Computer Laboratory Manual

DSP-Starter Kit TMS320C6416T DSK

(CE-405)

Computer Engineering Department

Sir Syed University of Engineering & Technology
University Road, Karachi 75300
http://www.ssuet.edu.pk

LAB # 1

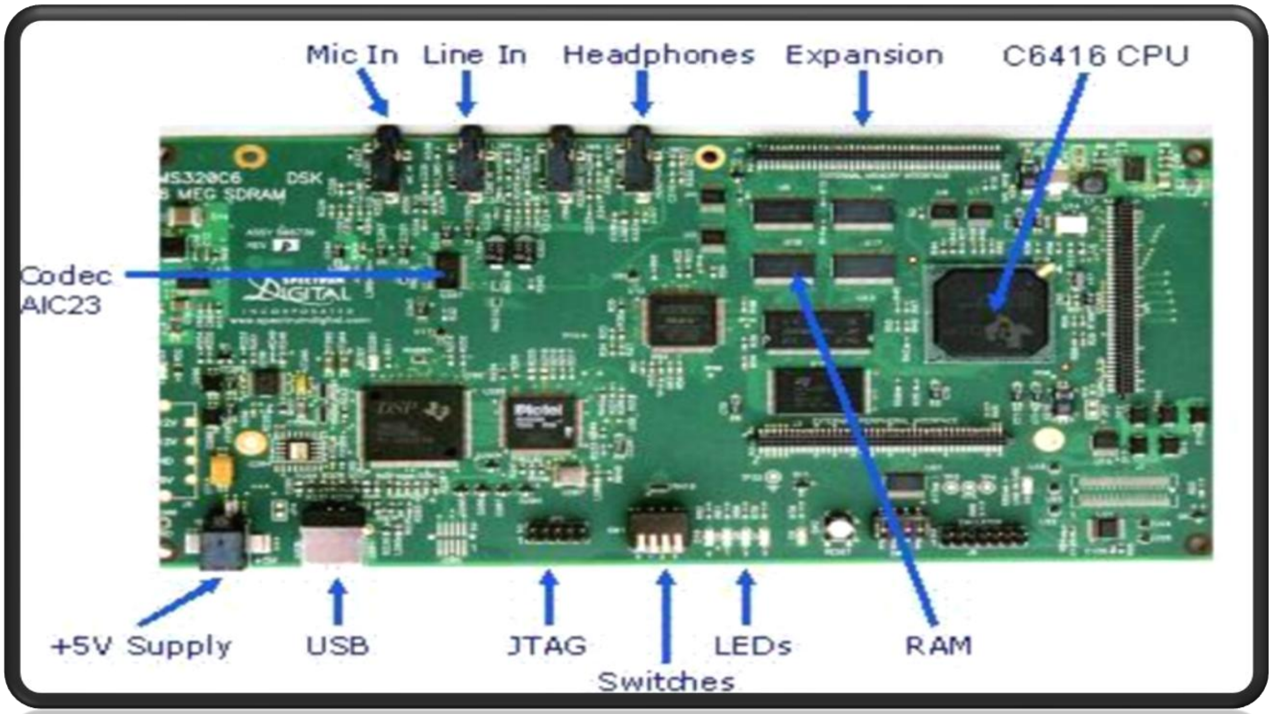
OBJECTIVE:

To Study about the description of the TMS320C6416T DSK along with the key features and a block diagram of the circuit board.

DESCRIPTION

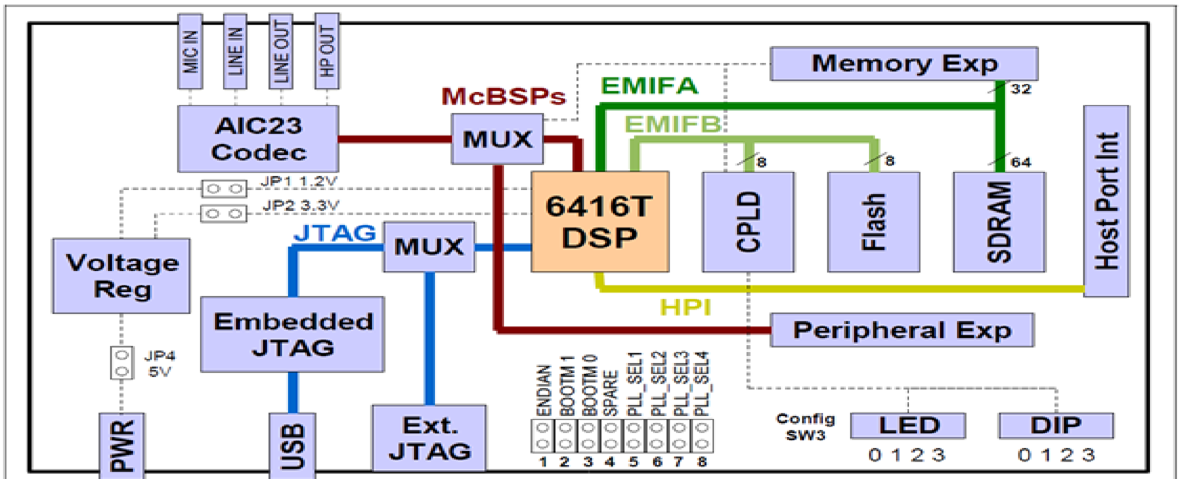
The DSP on the 6416T DSK interfaces to on-board peripherals through one of two busses, the 64-bit wide EMIFA and the 8-bit wide EMIFB. The SDRAM, Flash and CPLD are each connected to one of the busses. EMIFA is also connected to the daughter card expansion connectors which is used for third party add-in boards. An on-board AIC23 codec allows the DSP to transmit and receive analog signals. McBSP1 is used for the codec control interface and McBSP2 is used for data. Analog I/O is done through four 3.5mm audio jacks that correspond to microphone input, line input, line output and headphone output. The codec can select the microphone or the line input as the active input. The analog output is driven to both the line out (fixed gain) and headphone (adjustable gain) connectors. McBSP1 and McBSP2 can be re-routed to the expansion connectors in software. A programmable logic device called a CPLD is used to implement glue logic that ties the board components together. The CPLD also has a register based user interface that lets the user configure the board by reading and writing to the CPLD registers. The DSK includes 4 LEDs and 4 position DIP switch as a simple way to provide the user with interactive feedback. Both are accessed by reading and writing to the CPLD registers. An included 5V external power supply is used to power the board. On-board switching voltage regulators provide the 1.2V DSP core voltage and 3.3V I/O supplies. The board is held in reset until these supplies are within operating specifications. A separate regulator powers the 3.3V lines on the expansion interface. Code Composer communicates with the DSK through an embedded JTAG emulator with a USB host interface. The DSK can also be used with an external emulator through the external JTAG connector.

Figure below shows an overview of the Digital DSK board .



1.1 Block Diagram

The C6416T DSK is a low-cost standalone development platform that enables users to evaluate and develop applications for the TI C64xx DSP family. The DSK also serves as a hardware reference design for the TMS320C6416T DSP. Schematics, logic equations and application notes are available to ease hardware development and reduce time to market.



Block Diagram C6416T DSK

The DSK comes with a full compliment of on-board devices that suit a wide variety of application environments.

1.2 Key Features

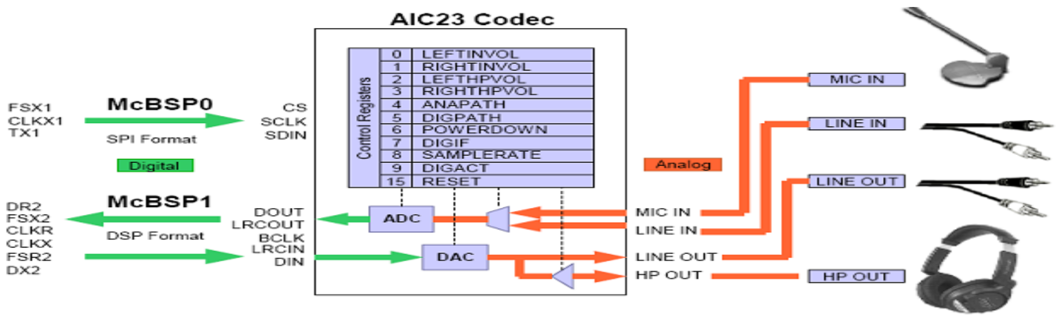
- A Texas Instruments TMS320C6416T DSP operating at 1 Gigahertz.
- An AIC23 stereo codec.
- 16 Mbytes of synchronous DRAM.
- 512 Kbytes of non-volatile Flash memory.
- 4 user accessible LEDs and DIP switches.
- Software board configuration through registers implemented in CPLD.
- Configured boot options and clock input selection.
- Standard expansion connectors for daughter card use.
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator.
- Single voltage power supply (+5V).

1.3 Identify the all Main Parts

1) TMS320C6416T DSP

The digital signal processor is considered to be the heart of the main system. It is designed to perform the beam forming as well as the source localization task. The processor used is a Texas Instruments C6416 DSP which operates at clock speed of 1 GHz. The processor comes preinstalled onto a DSP Starter Kit by Spectrum Digital, called as the TMS320C6416T DSK board .The DSK board makes it easier to interface the DSP with external peripherals, analogue to digital converters, external memory units, power supplies, and is controlled using the CCS Studio software (Texas Instruments) through a USB interface from a PC running on normal Microsoft Windows environment.

2) AIC23 stereo codec.



On the DSK board there is a TLV320AIC23 (AIC23) 16-bit stereo audio CODEC (coder/decoder). The chip has a mono microphone input, stereo line input, stereo line output and stereo headphone output. These outputs are accessible on the DSK board. The AIC23 figure shows a simplified block diagram of the AIC23 and its interfaces. The CODEC interfaces to the DSP through its McBSP serial interface. The CODEC is a 16-bit device and will be set up to deliver 16-bit signed 2's complement samples packed into a 32-bit word. Each 32-bit word will contain a sample from the left and right channel in that order. The data range is from - 2 (16−1) to (2 (16−1) -1) or -32768 to 32767.

3) 16 Mbytes of synchronous DRAM.

In order to keep up with the growing need for memory bandwidth at low cost, a new synchronous DRAM (SDRAM) architecture is proposed. The SDRAM has programmable latency, burst length, and burst type for wide variety of applications. The experimental 16M SDRAM (2M×8) achieves a 125-Mbyte/s data rate using 0.5-μm twin well CMOS technology.

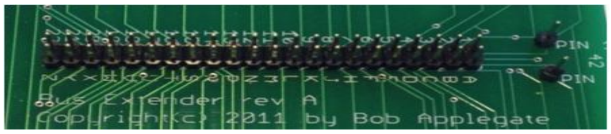
4)512 Kbytes of non-volatile Flash memory.

The DSK uses a 512Kbyte external Flash as a boot option. It is connected to CE1 of
EMIFB with an 8-bit interface. Flash is a type of memory which does not lose its
Contents when the power is turned off. When read it looks like a simple asynchronous
Read-only memory (ROM). Flash can be erased in large blocks commonly referred to
As sectors or pages. Once a block has been erased each word can be programmed
Once through a special command sequence. After than the entire block must be erased
Again to change the contents.
The Flash requires 70ns for both reads and writes. The general settings used with the
DSK use 8 cycles for both read and write strobes (80ns) to leave a little extra margin.

5) 4 user accessible LEDs and DIP switches.

