2*PI
/*34*/  sine_value=(short)1000*amp*sin(angle);  // calculate current output sample
/*35*/  output_left_sample(sine_value);      // output each sine value
/*36*/  return;                              // return from interrupt
/*37*/ }
/*38*/
/*39*/ void main()
/*40*/ {
/*41*/   comm_intr();                        // init DSK, codec, SP0 for interrupts
/*42*/   while(1);                           // wait for an interrupt to occur
/*43*/ }
```

Figure 2: Listing of `sine_gen_intr.c`.

DSK6713_AIC23_FREQ_96KHZ. Notice that you can see the numerical value of a `define` such as DSK6713_AIC23_FREQ_8KHZ by hovering the cursor over the text anywhere it appears in the `sine_gen.c` file window.

The valiables `f0`, `fs_float`, `angle`, and `offset` are all `float` type which means they hold IEEE single precision (32-bit) floating point numbers. For information on this IEEE standard, see [3]. Type `float` is required by the `sine` calculation on line 34.

The variable `sine_value` is of type `short` which means it holds a 16-bit signed integer value. Since the C-library sine function returns type float, the casting operator is used to convert to type `short`.

Notice that all of the lines that contain statements end with a semicolon. This is standard in C code. The only lines that do not get semicolons are the following:

- function names, like `c_int11()`,
- conditional statements, like `if()`,
- opening and closing braces ({ }) associated with them, and
- preprocessor directives (lines starting with #)

The last section of code (lines 39 through 43) contains the function `main()`. The format of the `main()` function will NOT change substantially from program to program, and in particular the lines seen here or a slight variation will generally appear.

Line 41 calls the function `comm_intr()`, which initializes interrupt hardware on the codec and DSP chip. This function is located within the file `c6713dskinit.c`, which is one of the *support* files given to you. In order to process interrupts, an interrupt source must be designated, a specific interrupt (one of INT4 through INT15) must be assigned and enabled to handle the interrupt, and the non-maskable interrupt (NMI) and global interrupt enable bits must be set in the interrupt enable register (IER) and control status register (CSR), respectively. These registers are located on the DSP chip and are a part of the extra hardware required to handle interrupts. The initializations are done by calling low-level functions prototyped in the support files `csl_irq.h` and `csl_irqhal.h` (provided by TI), called by the function `comm_intr()`, located in the support file `c6xdskinit.c`. In this program, the interrupt source with be the on-board codec. When the codec is ready to receive a sample from the DSP chip, it will set the transmit ready bit of the SPCR in McBSP0 on the DSP chip. Assuming that the chip has been correctly configured to handle interrupts, the transmit ready interrupt on SP0, namely XINT0, will be generated.

Configuring the DSP chip for this interrupt requires four steps. First, a function call to `Config_Interrupt_Selector(11, XINT0))` is made, which will designate the interrupts on XINT0 will be made and that these interrupts are assigned to INT11. This initialization will configure the higher and lower interrupt multiplexing registers (IMH and IML respectively). In the case of INT11, the initialization will alter the higher (upper 16-bits of the) interrupt multiplexing register, which is determined by the hardware. Second, the function call `enableSpecificINT(11)` is made, which will enable INT11. This is required for INT11 to be recognized as an active interrupt handler. Third, a function call to `enableNMI()` is made to enable non-maskable interrupts. This is required since maskable interrupts (INT4 through

INT15) will not be recognized unless the NMI bit in the IER is enabled. NB: The NMI is cleared (disabled) whenever the DSK is reset. Finally, a function call to enableGlobalINT() is made, which will enable the GIE bit of the CSR. All of these configurations are required for maskable interrupts to be processed. To learn more about configuring the DSP chip for handling interrupts, examine the code in `c6713dskinit.c` and refer to either [4], [3], [6], or [7]. Now, the DSP chip and codec have been configured to communicate via interrupts, which the codec will generate every .125ms. The program coded in `sine_gen.c` now waits for an interrupt from the codec, so an infinite loop (line 42) keeps the processor idle until an interrupt occurs. This does not have to be the case, since an interrupt will halt the CPU regardless whether it is processing or idling. But in this program, there is no other processing, so we must keep the processor idling while waiting for an interrupt to occur.