The DSK includes 4 software accessible LEDs (D7-D10) and DIP switches (SW1) that
provide the user a simple form of input/output. Both are accessed through the CPLD
USER_REG register.

6) Standard expansion connectors



The memory connector provides access to the DSP’s asynchronous EMIF signals to Interface with memories and memory mapped devices. It supports byte addressing on 32 bit boundaries. The peripheral connector brings out the DSP’s peripheral signals Like McBSPs, timers, and clocks. Both connectors provide power and ground to the Daughter card.

7) JTAG emulator with USB host interface or external emulator.

The JTAG emulator is the bridge between the DSP and the PC. The JTAG emulator does all the work of talking to the DSP, and grants the PC direct access to the DSP's registers and on-chip peripherals. The JTAG emulator connects through the JTAG connector on each board, and comes in a variety of different ways to connect to the PC.
The USB and Parallel Port are the most common interfaces used.

8) Single voltage power supply (+5V).

The DSK operates from a single +5V external power supply connected to the main Power input (J5). Internally, the +5V input is converted into +1.2V and +3.3V using a Dual voltage regulator. The +1.2V supply is used for the DSP core while the +3.3V Supply is used for the DSP's I/O buffers and all other chips on the board. The power Connector is a 2.5mm barrel-type plug.

LAB TASK

1) See the reference manual with this kit and make a brief summary report on this kit.

LAB # 2

OBJECTIVE:

To Study & Start Installation Kit with PC.

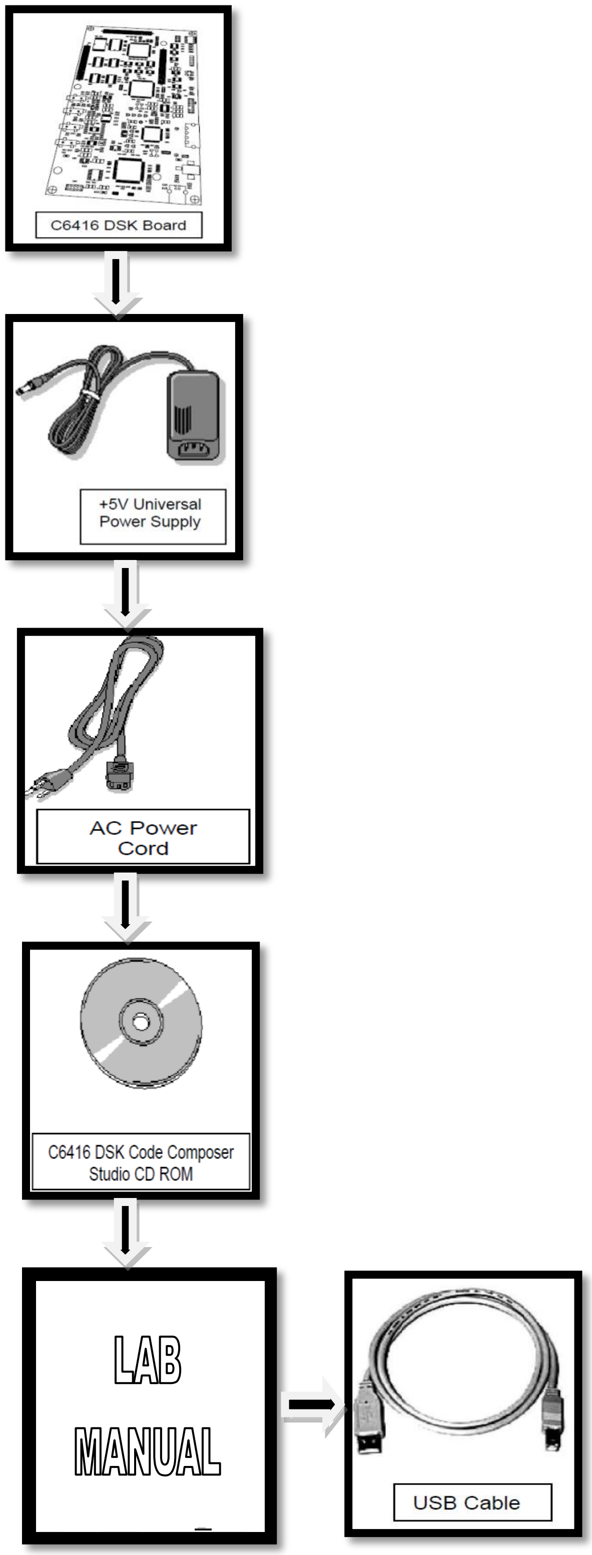
THEORY

The TMS320C6416 DSP Starter Kit (DSK) as shown in Fig. 1 developed jointly with Spectrum Digital is a low-cost development platform designed to speed the development of high performance applications based on TI´s TMS320C64x DSP generation. The kit uses USB communications for true plug-and-play functionality. Both experienced and novice designers can get started immediately with innovative product designs with the DSK´s full featured Code Composer Studio™ IDE and eXpressDSP™ Software which includes DSP/BIOS and Reference Frameworks.

The C6416 DSK tools includes the latest fast simulators from TI and access to the Analysis Toolkit via Update Advisor which features the Cache Analysis tool and Multi-Event Profiler. Using Cache Analysis developers improve the performance of their application by optimizing cache usage. By providing a graphical view of the on-chip cache activity over time the user can quickly determine if their code is using the on-chip cache to get peak performance.

The C6416 DSK allows you to download and step through code quickly and uses Real Time Data Exchange (RTDX™) for improved Host and Target communications. The DSK utilities include Flashburn to program flash, Update Advisor to download tools, utilities and software and a power on self test and diagnostic utility to ensure the DSK is operating correctly.

Kit Contents


C6416 DSK Board
+5V Universal Power Supply
AC Power Cord
C6416 DSK Code Composer Studio CD ROM
LAB MANUAL
USB Cable

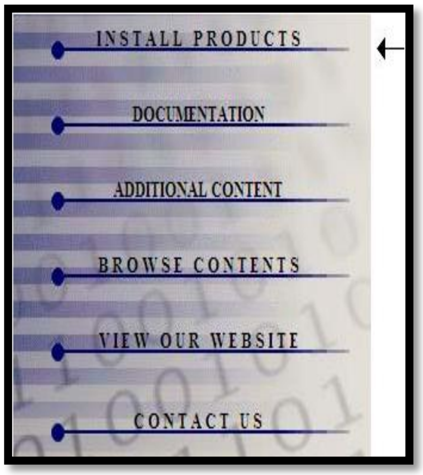
System Requirements

- 500MB of free hard disk space
- Microsoft Windows™ 2000/XP
- 128MB of RAM
- 16-bit color display
- CD-ROM Drive

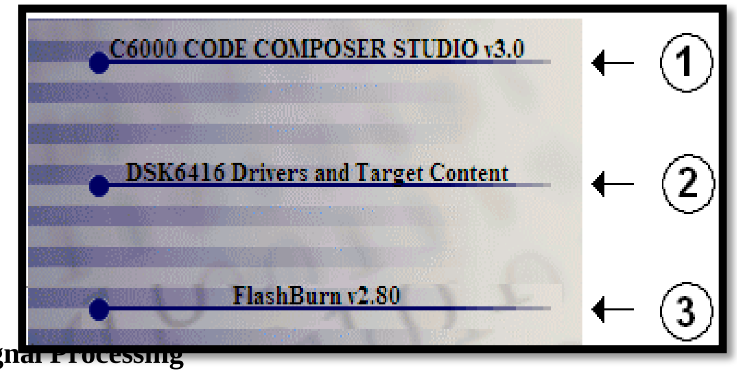
Install DSK Content from the CD-ROM

Before you install the DSK software, please make sure you are using **Administrator privileges** and any **virus checking** software is turned off. The DSK board should **not be plugged in** at this point.

1. Insert the Code Composer Studio installation CD into the CD-ROM Drive. An install menu (see below) should appear. If it does not, manually run Launch.exe from the CD-ROM. Select the **Install Products** option from the menu.



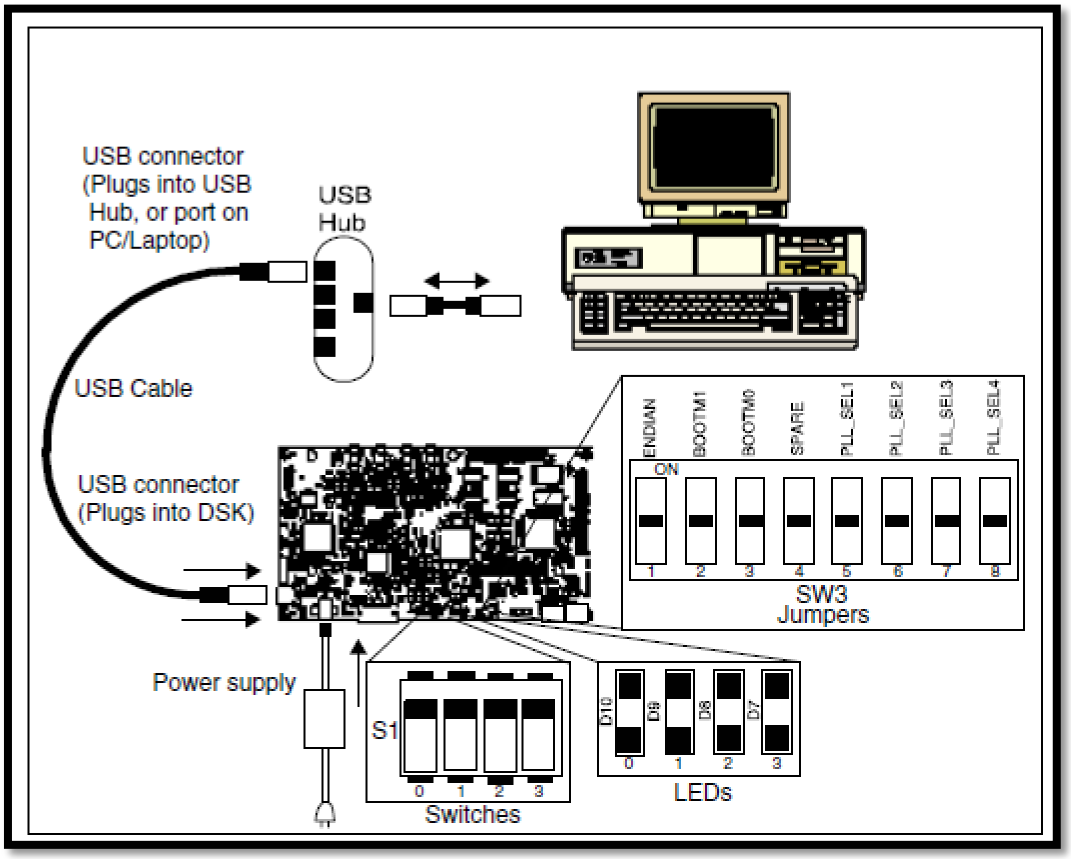
2. Install any components you need. To debug with the DSK you must have 1) a copy of **Code Composer Studio**, 2) the **target content** package for your board and 3) a copy of the **FlashBurn** plug-in.
Users of the full Code Composer Studio package can skip the DSK Code Composer installation and simply install the target content packages.


C6000 CODE COMPOSER STUDIO v3.0
DSK6416 Drivers and Target Content
FlashBurn v2.80
1
2
3

3. The installation procedure will create two icons on your desktop:
6416 DSK CCStudio 3
6416 DSK Diagnostics

Connect the DSK to Your PC

1. Connect the supplied USB cable to your PC or laptop. We recommend that anyone making hardware modifications connect through a USB hub for safety.
2. If you plan to connect a microphone, speaker, or expansion card these must be plugged in properly before you connect power to the DSK board.
3. Connect the included 5V power adapter brick to your AC power source using the AC power cord.
4. Apply power to the DSK by connecting the power brick to the 5V input on the DSK.
5. When power is applied to the board the Power On Self Test (POST) will run. LEDs 0-3 will flash. When the POST is complete all LEDs blink on and off then stay on. At this point your DSK is functional and you can now finish the USB driver install.



6. Make sure your DSK CD-ROM is installed in your CD-ROM drive. Now connect the DSK to your PC using the included USB. After few seconds Windows will launch its "Add New Hardware Wizard" and prompt for the location of the DSK drivers.
7. Follow the instructions on the screens and let Windows find the USB driver files "**sdusbemu.inf**" and "**sdusbemu.sys**" on the DSK CD-ROM. On XP systems Windows will find the drivers automatically.

Testing Your Connection

If you want to test your DSK and USB connection you can launch the C6416 DSK Diagnostic Utility from the icon on your desktop.

From the diagnostic utility, press the start button to run the diagnostics. In approximately 30 seconds all the on-screen test indicators should turn green.

LAB TASK

1) Turn on the Kit and go through the connectivity and self test operation.

LAB # 3

OBJECTIVE:

To Study Installation of Code Composer Studio & DSK Board Connection to PC.

THEORY

This Lab describes how to install and set up Code Composer Studio on your computer.
Code Composer Studio v4 is a major new release of Code Composer Studio (CCS) that is based on the Eclipse open source software framework. The Eclipse software framework is used for many different applications but it was originally developed as a open framework for creating development tools. We have chosen to base CCS on Eclipse as it offers an excellent software framework for building software development environments and is becoming a standard framework used by many embedded software vendors. CCSv4 combines the advantages of the Eclipse software framework with advanced embedded debug capabilities from Texas Instruments resulting in a compelling feature rich development environment for embedded developers.

Development phases of a **CCS** application are shown in fig 1;

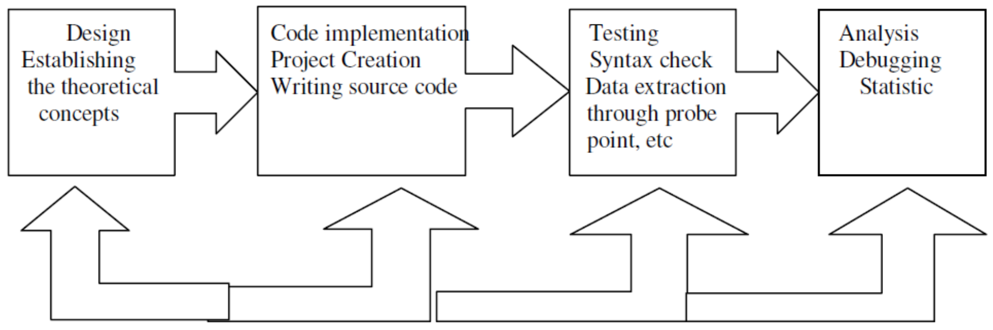


Fig.1 Development phases of a CCS application

Code Composer Studio Setup

Code Composer Studio Setup defines the target board or simulator you will use for your Code Composer Studio project. This definition is called the system configuration, and it consists of a device driver that handles communication with the target plus other information and files that describe the characteristics of your target, such as memory size and the peripherals present. A system configuration is needed even before building an application because the configuration determines which tools will be used by Code Composer Studio IDE.

System configurations come from two main sources - standard configurations included with the install, and custom created or modified configurations. Third parties who provide target boards or emulators may also provide predefined configurations or instructions for creating appropriate configurations. All configurations must include a device driver, as creating device drivers is beyond the scope of Code Composer Studio Setup.

The Setup utility allows you to import a defined system configuration, customize a system configuration, and export a system configuration for use by others. These options are covered in the remainder of this help.

If you have just installed Code Composer Studio, you will need to select a configuration to begin using the program.

Setup Interface

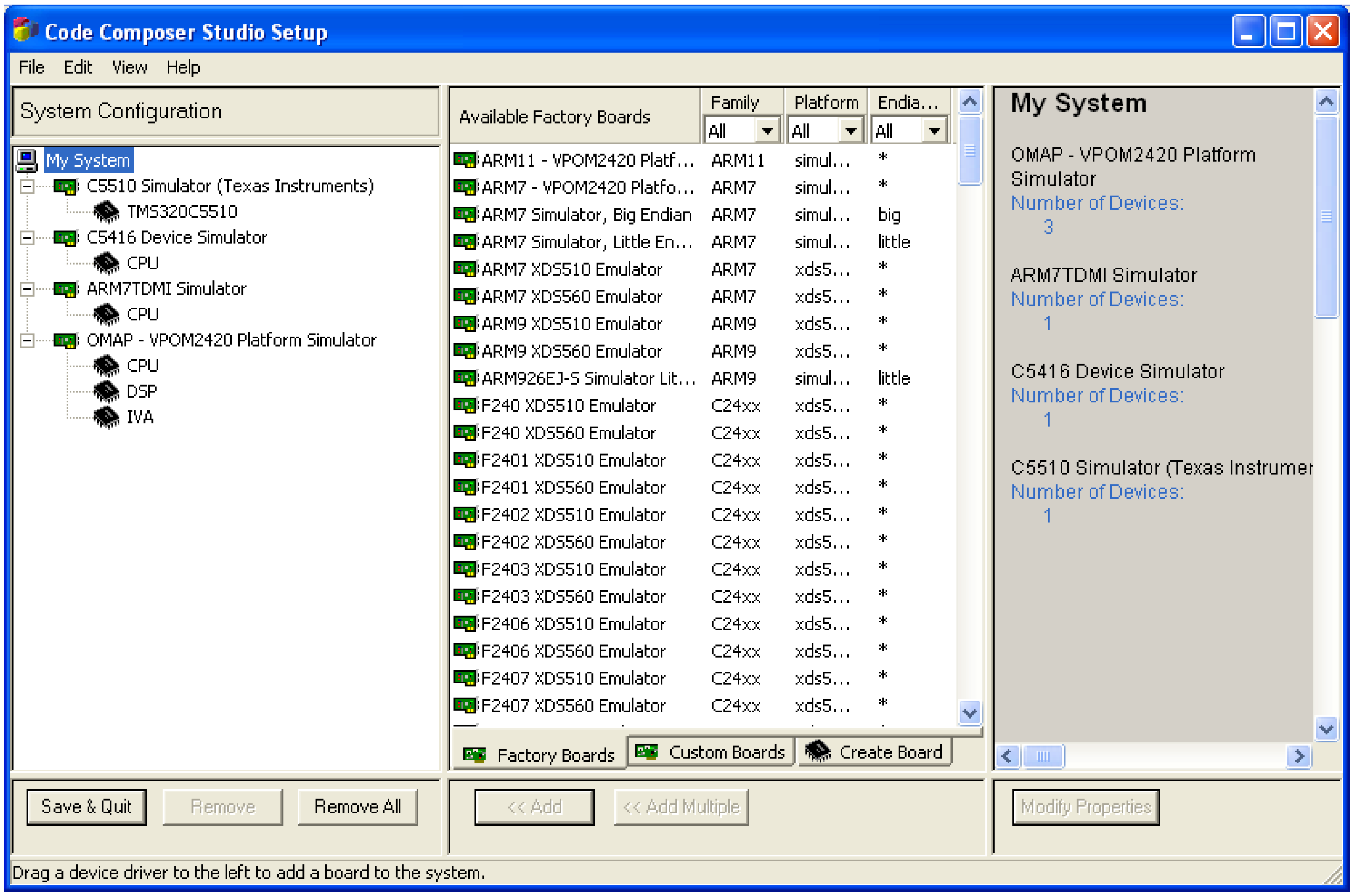
If you have not already done so, launch Setup by selecting File® Launch Setup within Code Composer Studio IDE, by selecting Start® All Programs® Texas Instruments® Code Composer Studio®.

Setup Code Composer Studio from the start menu, or by clicking the Setup CCStudio shortcut icon on your desktop.

After the initial setup process, when Setup is launched, the current configuration, or working configuration, is displayed. This configuration is stored in the System Registry. Changes made to the working configuration do not become permanent until it is saved or exported to a configuration file.

The Setup interface is divided into three panes:

1) System Configuration



Code Composer Studio Setup

File Edit View Help

System Configuration

```
My System
⊟ C5510 Simulator (Texas Instruments)
      TMS320C5510
⊟ C5416 Device Simulator
      CPU
⊟ ARM7TDMI Simulator
      CPU
⊟ OMAP - VPOM2420 Platform Simulator
      CPU
      DSP
      IVA
```

Available Factory Boards	Family	Platform	Endia...
	All	All	All
ARM11 - VPOM2420 Platf...	ARM11	simul...	*
ARM7 - VPOM2420 Platfo...	ARM7	simul...	*
ARM7 Simulator, Big Endian	ARM7	simul...	big
ARM7 Simulator, Little En...	ARM7	simul...	little
ARM7 XDS510 Emulator	ARM7	xds5...	*
ARM7 XDS560 Emulator	ARM7	xds5...	*
ARM9 XDS510 Emulator	ARM9	xds5...	*
ARM9 XDS560 Emulator	ARM9	xds5...	*
ARM926EJ-S Simulator Lit...	ARM9	simul...	little
F240 XDS510 Emulator	C24xx	xds5...	*
F240 XDS560 Emulator	C24xx	xds5...	*
F2401 XDS510 Emulator	C24xx	xds5...	*
F2401 XDS560 Emulator	C24xx	xds5...	*
F2402 XDS510 Emulator	C24xx	xds5...	*
F2402 XDS560 Emulator	C24xx	xds5...	*
F2403 XDS510 Emulator	C24xx	xds5...	*
F2403 XDS560 Emulator	C24xx	xds5...	*
F2406 XDS510 Emulator	C24xx	xds5...	*
F2406 XDS560 Emulator	C24xx	xds5...	*
F2407 XDS510 Emulator	C24xx	xds5...	*
F2407 XDS560 Emulator	C24xx	xds5...	*

Factory Boards | Custom Boards | Create Board

My System

OMAP - VPOM2420 Platform Simulator
Number of Devices:
3

ARM7TDMI Simulator
Number of Devices:
1

C5416 Device Simulator
Number of Devices:
1

C5510 Simulator (Texas Instrumer
Number of Devices:
1

Drag a device driver to the left to add a board to the system.

The System Configuration pane displays a hierarchical representation of the working system configuration. The top level is called My System. It cannot be modified. The second level displays the target (simulator, emulator, connection, or target board.) This level corresponds to particular XML file and device drivers that specify the different connections. The third level of hierarchy lists the CPUs or nondebuggable devices on your target. Some specific nondebuggable targets, such as TI's ICEPick™ technology, will also have a sub path and router path allowing for multiple target debugging.

The simplest system configurations include a single target and a single CPU. More complicated configurations are discussed under Parallel Debug Manager.

When you start Setup, the last saved system configuration is displayed as the working configuration.

- To clear the working configuration, select Remove All from the File menu.
- To clear one processor or connection, select Remove from the Edit menu.
- To import a configuration file, select Import from the File menu.
- To save the working configuration as the current configuration, select Save from the File menu.
- To export the working configuration to a file, select Export from the File menu.
- To exit, select Exit from the File menu.