The middle section of code (lines 12 through 37) is used to define the interrupt service routine or ISR. When an interrupt occurs, the program branches to the ISR `c_int11()` as specified by `Vectors_intr.asm`. This interrupt generates the current sample of the sinusoid and outputs it to the codec.

For calculation of the argument of the sine function (line 34), we need the actual sampling rate in `Hz`, which is contained by variable `fs_float` . The conversion is in the `switch()` construct on lines 14-29 and used in the offset calculation on line 30. Line 30 determines the offset value $2\pi\frac{f_o}{f_s}$. For a given f_o, this value will not change, so it does not need to be calculated every time an interrupt occurs. By the same token, `fs_float` need not be calculated every interrupt. However, by calculating these values here, we will be able to change the value of our sinusoid frequency `f0`, as well as the sampling frequency `fs` and derived `fs_float` using a Watch Window. This is demonstrated in the next section. Line 31 calculates the current argument to the `sin()` function. The `sin(x)` function in C approximates the value of $\sin(x)$ for any value of x, but a better and more efficient approximation will be computed if $0 \leq x < 2\pi$. Therefore, lines 32 and 33 are used to reset the value of the argument if it is greater than 2π. Since $\sin(x)$ is periodic 2π in x, subtracting 2π from x will not change the value of the output.

Line 34 calculates the current sine value sample point. The right-hand-side is *typecast* as `(short)` before it is stored in the variable `sine_value`. Typecasting tells the compiler to convert a value from one data type to another before storing it in a variable or sending it to a function. In this case, the value returned from the `sin()` is a single precision floating point number (between -1.0 and 1.0) that gets scaled by `20000`, since `amp` was initialized to 20 . By typecasting this number as a `short` (16-bit signed integer between the values -32768 and 32767), the CPU will round the number to the nearest integer and store it in a 16-bit signed integer format (2's complement). This value is scaled by `20000` for two reasons. First, it is needed so that rounding errors are minimized, and second, it amplifies the signal so it can be observed on the oscilloscope and heard through speakers or headphones. This scaling factor must be less than 32768 to prevent overdriving the codec, which we will explore later in this lab. Line 35 sends the current sine value to the *left channel* of the codec by calling the function `output_left_sample()`. Notice again that by hovering the cursor over `output_left_sample()` on line 35, you can see function prototypes which show the way the function can be called.

The code for `output_left_sample()` is located in file `c6713dskinit.c`. Open the file `c6713dskinit.c` in CCS and examine the code for this function. This function forces the least significant digit of the sample that it receives to zero and sends it to a function `MCBSP_write()` which writes the sample to the transmit buffer in the McBSP. This will cause the McBSP to transmit the data sample to the on-board codec. The masking of the least significant digit of the output sample is

needed so that the on-board codec interprets the received binary number as a data sample and not as secondary information[7]. Upon completion of the interrupt (generating a sinusoid sample and outputting it to the on-board codec), the interrupt service routine restores the saved execution state (see the command `return;` in line 36). In this program, the saved execution state will always be the infinite while loop in the `main()` function.

10 Debugging

In this section we will use the IDE to i) find and fix source syntax errors, ii) single-step through the program to trace execution, and iii) observe and change the variables of a running program. We will learn additional debugging techniques later in the course.

10.1 Modifying Code to Fix Syntax Errors *

Create a deliberate syntax error by deleting the comma at the end of line 4 of `sine_gen_intr.c` Recompile the project by selecting `Project->Rebuild All`. Note the error message produced in the `Console` window and the fact that one syntax error can produce multiple lines of messages:

```
../sine_gen_intr.c
../sine_gen_intr.c", line 8: error #66: expected a ";"

../sine_gen_intr.c", line 33: error #20: identifier "f0" is undefined
>> Compilation failure
2 errors detected in the compilation of "../sine_gen_intr.c".
gmake: *** [sine_gen_intr.obj] Error 1
```

Experiment by producing other errors and noting the compile error messages. Be sure to fix all errors before continuing.