2) <u>My System/Description</u>

The third (rightmost) pane lists all the target boards, processors, connections, or simulators in the System Configuration pane. It also lists the properties for each of the boards. If a specific board, processor, or connection is selected in the System Configuration pane, this pane will change to show the description for the connection, and the Modify Properties button will be available.

The third pane allows you to modify the following properties of the connection, depending on what has been selected in the other panes:

3) <u>Quick Setup</u>

Code Composer Studio Setup includes a set of standard configuration files that are predefined for the most common system configurations. If one of the standard files matches your system's configuration, simply load that file by adding it to your system, and you are ready to start a Code Composer Studio session.

<u>To Load a Standard Configuration File</u>

Invoke the setup application by double clicking on the Setup CCStudio icon
that appears on your desktop. The list of standard configurations is listed in the center pane of the setup program.

1) Click on the Factory Boards tab located at the bottom of the center pane. This will display a list of standard configurations provided by Texas Instruments or installed third-party packages.
2) Select the appropriate board or simulator configuration. Use Filters to help narrow the available choices, and either double click on the item, drag the selected configuration to the left pane or click on the configuration and hit the add button to add it to your system.
3) Repeat the step above for each board or simulator you wish to add to your system. Your configuration may contain more than one target board. Within the setup interface, the configuration you selected is graphically
displayed under My System in the left pane. If you select more than one target configuration in the setup, the Parallel Debug Manager (PDM) will be launched when you start Code Composer Studio IDE. The PDM lets you control multiple debug sessions.
4) Click the button labeled Save and Quit. The configuration is saved in the System Registry.
5) Click Yes to the question, “Start Code Composer Studio on Exit?” The Setup utility is closed and Code Composer Studio IDE is started.

System Requirements

To use Code Composer Studio, your operating platform must meet the
Following minimum requirements:

- IBM PC (or compatible)
- Microsoft Windows 95, Windows 98, or Windows NT 4.0
- 32 Mbytes RAM, 100 Mbytes of hard disk space, Pentium processor, SVGA display (800x600)

Installing Code Composer Studio

These operating platform requirements are necessary to install the Code
Composer Studio Integrated Development Environment (IDE):

Minimum

- 1-GHz CPU
- 512 MB of RAM
- 600 MB of free hard disk space
- 1024×768 display
- Internet Explorer™ 5.0 or later
- Local CD-ROM drive

Recommended

- 2-GHz CPU
- 2 GB of RAM
- 16-bit color

Supported Operating Systems

- Windows® 2000 Service Pack 4
- Windows XP Pro Service Pack 1 & 2

- Windows XP Home Service Pack 1 & 2

LAB TASK

1. Copy the template files into your temporary working directory, edit the project's directory as described above, and build the project in CCS. Connect your MP3 player to the line input of the DSK board and play your favorite song, or, you may play one of the wave files in the directory: c:\dsplab\wav.

2. Review the template project's build options using the menu commands: Project -> Build Options. In particular, review the Basic, Advanced, and Preprocessor options for the Compiler, and note that the optimization level was set to none. In future experiments, this may be changed to -o2 or -o3. For the Linker options, review the Basic and Advanced settings. In particular, note that the default output name a.out can be changed to anything else. Note also the library include paths and that the standard included libraries are:

   rts6700.lib (run-time library), C:\CCStudio_v3.1\C6000\cgtools\lib\rts6700.lib
   csl6713.lib (chip support library), C:\CCStudio_v3.1\C6000\csl\lib\csl6713.lib
   dsk6713bsl.lib
   (board support library), C:\CCStudio_v3.1\C6000\dsk6713\lib\dsk6713bsl.lib

3. The run-time library must always be included. The board support library (BSL) contains functions for managing the DSK board peripherals, such as the codec. The chip support library (CSL) has functions for managing the DSP chip's features, such as reading and writing data to the chip's McBSP. The user manuals for these may be found on the DSP course site and students should check the user manual for such options.

4. The gain parameter g can be controlled in real-time in two ways: using a watch window, or using a GEL file. Open a watch window using the menu item: View -> Watch Window, then choose View -> Quick Watch and enter the variable g and add it to the opened watch window using the item Add to Watch. Run the program and click on the g variable in the watch window and enter a new value, such as g = 0.5 or g = 2, and you will hear the difference in the volume of the output.

LAB # 4

OBJECTIVE:

To Study & Start Using Code Composer Studio Software.

THEORY

CCStudio's use as a programming tool and debugging tool comes only after you have configured CCStudio to account for your emulator and board. Configuration of CCStudio requires specific parameters in order to communicate with the emulator and board setup. Each emulator is communicated to differently and through a different I/O port. Each attached board has a unique JTAG scan chain built of individual processors.

Code Composer Studio IDE Key Benefits

• Quick start with familiar tools and interfaces
• Easily manage large multi-user, multi-site and multi-processor projects
• Utilize fast code creation, optimization and debugging tools
• Maximize reuse and portability for faster code development
• Perform real-time analysis enabled by RTDX and DSP/BIOS technologies

Starting Code Composer Studio

To open Code Composer Studio in a Windows 2000 system, click on Start *and select* Programs --> Texas Instruments --> Code Composer Studio DSK Tools 2 (‘C2000) --> Code Composer Studio.

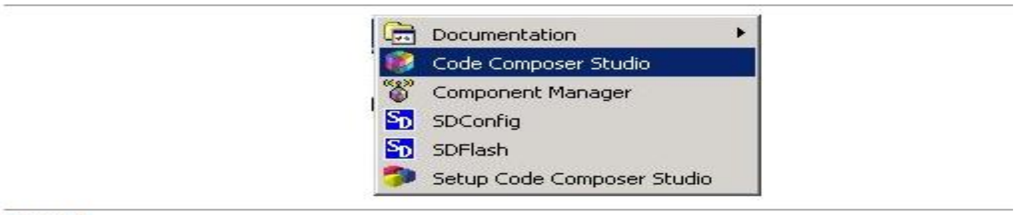


Figure 1

You should see the following screen:

\

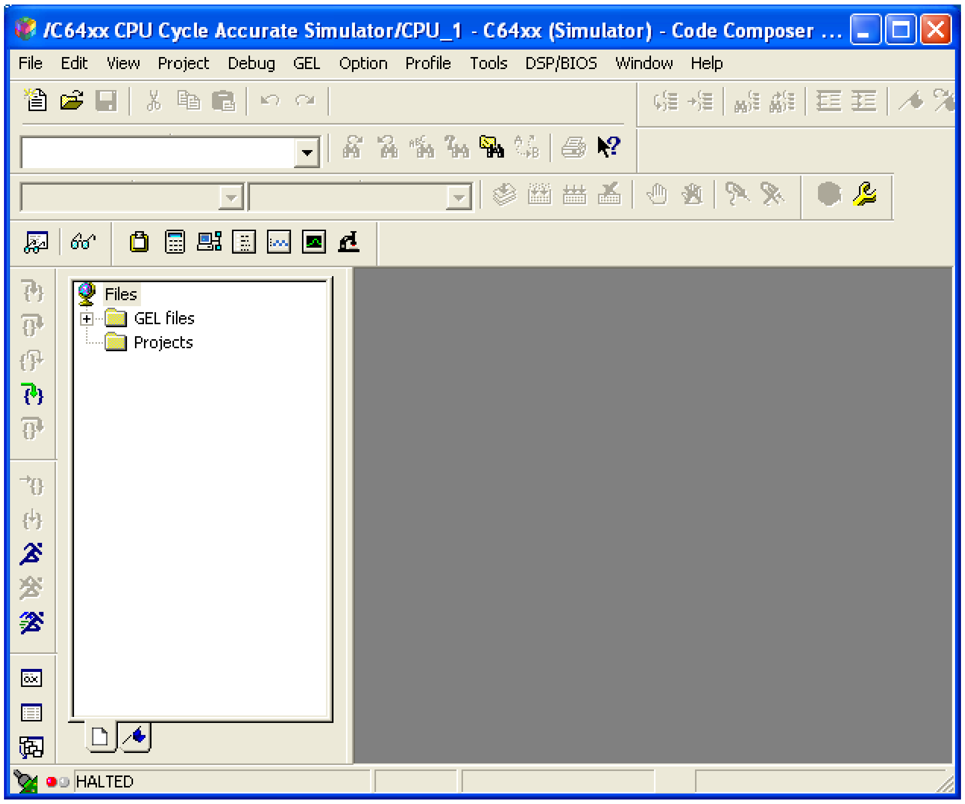
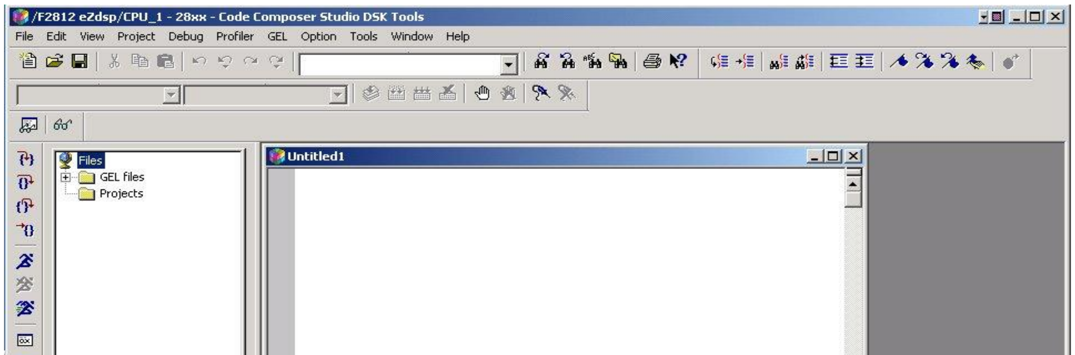

/C64xx CPU Cycle Accurate Simulator/CPU_1 - C64xx (Simulator) - Code Composer ...
File Edit View Project Debug GEL Option Profile Tools DSP/BIOS Window Help
Files
GEL files
Projects
HALTED

Figure 2

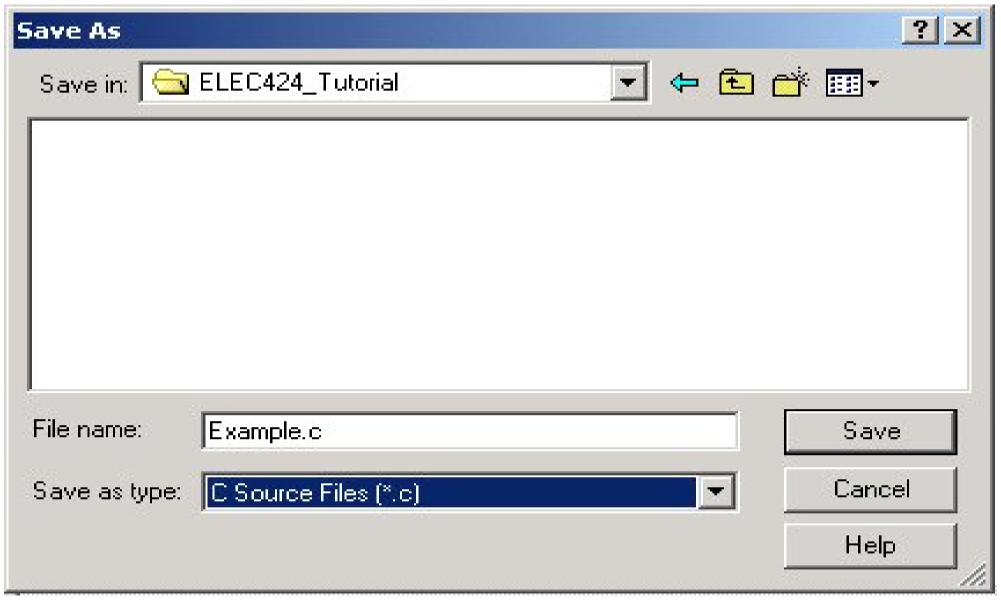
Creating a File

To create a new source code file, click on *File* --> *Source File*. A window titled '*Untitled1*' will appear within the Composer environment: **Figure 3**


/F2812 eZdsp/CPU_1 - 28xx - Code Composer Studio DSK Tools
File Edit View Project Debug Profiler GEL Option Tools Window Help
Files
GEL files
Projects
Untitled1

You can double-click on the '*Untitled1*' window to make it larger.

To save your file, click on *File* --> *Save*. The following prompt will appear



File name: Example.c

Save as type: C Source Files (*.c)

Figure 4

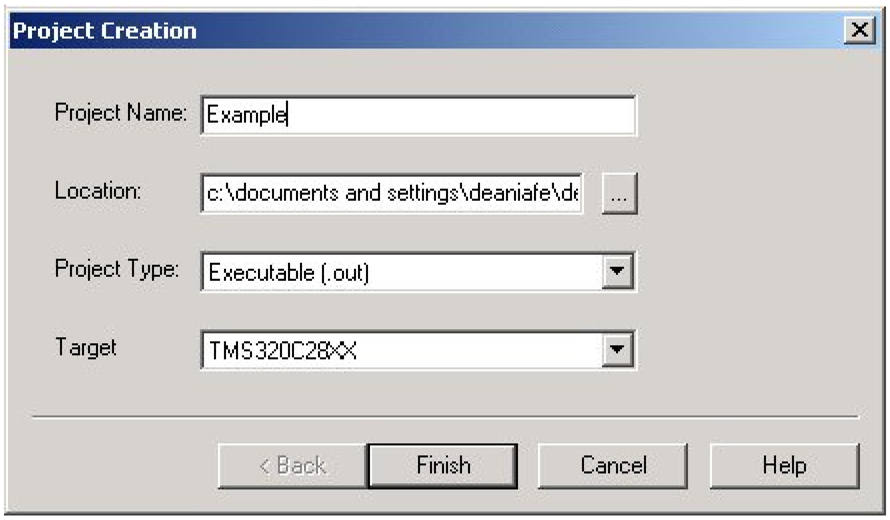
- **.lib** This library provides runtime support for the target DSP chip
- **.c** This file contains source code that provides the main functionality of this project
- **.h** This file declares the buffer C-structure as well as define any required constants
- **.pjt** This file contains all of your project build and configuration options

- **.asm** This file contains assembly instructions
- **.cmd** This file maps sections to memory

There are several *Save as type…* options. Choose the appropriate one for your code. It will give your file the proper file extension. It is also lets the linker know when you compile your code what kind of language the file is written in. When you are done, click *Save*.

Creating a New Project

After you have finished creating the files you need to run on the DSP, you will now want to create a project that will include all the files. Select *Project* --> *New…*



Project Creation

Project Name: Example

Location: c:\documents and settings\deaniafe\de

Project Type: Executable (.out)

Target TMS320C28XX

Figure 5

5. In the **Project Name** field, type Example.
6. In the **Location** field, browse to the working folder you created in step 1.
7. In the **Project Type** field, select Executable (.out).
8. In the **Target** field, select your target configuration and click **Finish**.
9. The Code Composer Studio™ Program creates a project file called volume1.pjt. This file stores your project settings and references the various files used by your project.

Example.pjt *will appear in the left-hand side of the Composer environment. Click on the '+' to expand the project:*

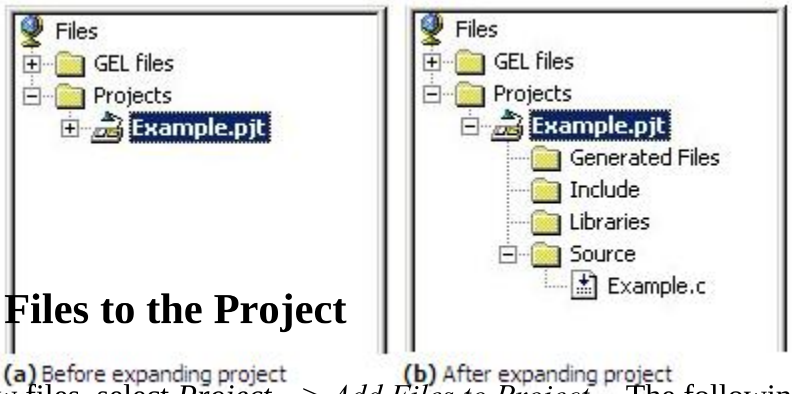
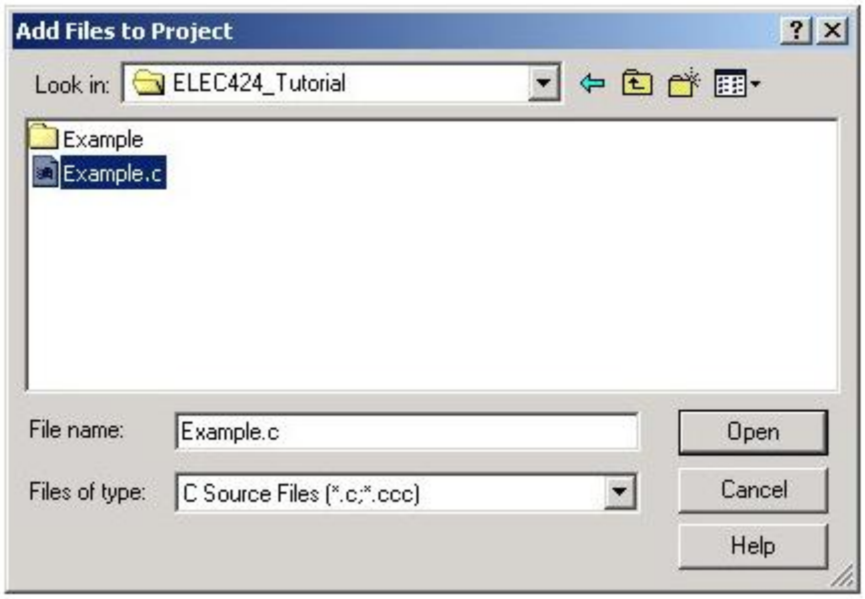


Figure 6

Adding New Files to the Project

In order to add new files, select *Project* --> *Add Files to Project…* The following prompt will appear:


Add Files to Project

Look in: ELEC424_Tutorial

Example
Example.c

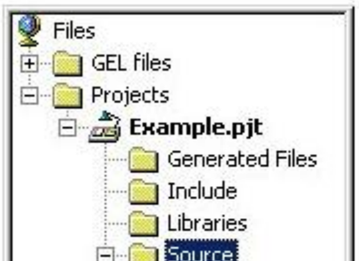
File name: Example.c

Files of type: C Source Files (*.c;*.ccc)

Figure 7

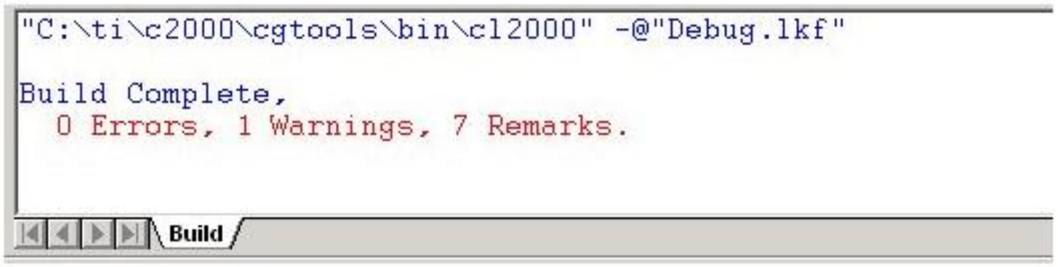
Select the file you want to add. To make it easier, you can narrow your search of a particular file by choosing which type of file you want in the *Files of type…*

When you are done, click *Open*. You should now see the file after clicking on the '+' in the left-hand window:



Compiling Files and Building Projects

Once you have added all the files you want to be included in your project, you will now need to build your project. Select *Project --> Rebuild All*. You should see the following window at the bottom of the Code Composer Studio Environment:



```
"C:\ti\c2000\cgtools\bin\cl2000" -@"Debug.lkf"

Build Complete,
  0 Errors, 1 Warnings, 7 Remarks.
```

Figure 9

If there are errors in your code, they will be listed with the corresponding line numbers. Correct them and rebuild your project.

Loading/Reloading Programs

After your code has been successfully compiled and built, you must now load your program onto the DSP. Select *File --> Load Program…* You will see the following prompt dialog:

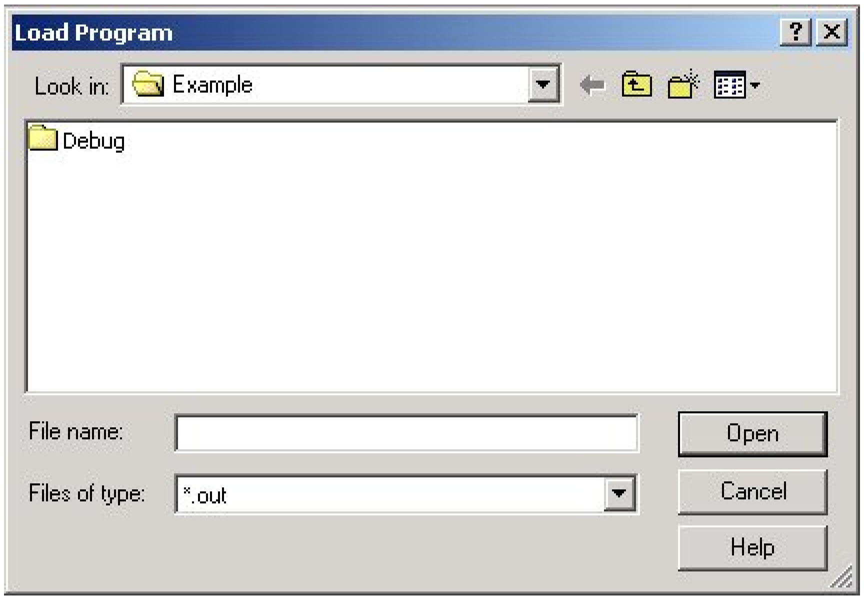
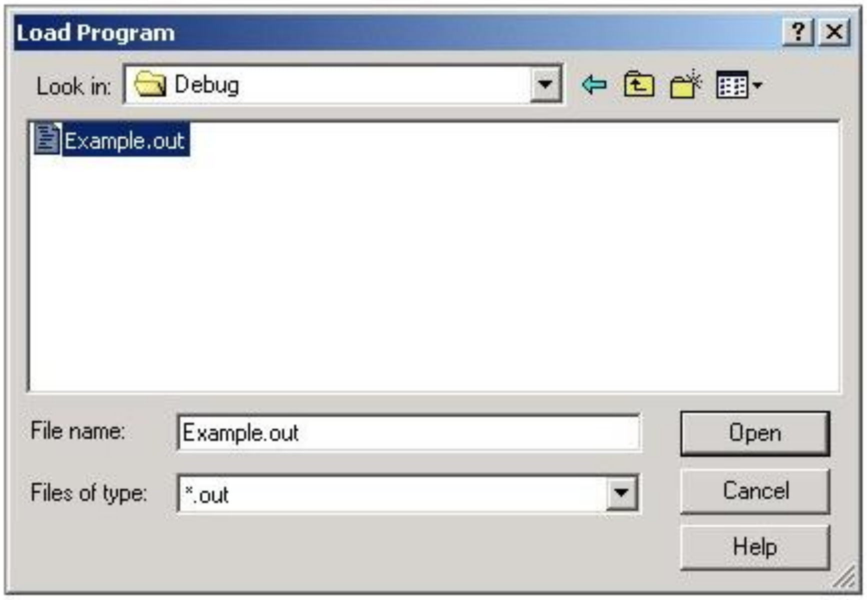


Figure 10

When you rebuild your project, Code Composer Studio is set at default to create a new folder in your directory called *Debug*. This is where the executable file is created. Double-click on the *Debug* folder and you should see you're **.out* file:



File name: Example.out
Files of type: *.out

Figure 11

After you select your file, click Open.