10.2 Tracing Program Execution *

Tracing program execution is a useful way to determine why a program is behaving unexpectedly. CCS gives you this capability.

Load `sine_gen_intr.out` by clicking on the bug icon or `Run->Debug` while in the "Edit Perspective". When CCS switches to the "CCS Debug Perspective", do the following: in the `sine_gen_intr.c` window, set a breakpoint by double-clicking in the gray vertical column to the left of the text on line 14. A gray dot should appear signifying a breakpoint; if the breakpoint window is open, you should see an entry for the breakpoint. Stretch the "Identity" column until you see all the text, including the line number and state. Enable the breakpoint if necessary by checking the box on the left in the breakpoint window.

[7] The on-board codec is programmed to have certain characteristics in its data and voice channels. This is done through secondary communication where the least significant digit of the binary number sent to the on-board codec is one. To learn more about programming the on-board codec, see [8].

Now click on the "Resume" icon (green triangle); the program should run briefly then stop at the breakpoint with the execution point indicated by an arrow in the left column. You can then single-step through the code, stepping into any functions called, by clicking `Run->Step Into` or pressing the F5 key; you can step only through the current file, skipping over any functions called by clicking `Run->Step Over` or pressing the FF6 key respectively. While the program is halted you can quickly check the current value of any variable by hovering the cursor (click on a blank space in the code window to bring focus if hovering is not working). For example, at this point the value of `fs` should be defined and equal to 1 assuming the sampling rate is 8 KHz (line 4). However, the value of variables below the cursor, such as `offset` should have undefined random values since this is the first pass through the function. Click "Resume" one more time and observe the value of `offset` again.

10.3 Using the "Expressions" Window to Monitor Variable Values *

Once an algorithm has been coded, it is beneficial to have software tools for observing and modifying the local and global variables after a program has been loaded onto the DSK.

CCS has a number of ways view local variables and to view and modify global variables during execution. In this lab, we will not view any local variables, but we will view and modify global variables.

With the program halted in the "Debug Perspective", bring the `Expressions` window into focus. Normally it is seen at the upper right of the screen visible as a tab. If necessary, click the tab to bring it to the top and in focus. In the expression column, add the following variable names: `sine_value, angle, offset, and f0`. With the breakpoint still active at at line 14, click on the "Resume" icon several times. The program will execute one sample update and you should see that behavior in all the variables except `offset` and `f0`.

Now notice the `f0` frequency under 'Value', you should see the number `1000`, which is the frequency of the observed sinusoid. Click on the value `1000` and change it to `2000`. When you resume the program, you should see `offset` change to twice the previous value and `angle` increase at twice the rate as before.

Now disable the breakpoint at line 14 by right-clicking on it in the source code window and selecting disable or unchecking it in the breakpoint window. With the DSK connected to the oscilloscope and optionally headphones), click on the "Resume" icon and you should observe a 2KHz sine wave, assuming `f0` is still `2000`. Halt the running program by clicking on the "Suspend" icon (double bars just to the right of the "Resume" icon) and change `f0` back to `1000`. You should see that change in the scope display and headphone tone.

Change the sampling frequency by adding `fs` to the watch window and changing it to the value for 96KHz (this will be a small integer and NOT `96000`. Find the appropriate integer value for `fs` for 96KHz by looking in `dsk6713_aic23.h` as described above. Notice the difference in the appearance of the sine wave for the higher sampling rate vs 8KHz. You should also see a different behavior in `offset` because of the finer sampling.

Be sure to undo your changes by editing or re-reading the file `sine_gen_intr.c` stored on disk by clicking `File->Open` and selecting `sine_gen_intr.c`.

11 Producing Artifacts by "Improper" Operation

Here we will look at two of the ways the DSK can produce unexpected results by operating outside the parameters of the design.

11.1 Overdriving the Codec *

In `sine_gen_intr.c`, the variable `sine_value` goes from -20000 to 20000 because of how variable `amp` is initialized. The codec is setup to receive 16-bit signed integers, which are numbers in the range $-2^{15} = -32768$ to $2^{15} - 1 = 32767$. Any values that are greater than 32767 or less than -32768 will result in a 2's compliment overflow. This means that values of `amp` less than or equal to `32` will not overflow the codec and values greater than or equal to `33` will overflow the codec. Keeping in mind that `amp` must be integer valued, use the `Expressions` window to experiment with various values of `amp`, both in the underflow (less than or equal to `32`) and overflow (greater than `32`) ranges. Also, try negative values for `amp`. Listen to the results. Can humans detect polarity or phase offsets[8] when listening to musical tones?

11.2 Aliasing Effects *

In the program `sin_gen_intr.c`, a sinusoid is generated by evaluating samples of the function $\sin(2\pi f_0 t)$ every $t_s = 0.125$ms. When sinusoids with $f_0 > 4$KHz are sampled at $f_s = 1/t_s = 8$KHz, they are undersampled. This results in an aliasing effect. To see this, in the `Expression` window change the value of `f0` in `sin_gen_intr.c` to both `5000` and `7000`.

Assignment

1. What frequencies are observed when $f_0 = 5$KHz and $f_0 = 7$KHz? Using your knowledge of signals and systems, justify the frequencies observed using both a using frequency-based and a time-based method. In the frequency-based method, use the Shannon-Whitaker sampling theorem. In the time-based method, examine the sample points of the function $\sin(2\pi f_0 t)$ for an $f_0 > 4$KHz and its aliased frequency observed on the oscilloscope. That is, sketch the sine waves actually output when you sample 5KHz and 7KHz sine functions every 0.125ms.

12 Generating a Sine Wave Using Polling*

This section has three purposes: to demonstrate how to reuse a previously created project, create a real-time communication link between the CPU and codec using polling, and generate a sinusoid using a lookup table. To create the project `sine_lookup_poll`, follow these instructions:

1. Open project `sine_gen_intr` (if not already open) and click `Project->Clean`. MAKE SURE the box labelled "Start a build immediately" is unchecked. This will delete the folder labelled "Binaries".

[8] Flipping the polarity of sine wave is the equivalent of introducing a 180 degree phase shift into the original sine wave. In other words, changing the polarity is a specific type of phase offset (a more general concept).

2. Delete the folder labelled "Debug". It will be regenerated if you later need to recompile this project.

3. Now right-click on the project folder `sine_gen_intr` and do a copy-paste operation as would be done with the windows file explorer. Choose a descriptive name such as `sine_gen_poll`.

4. Close first project using the context menu, i.e. right-click and click `Close Project`.

   At this point, we have copied over all the necessary source files and setting to make a new compilable project. Now we can modify the top-level source file.

5. Using the context menu in the `Project Explorer`, rename the file `sine_gen_intr.c` to `sine_lookup_poll.c`.

6. Since we will be using polling instead of interrupts, enable the file `Vectors_poll.asm` by right-clicking the file and unchecking `Exclude from Build`. Similarly, right-click and check `Exclude from Build` for file `Vectors_int.asm`

7. In CCS, double-click on `sine_lookup_poll.c` in the left hand window. In the edit window, delete code from the function `cint_11()` to the end of the file. Add the following code to the area vacated:

```
short sine_table[8] = {0,14142,20000,14142,0,-14142,-20000,-14142};
short ctr;
void main()
{
  ctr=0;
  comm_poll();
  while(1)
    {
      output_left_sample(sine_table[ctr]);
      if (ctr < 7) ++ctr;
      else ctr = 0;
    }
}
```

   You can delete any unused variables associated with `c_int11()`) but you must retain the line defining `fs` in order to specify the sampling rate.