Executing, Halting, or Stopping Your Program

To execute your program, select *Debug* --> *Run* or press the *F5* key:

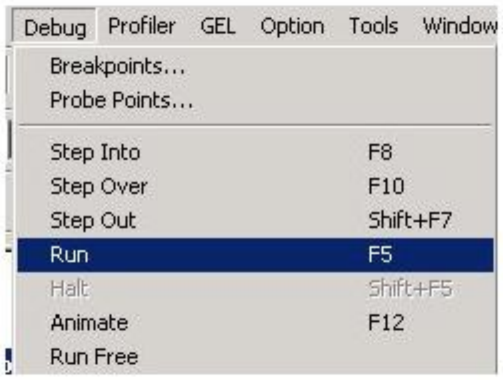
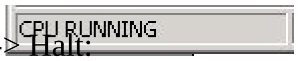


Figure 12

Your program will then begin to run. You will see the following at the bottom left-hand corner of the Code Composer Studio environment:



To stop running your code, select Debug --> Halt:

Figure 13

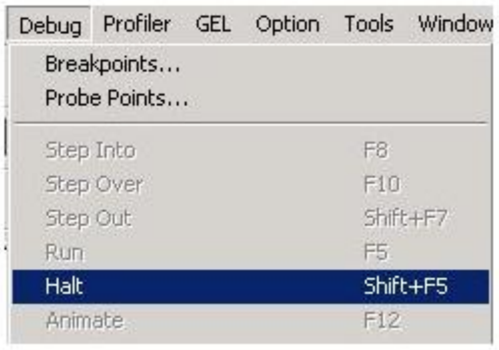


Figure 14

You should then see the following at the bottom left-hand corner of the work environment:

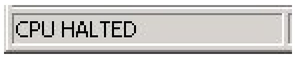


Figure 15

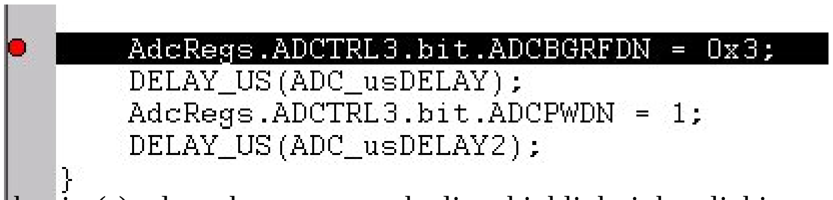
To resume running your code, press the F5 button.

Debugging Your Code

Since there are few of us that can get our code working right the first time, you will probably have to debug your code. To figure out what could be wrong, there are several methods you can use to break the problem down.

Setting Breakpoints

To execute your code a little at a time or to stop it after a certain point, you can place breakpoints. You can do this by placing the cursor on the line you want to set the breakpoint on, highlighting it by clicking once, and double-click. You should see a solid red circle on the left:

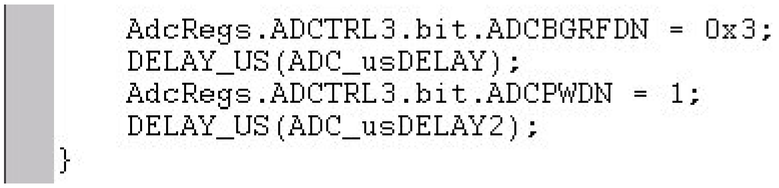


```
●    AdcRegs.ADCTRL3.bit.ADCBGRFDN = 0x3;
     DELAY_US(ADC_usDELAY);
     AdcRegs.ADCTRL3.bit.ADCPWDN = 1;
     DELAY_US(ADC_usDELAY2);
}
```

To remove the breakpoint(s), place the cursor on the line, highlight it by clicking once, and double-click. The solid red circle should disappear:

Figure 16

You can set as many as you like. Rebuild and reload the program. Execute it. The DSP will stop at the first breakpoint. To get to the next breakpoint, press the *F5* button to run the DSP again.



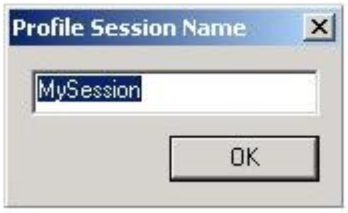
```
     AdcRegs.ADCTRL3.bit.ADCBGRFDN = 0x3;
     DELAY_US(ADC_usDELAY);
     AdcRegs.ADCTRL3.bit.ADCPWDN = 1;
     DELAY_US(ADC_usDELAY2);
}
```

Figure 17

Profiling Sessions

You can benchmark your code by profiling a session. To do this, you can set one breakpoint at the line where you want the counter to start counting and another breakpoint at where to stop.

Select Profiler --> Start New Session… You will see the following dialog:

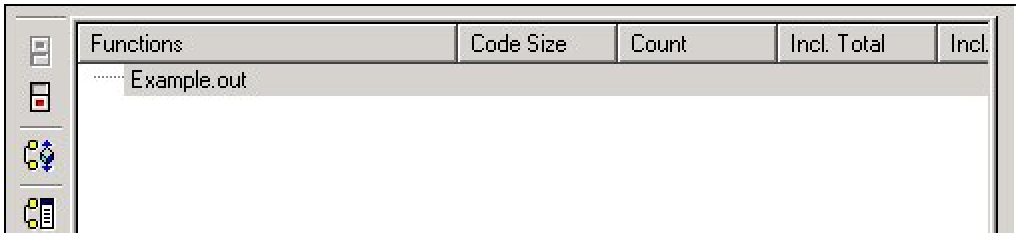


Profile Session Name

MySession

Figure 23

Type in a name for your profile session and click *OK*. A window will appear at the bottom:



Functions	Code Size	Count	Incl. Total	Incl.
Example.out				

To profile a function, you need to add it to the profile session you just created. Double-click on the file you want to do this in, and place the cursor on any line inside the function and right click. Choose Profile Function --> in (session-name*)* Session.

Execute the code. When the CPU stops, click on the '*Functions*' tab to see the result of the profiling.

Troubleshooting

If you are having any trouble, sometimes it is best to reset the CPU. You can do this by selecting Debug --> Restart CPU. You will have to reload the program.

Another option is to simply quit Code Composer Studio by selecting File --> *Quit* and restart the application.

SAMPLE PROGRAM # 1

Code:

```
// template.c - to be used as starting point for interrupt-based programs
// ---------------------------------------------------------------------------------
#include "dsplab.h" // DSK initialization declarations and function prototypes

short xL, xR, yL, yR; // left and right input and output samples from/to codec
float g=1; // gain to demonstrate watch windows and GEL files

// here, add more global variable declarations, #define's, #include's, etc.
// ---------------------------------------------------------------------------------

void main() // main program executed first
{
        initialize(); // initialize DSK board and codec, define interrupts
        sampling_rate(8); // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
        audio_source(LINE); // LINE or MIC for line or microphone input
        while(1); // keep waiting for interrupt, then jump to isr()
}

// --------------------------------------------------------------------------------
interrupt void isr()      // sample processing algorithm - interrupt service routine
{

        read_inputs(&xL, &xR); // read left and right input samples from codec
        yL = g * xL; // replace these with your sample processing algorithm
        yR = g * xR;
        write_outputs(yL,yR); // write left and right output samples to codec
        return;

}

// --------------------------------------------------------------------------------
// here, add more functions to be called within isr() or main()
```

Rebuild your program with these changes and play a song. In your lab write-up explain why and how this code works.

LAB # 5

OBJECTIVE:

Performing sampling, quantization, DSP codec integration, MATLAB code Integration and exploring the effects of aliasing on sound signals.

THEORY

This part demonstrates aliasing effects. The smallest sampling rate that can be defined is 8 kHz with a Nyquist interval of [−4, 4] kHz. Thus, if a sinusoidal signal is generated (e.g. with MATLAB) with frequency outside this interval, e.g., f = 5 kHz, and played into the line-input of the DSK, one might expect that it would be aliased with fa = f − fs = 5 − 8 = −3 kHz. However, this will not work because the anti-aliasing oversampling decimation filters of the codec filter out any such out-of-band components before they are sent to the processor.

An alternative is to decimate the signal by a factor of 2, i.e., dropping every other sample. If the codec sampling rate is set to 8 kHz and every other sample is dropped, the effective sampling rate will be 4 kHz, with a Nyquist interval of [−2, 2] kHz. A sinusoid whose frequency is outside the decimated Nyquist interval [−2, 2] kHz, but inside the true Nyquist interval [−4, 4] kHz, will not be cut off by the ant aliasing filter and will be aliased. For example, if f = 3 kHz, the decimated sinusoid will be aliased with fa = 3 − 4 = −1 kHz and even −5 kHz.

5.1 Lab Procedure

Copy the template programs to your working directory. Set the sampling rate to 8 kHz and select line input. Modify the template program to output every other sample, with zero values in-between. This can be accomplished in different ways, but a simple one is to define a “sampling pulse” periodic signal whose values alternate between 1 and 0, i.e., the sequence [1,0,1,0,1,0,...] and multiply the input samples by that sequence. The following simple code segment implements this idea:

1) **Code:**

```
yL = pulse * xL;
yR = pulse * xR;
pulse = (pulse==0);
```

where pulse must be globally initialized to 1 before *main()* and *isr()*. Next, rebuild the new program with CCS.

5.2 MATLAB Code Integration

Open MATLAB and generate three sinusoids of frequencies f1 = 1 kHz, f2 = 3 kHz, and f3 = 1 kHz, each of duration of 1 second, and concatenate them to form a 3-second signal. Then play this out of the PCs sound card using the *sound()* function. For example, the following MATLAB code will do this:

```
fs = 8000; f1 = 1000; f2 = 3000; f3 = 1000;
L = 8000; n = (0:L-1);
A = 1/5; % adjust playback volume
x1 = A * cos(2*pi*n*f1/fs);
x2 = A * cos(2*pi*n*f2/fs);
x3 = A * cos(2*pi*n*f3/fs);
sound([x1,x2,x3], fs);
```

Connect the sound card’s audio output to the line-input of the DSK and rebuild/run the CCS down sampling program after commenting out the line pulse = (pulse==0); This disables the down sampling operation. Send the above concatenated sinusoids to the DSK input and you should hear three distinct 1-sec segments, with the middle one having a higher frequency.