8. Add comments to your code where appropriate and save the file in CCS.

Now, build your project by clicking on the "build" button and load it onto the DSK and observe a 1KHz sine wave on an oscilloscope.

Notice that the sine wave algorithm is now coded within the infinite while loop (`while(1)`). This is the general structure for polling. In both polling and interrupt based programs, the algorithm must be small enough to execute within 0.125ms (at an 8KHz rate) in order to maintain a constant output to the on-board codec. Algorithms can be coded under either scheme, using polling or interrupts. In this class, most of the algorithms will be coded using interrupts.

Assignment

2. Study the code above and the code in `c6713dskinit.c`. Pay particular attention to the functions `output_left_sample()` and `MCBSP_write()`. Where is the polling done? Other than the fact that polling is used instead of interrupts, how is this algorithm different from the algorithm in `sine_gen_intr.c`?

3. Re-implement the project `sine_lookup_poll` using interrupts by copying to a new project named `sine_lookup_intr`. Rename the source C file to `sine_lookup_intr.c` and modify it. Explain the procedure you used to change a polling based program to an interrupt driven program. Include a copy of your C program.

13 Design Problem *

In `sine_gen_intr.c`, a discrete-time output sample is calculated at every interrupt. In this code, there are an integer number of sample points $N_s = f_s/f_0$ per cycle of the sine wave, where f_s is the sample frequency and f_0 is the sine frequency. The same set of sine values repeat for each cycle: the output samples form a **discrete-time periodic sequence**. Note that for sine frequencies which are not an integer divisor of the sample frequency, such as $\frac{1}{4000\pi}$Hz (or 2000 rad/sec), the sequence is aperiodic. There is no limitation in the code to prevent this and the program will still work properly.

However, when the sequence of discrete-time samples IS periodic, a lookup table may be a better option than repeatedly re-calculating the same output sample values. This is true whether using interrupts or polling to send values to the codec (see `sine_lookup_poll.c`). We will see another example of the table advantage in Lab 6, which employs pre-computing and storing the 128 twiddle factors of a 256-point FFT algorithm.

For generating sinusoids of various frequencies, a large sine table (e.g. 1000 points or more) may be employed. The frequency of the sinusoid can be changed by incrementing the counter variable `ctr` by any integer smaller than the length of the table at each interval. In the previous code, the command `++ctr;` incremented the counter by one, which in C is equivalent to coding either `ctr += 1;` or `ctr = ctr + 1;`.

In MATLAB, the values of `sine_table[8]` were generated by the command

```
20000*sin(2*pi*[0:7]/8).
```

Since the number of discrete-time samples was small, they were included directly into the C source code. For larger sine tables, it is recommended that you store the values in a header file (extension `.h`) and include the file in the beginning part of your program. Generating this header file can be done directly using the homebrew MATLAB function `make_sine_table.m`.

To see this, create a new project `sine_lookup_alt` in the workspace for `Lab_01` by copying the project `sine_lookup_intr` (see Question 3) and renaming it. Rename the source C file to match the project name.

Download the MATLAB file `make_sine_table.m` from the class webpage into the `sine_lookup_alt` folder. Open MATLAB and change your current working (MATLAB) directory to the directory where you downloaded `make_sine_table.m`. At the MATLAB command prompt, type

```
>> make_sine_table('sine_table8',8).
```

This will create the file `sine_table8.h`, which is the header file that you will want to include in your C code. Note: It is important that this header file be in the same folder as the C source code file for this example, namely `sine_lookup_alt.c`. In CCS, open the project `sine_lookup_alt` (if necessary) and update `sine_lookup_alt.c` by replacing the line[9]

```
short sine_table[8] = {0,14142,20000,14142,0,-14142,-20000,-14142};
```

with

```
#include "sine_table8.h"
```

Build this project and run it on the DSK. You should see the same 1KHz sine wave from before.