Now, uncomment the above line so that down sampling takes place and rebuild/run the program. Send the concatenated sinusoids to the DSK and you should hear all three segments as though they have the same frequency (because the middle 3 kHz one is aliased with other ones at 1 kHz). You may also play your favorite song to hear the aliasing distortions, e.g., out of tune vocals.

Now set the codec sampling rate to 44 kHz and repeat the previous two steps. What do you expect to hear in this case?

To confirm the anti-aliasing pre-filtering action of the codec, replace the first two lines of the above MATLAB code by the following two:

fs = 16000; f1 = 1000; f2 = 5000; f3 = 1000;
L = 16000; n = (0:L-1);

Now, the middle sinusoid has frequency of 5 kHz and it should be cutoff by the anti-aliasing prefilter. Set the sampling rate to 8 kHz, turn off the down sampling operation, rebuild and run your program, and send this signal through the DSK, and describe what you hear.

5.3 Quantization

The DSK's codec is a 16-bit ADC/DAC with each sample represented by a two's complement integer. Given the 16-bit representation of a sample, $[b_1\ b_2\ \dots\ b_{16}]$, the corresponding 16-bit integer is given by:

$$x = (-b_1 2^{-1} + -b_2 2^{-2} + -b_3 2^{-3} + \cdots + -b_{16} 2^{-16}) 2^{16}$$

The MSB bit b_1 is the sign bit. The range of represent able integers is: $-32768 \leq x \leq 32767$. For high-fidelity audio at least 16 bits are required to match the dynamic range of human hearing; for speech, 8 bits are sufficient. If the audio or speech samples are quantized to less than 8 bits, quantization noise will become audible. The 16-bit samples can be requantized to fewer bits by a right/left bit-shifting operation. For example, right shifting by 3 bits will knock out the last 3 bits, then left shifting by 3 bits will result in a 16-bit number whose last three bits are zero, that is, a 13-bit integer. These operations are illustrated below:

$$[b_1, b_2, \dots, b_{13}, b_{14}, b_{15}, b_{16}] \Rightarrow [0, 0, 0, b_1, b_2, \dots, b_{13}] \Rightarrow [b_1, b_2, \dots, b_{13}, 0, 0, 0]$$

1) Code and procedure:

Modify the basic template program so that the output samples are requantized to B bits, where $1 \leq B \leq 16$. This requires right/left shifting by $L = 16 - B$ bits, and can be implemented very simply in C as follows:

yL = (xL >> L) << L;
yR = (xR >> L) << L;

Start with *B* = 16, set the sampling rate to 8 kHz, and rebuild/run the program. Send a wave file as input and listen to the output. Now repeat with the following values: *B* = 8, 6, 4, 2, 1, and listen to the gradual increase in the quantization noise.

5.4 Data Transfers from/to Codec

The codec samples the input in stereo, combines the two 16-bit left/right samples *xL*, *xR* into a single 32-bit unsigned integer word, and ships it over to a 32-bit receive register of the McBSP of the C6416 processor. This is illustrated in Fig. 1.

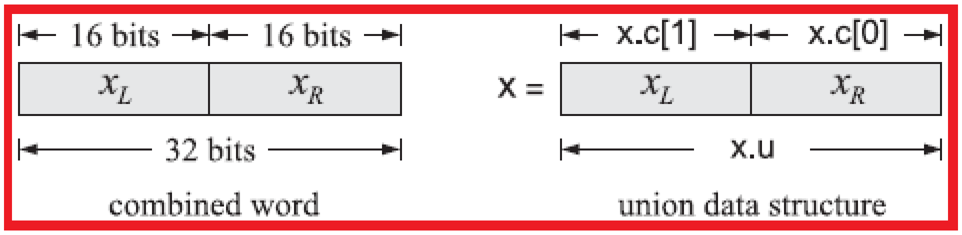


Figure 1: Sample formation in Codec of the DSP board

The packing and unpacking of the two 16-bit words into a 32-bit word is accomplished with the help of a union data structure defined as follows:

```
union {                              // union structure to facilitate 32-bit data transfers
        Uint32 u;                    // both channels packed as codec.u = 32-bits
        short c[2]; // left-channel = codec.c[1], right-channel = codec.c[0]
} codec;
```

The two members of the data structure share a common 32-bitmemory storage. The member codec.u contains the 32-bit word whose upper 16 bits represent the left sample, and its lower 16 bits, the right sample. The two-dimensional short array member codec.c holds the 16-bit right-channel sample in its first component, and the left-channel sample in its second, that is, we have:

xL = codec.c[1];
xR = codec.c[0];

The functions *read_inputs()* and *write outputs()*, which are defined in the common file dsplab.c, use this structure in making calls to low-level McBSP read/write functions of the chip support library. They are defined as follows:

1) **A. Code:**

```
void read_inputs(short *xL, short *xR)        // read left/right channels
```

```
{
        codec.u = MCBSP_read(DSK6713_AIC23_DATAHANDLE);       // read 32-bit word
        *xL = codec.c[1]; // unpack the two 16-bit parts
        *xR = codec.c[0];
}

// ---------------------------------------------------------------------------

void write_outputs(short yL, short yR)          // write left/right channels
{
        codec.c[1] = yL; // pack the two 16-bit parts
        codec.c[0] = yR; // into 32-bit word
        MCBSP_write(DSK6713_AIC23_DATAHANDLE,codec.u);   //output left/right samples
}
```

B. Procedure:

The purpose of this task is to clarify the nature of the union data structure. Copy the template files into your working directory, rename them unions.*, and edit the project file by keeping in the source-files section only the run-time library and the main function below.

2) A. Code:

```
// unions.c - test union structure
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void main(void)
{
        unsigned int v;
        short xL,xR;
        union {
                        unsigned int u;
                        short c[2];
              } x;

        xL = 0x1234;
        xR = 0x5678;
        v = 0x12345678;
        printf("\n%x %x %d %d\n", xL,xR, xL,xR);
        x.c[1] = xL;
        x.c[0] = xR;
        printf("\n%x %x %x %d %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);
        x.u = v;
        printf("%x %x %x %d %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);
        x.u = (((int) xL)<<16 | ((int) xR) & 0x0000ffff);
        printf("%x %x %x %d %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);

}
```

B. Code Explanation:

The program defines first a union structure variable x of the codec type. Given two 16-bit left/right numbers xL,xR (specified as 4-digit hex numbers), it defines a 32-bit unsigned integer v which is the concatenation of the two. The first printf statement prints the two numbers xL,xR in hex and decimal format. Note that the hex printing conversion operator %x treats the numbers as unsigned (some caution is required when printing negative numbers), whereas the decimal operator %d treats them as signed integers.

Next, the numbers xL,xR are assigned to the array members of the union x, such that x.c[1] = xL and x.c[0] = xR, and the second printf statement prints the contents of the union x, verifying that the 32-bit member x.u contains the concatenation of the two numbers with xL occupying the upper 16 bits, and xR, the lower 16 bits. Explain what the other two printf statements do.

Build and run the project (you may have to remove the file vectors.asm from the project's list of files). The output will appear in the stdout window at the bottom of the CCS. Alternatively, you may run this outside CCS using GCC. To do so, open a DOS window in your working directory and type the DOS command djgpp. This establishes the necessary environment variables to run GCC, then, run the following GCC command to generate the executable file unions.exe:

```
gcc unions.c -o unions.exe –lm
```

Repeat the run with the following choice of input samples:

xL = 0x1234;
xR = 0xabcd;
v = 0x1234abcd;

Explain the outputs of the print statements in this case by noting the following properties, which you should prove in your report:

$(0xffff0000)_{unsigned} = 2^{32} - 2^{16}$
$(0xffffabcd)_{unsigned} = 2^{32} + (0xabcd)_{signed}$
$(0xffffabcd)_{signed} = (0xabcd)_{signed}$

5.5 Guitar Distortion Effects

Usually the input/output maps are memory less. But there are several implementations where memory units are needed. A memory less mapping can be linear but time-varying, for example the case of panning between the speakers or the AM/FM wavetable. The mapping can also be nonlinear. Many guitar distortion effects combine delay effects with such nonlinear maps. In this section we will study only some nonlinear maps in which each input sample *x* is mapped to an output sample *y* by a nonlinear function *y = f(x)*. Typical examples are hard clipping (called fuzz) and soft clipping that tries to emulated the nonlinearities of tube amplifiers.

A typical nonlinear function is *y = tanh(x)*. It has a sigmoidal shape that you can see by the quick MATLAB plot:

fplot('tanh(x)', [-4,4]); grid;

By keeping only the first two terms in its Taylor series expansion, that is, *tanh(x)* ≈ *x*-*x*3/3, we may define a more easily realizable nonlinear function with built-in soft clipping:

$$y = f(x) = \begin{cases} +2/3, & x \geq 1 \\ x - x^3/3, & -1 \leq x \leq 1 \\ -2/3, & x \leq -1 \end{cases}$$

In MATLAB, this can be plotted by

fplot('(abs(x)<1).(x-1/3*x.^3) + sign(x).*(abs(x)>=1)*2/3', [-4,4]); grid;*

The threshold value of 2/3 is chosen so that the function f(x) is continuous at x = ±1. To add some flexibility and to allow a variable threshold, we consider the following modification:

$$y = f(x) = \begin{cases} +\alpha c, & x \geq c \\ x - \beta c(x/c)^3, & -c \leq x \leq c \\ -\alpha c, & x \leq -c \end{cases} , \qquad \beta = 1 - \alpha$$

where we assume that c > 0 and 0 < α < 1. The choice β = 1 − α is again dictated by the continuity requirement at x = ±c. Note that setting α = 1 gives the hard-thresholding, fuzz, effect:

$$y = f(x) = \begin{cases} +c, & x \geq c \\ x, & -c \leq x \leq c \\ -c, & x \leq -c \end{cases}$$

Lab Tasks:

1) Complete all the previous tasks

2) For Guitar distortion, perform the following:

   - First, run the above two fplot commands in MATLAB to see what these functions look like. The following program is a modification of the basic template.c program that implements the above non linear equation:

Code:

```
// soft.c - guitar distortion by soft thresholding
// ------------------------------------------------------------------------------
#include "dsplab.h" // init parameters and function prototypes
#include <math.h>
// ------------------------------------------------------------------------------
#define a 0.67 // approximates the value 2/3
#define b (1-a)
short xL, xR, yL, yR; // codec input and output samples
int x, y, on=1, c=2048; // on/off variable and initial threshold c
int f(int); // function declaration
// ------------------------------------------------------------------------------
void main() // main program executed first
{
        initialize(); // initialize DSK board and codec, define interrupts
        sampling_rate(16); // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
        audio_source(LINE); // LINE or MIC for line or microphone input
        while(1); // keep waiting for interrupt, then jump to isr()
}
// ------------------------------------------------------------------------------
interrupt void isr() // sample processing algorithm - interrupt service routine
```

```
{
        read_inputs(&xL, &xR); // read left and right input samples from codec
        if (on) {
                yL = (short) f((int) xL); yL = yL << 1; // amplify by factor of 2
                yR = (short) f((int) xR); yR = yR << 1;
        }
        else
        {yL = xL; yR = xR;}
write_outputs(yL,yR); // write left and right output samples to codec
return;
}
// -------------------------------------------------------------------------------
int f(int x)
{
        float y, xc = x/c; // this y is local to f()
        y = x * (1 - b * xc * xc);
        if (x>c) y = a*c; // force the threshold values
        if (x<-c) y = -a*c;
        return ((int) y);
}
// --------------------------------------------------------------
```

- Create a project for this program. In addition, create a GEL file that has two sliders, one for the on variable that turns the effect on or off in real time, and another slider for the threshold parameter c. Let c vary over the range [0, 214] in increments of 512.