In CCS, open up the file `sine_table8.h`, which is located in the project explorer pane. Notice that this file contains the code that was originally in the C source code file.

The number of evenly spaced discrete-time samples of one period of a sinusoid (along with the rate of the codec) determine the frequency of the observed (analog) sine wave. If `N` is the number of samples and f_s is the sample rate of the codec, then the observed (analog) sine wave will have frequency $f_s/\texttt{N}$ Hz. In this example, $\texttt{N} = 8$ and $f_s = 8000$Hz, so the observed (analog) sinusoid frequency will be (8000Hz)/8 = 1KHz. The name of the header file included in the C source code is `sine_table8.h`, which was determined by passing the string `'sine_table8'` to the function `make_sine_table.m`. This file will prove to be useful in the next assignment. As a final note, the counter variable, `ctr`, needs to be reset to zero only after it increments past `N-1`. In the previous program, this value was `7`, since there were `8` samples of a sinusoid.

Assignment

4. Create a project, using either polling or interrupts, that generates a sinusoid using a lookup table of 1000 points. Use the program `make_sine_table.m` (mentioned above) to create a header file with 1000 samples of one period of a sinusoid. Define a new global variable `step` of type `short` and initially assign it the value `20`. Increment your counter variable by the value of `step` instead of `1` in each iteration. Now, when the value of `step` is not a factor of the size of the lookup table (1000 in this case), the counter variable will not necessarily be reset to zero. To compensate for this, use the modulus (or remainder) operator `%`. (HINT: Use the help menu in CCS to research the modulus operator and refer to "Class Demo 1".) Describe the design process and submit a copy of your C source code. Test your program by observing a 160Hz sine wave on an oscilloscope for `step=20`.

5. Once your program is working, derive an analytical formula for the sinusoidal frequency f_o in terms of the variable `step`. Then, determine analytically the actual frequency observed for the following values of `step`: `1`, `4`, `59`, `200`, and `500`. Once you have calculated these values, use an oscilloscope to measure the frequency observed for each value of `step`. If you have coded this correctly, each of these sine waves should appear to be relatively smooth. Use a table to compare the analytical and measured values of the actual frequencies f_o.

[9] When `#include` is used to include files that are not pre-compiled header files (like `math.h`), quotation marks are used in place of the `<` and `>` signs.

14 End Notes

The first lab was used to learn how to create a project and implement it on the DSK. In all real-time DSP algorithm implementations, the processing rate of a digital signal processing system is very important. For this lab, only an 8KHz rate was used to implement algorithms. In future labs, this rate will be increased by the use of an audio daughter card, but the issues of timing will be the same. In the next lab, we will explore the concepts of input and output, and develop the foundation for designing systems and processing signals on the C6713 DSK. For more introductory information about the C6713 see [4].

One of the requirements for this class is that you, the student, design a lab of your own. In future labs, the end notes will be used to give you ideas for creating your own lab.

References

[1] "Dsp chips - internet resources." World Wide Web. http://www.eg3.com.

[2] "Texas instruments homepage." World Wide Web. http://www.ti.com.

[3] S. A. Tretter, *Communication Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C6701 and TMS320C6711*. Kluwer Academic/Plenum Publishers, New York, 2003.

[4] R. Chassaing, *Digital Signal Processing and Applications with the C6713 and C6416 DSK*. Wiley, New Jersey, 2005.

[5] Texas Instruments, Dallas, TX, *TMS320C6713B Floating Point Digital Signal Processor, SPRS294B*, 2006.

[6] Texas Instruments, Dallas, TX, *TMS320C6000 Peripherals Reference Guide, SPRU190d*, 2001.

[7] Texas Instruments, Dallas, TX, *TMS320C6000 CPU and Instruction Set Reference, SPRU189G*, 2006.

[8] Texas Instruments, Dallas, TX, *TLV320AIC23 Stereo Audio CODEC, SLWS106D*, 2002.