- Build and run the program, load the gel file, and display the two sliders. Then, play your favorite guitar piece and vary the slider parameters to hear the changes in the effect. (use any guitar wave file for this purpose)

- Repeat the previous part by turning off the nonlinearity (i.e., setting $\alpha = 1$), which reduces to a fuzz effect with hard thresholding.

LAB # 6

OBJECTIVE:

To implement AM and FM modulations on different signals, the concept of a wavetable is introduced and applied first to the generation of sinusoidal signals of different frequencies and then to square waves. AM and FM examples are constructed by combining two wavetables. Ring modulation and tremolo audio effects are studied as special cases of AM modulation.

THEORY

A wavetable is defined by a circular buffer ***w*** whose dimension *D* is chosen such that the smallest frequency to be generated is:

$$f_{\min} = \frac{f_s}{D} \quad \Rightarrow \quad D = \frac{f_s}{f_{\min}}$$

For example, if fs = 8 kHz and the smallest desired frequency is fmin = 10 Hz, then one must choose D = 8000/10 = 800. The D-dimensional buffer holds one period at the frequency fmin of the desired waveform to be generated. The shape of the stored waveform is arbitrary, and can be a sinusoid, a square wave, sawtooth, etc. For example, if it is sinusoidal, then the buffer contents will be:

$$w[n] = \sin\left(\frac{2\pi f_{\min}}{f_s}\, n\right) = \sin\left(\frac{2\pi n}{D}\right), \quad n = 0, 1, \ldots, D-1$$

Similarly, a square wave whose first half is +1 and its second half, −1, will be defined as:

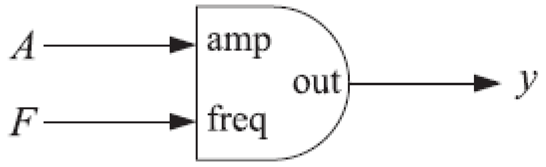
$$w[n] = \begin{cases} +1, & \text{if } 0 \le n < D/2 \\ -1, & \text{if } D/2 \le n < D \end{cases}$$

To generate higher frequencies (with the Nyquist frequency fs/2 being the highest), the wavetable is cycled in steps of c samples, where c is related to the desired frequency by:

$$f = c f_{\min} = c\frac{f_s}{D} \quad \Rightarrow \quad c = D\frac{f}{f_s} \equiv DF, \quad F = \frac{f}{f_s}$$

where F = f/fs is the frequency in units of [cycles/sample]. The generated signal of frequency f and amplitude A is obtained by the loop shown below:

```
repeat forever:
    $y = A\,w[q]$
    $q = (q + c)\bmod(D)$
```



The shift c need not be an integer. In such case, the quantity q + c must be truncated to the integer just below it. The truncation method will suffice for approximating values. The following function, *wavgen()*,implements this algorithm. The mod-operation is carried out with the help of the function *qwrap()*:

1) Code:

```
// -------------------------------------------------
// wavgen.c - wavetable generator
// Usage: y = wavgen(D,w,A,F,&q);
// ------------------------------------------------
int qwrap(int, int);
float wavgen(int D, float *w, float A, float F, int *q)
{
float y, c=D*F;
y = A * w[*q];
*q = qwrap(D-1, (int) (*q+c));
return y;
}
```

We note that the circular index q is declared as a pointer to int, and therefore, must be passed by address in the calling program. Before using the function, the buffer **w** must be loaded with one period of length D of the desired waveform.

6.1 Sinusoidal Wavetable

The following program, sine0.c, generates a 1 kHz sinusoid from a wavetable of length D = 4000. At a sampling rate of 8 kHz, the smallest frequency that can be generated is fmin = fs/D = 8000/4000 = 2 Hz. In order to generate f = 1 kHz, the step size will be c = D · f/fs = 4000 · 1/8 = 500 samples. In this example, we will not use the function *wavgen()* but rather apply the generation algorithm mentioned above explicitly. In addition, we will save the output samples in a buffer array of length N = 128 and inspect the generated waveform both in the time and frequency domains using CCS's graphing capabilities.

1) **Code:**

```
// sinex.c - sine wavetable example
//
// ----------------------------------------------------------------------------------
#include "dsplab.h" // DSK initialization declarations and function prototypes
#include <math.h>
//#define PI 3.141592653589793
short xL, xR, yL, yR; // left and right input and output samples from/to codec
#define D 4000 // fmin = fs/D = 8000/4000 = 2 Hz
#define N 128 // buffer length
short fs=8; // fs = 8 kHz
float c, A=5000, f=1; // f = 1 kHz
float w[D]; // wavetable buffer
float buffer[N]; // buffer for plotting with CCS
int q=0, k=0;
// ----------------------------------------------------------------------------------

void main() // main program executed first
{
int n;
float PI = 4*atan(1);
for (n=0; n<D; n++) w[n] = sin(2*PI*n/D); // load wavetable with one period
c = D*f/fs; // step into wavetable buffer
initialize(); // initialize DSK board and codec, define interrupts
sampling_rate(fs); // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
// audio_source(LINE); // LINE or MIC for line or microphone input
while(1); // wait for interrupts
}
// ----------------------------------------------------------------------------------

interrupt void isr()
{
yL = (short) (A * w[q]); // generate sinusoidal output
q = (int) (q+c); if (q >= D) q = 0; // cycle over wavetable in steps c
buffer[k] = (float) yL; // save into buffer for plotting
if (++k >= N) k=0; // cycle over buffer
write_outputs(yL,yL); // audio output
}
// --------------------------------------------------------------------------------
```

The wavetable is loaded with the sinusoid in *main()*. At each sampling instant, the program does nothing with the codec inputs, rather, it generates a sample of the sinusoid and sends it to the codec, and saves the sample into a buffer (only the last N generated samples will be in present in that buffer).

2) Procedure:

- Create a project for this program and run it. The amplitude was chosen to be A = 5000 in order to make the wavetable output audible. Hold the processor after a couple of seconds (SHIFT-F5).
- Using the keyboard shortcut, "ALT-V RT", or the menu commands View -> Graph -> Time/Frequency, open a graph-properties window. Select the starting address to be, buffer, set the sampling rate to 8 and look at the time waveform. Count the number of cycles displayed. Can you predict that number from the fact that N samples are contained in that buffer? Next right-click on the graph and select "Properties", and choose "FFT Magnitude" as the plot-type. Verify that the peak is at f = 1 kHz.

- Reset the frequency to 500 Hz. Repeat parts (a,b).

- Create a GEL file with a slider for the value of the frequency over the interval $0 \leq f \leq 1$ kHz in steps of 100 Hz. Open the slider and run the program while changing the frequency with the slider.

- Set the frequency to 30 Hz and run the program. Keep decreasing the frequency by 5 Hz at a time and determine the lowest frequency that you can hear (but, to be fair don't increase the speaker volume; that would compensate the attenuation introduced by your ears.)

- Replace the following two lines in the *isr()* function by a single call to the function *wavgen*, and repeat parts (a,b):

```
yL = (short) (A * w[q]);
q = (int) (q+c); if (q >= D) q = 0;
```

- Replace the sinusoidal table of part (f) with a square wavetable that has period 4000 and is equal to +1 for the first half of the period and −1 for the second half. Run the program with frequency f = 1 kHz and f = 200 Hz.

- Next, select the sampling rate to be fs = 96 kHz and for the sinusoid case, start with the frequency f = 8 kHz and keep increasing it by 2 kHz at a time till about 20 kHz to determine the highest frequency that you can hear—each time repeating parts (a,b).

6.2 AM Modulation

There are two wavetables to illustrate AM modulation. Figure 1 shows how one wavetable is used to generate a modulating amplitude signal, which is fed into the amplitude input of a second wavetable.

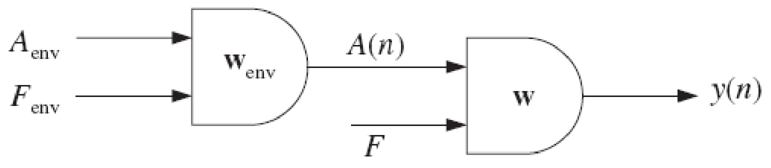


Figure 1: Wavetable implementation of AM modulation

The AM-modulated for tone signal is of the form:

$$x(t) = A(t)\sin(2\pi f t), \qquad \text{where} \quad A(t) = A_{env}\sin(2\pi f_{env} t)$$

The following program, amex.c, shows how to implement this with the function *wavgen()*. The envelope frequency is chosen to be 2 Hz and the signal frequency 200 Hz. A common sinusoidal wavetable sinusoidal buffer is used to generate both the signal and its sinusoidal envelope.

1) Code:

```
// amex.c - AM example
// ------------------------------------------------------------------------------------
#include "dsplab.h" // DSK initialization declarations and function prototypes
#include <math.h>
//#define PI 3.141592653589793
short xL, xR, yL, yR; // left and right input and output samples from/to codec
#define D 8000 // fmin = fs/D = 8000/8000 = 1 Hz
float w[D]; // wavetable buffer
short fs=8;
float A, f=0.2;
float Ae=10000, fe=0.002;
int q, qe;
float wavgen(int, float *, float, float, int *);
// ------------------------------------------------------------------------------------
void main()
{
int i;
float PI = 4*atan(1);
q=qe=0;
for (i=0; i<D; i++) w[i] = sin(2*PI*i/D); // fill sinusoidal wavetable
initialize();
sampling_rate(fs);
audio_source(LINE);
while(1);
}
// ------------------------------------------------------------------------------------
interrupt void isr()
{
float y;
// read_inputs(&xL, &xR); // inputs not used
A = wavgen(D, w, Ae, fe/fs, &qe);
y = wavgen(D, w, A, f/fs, &q);
yL = yR = (short) y;
write_outputs(yL,yR);
return;
}
```

Although the buffer is the same for the two wavetables, two different circular indices, q, q_e are used for the generation of the envelope amplitude signal and the carrier signal.

2) Procedure:

- Run and listen to this program with the initial signal frequency of f = 200 Hz and envelope frequency of fenv = 2 Hz. Repeat for f = 2000 Hz. Repeat the previous two cases with fenv = 20 Hz.

- Repeat and explain what you hear for the cases:
  f = 200 Hz, fenv = 100 Hz
  f = 200 Hz, fenv = 190 Hz
  f = 200 Hz, fenv = 200 Hz

6.3 FM Modulation:

The third program, fmex.c, illustrates FM modulation in which the frequency of a sinusoid is time varying. The generated signal is of the form:

$$x(t) = \sin[2\pi f(t)t]$$

The frequency f(t) is itself varying sinusoidally with frequency fm:

$$f(t) = f_0 + A_m \sin(2\pi f_m t)$$

Its variation is over the interval f0−Am ≤ f(t)≤ f0+Am. In this experiment, we choose the modulation depth *Am* = 0.3f_0, so that 0.7f_0 ≤ f(t)≤ 1.3f_0. The center frequency is chosen as f_0 = 500 Hz and the modulation frequency as *fm* = 1 Hz. Again two wavetables are used as shown in Fig. 2 with the first one generating *f*(*t*), which then drives the frequency input of the second generator.

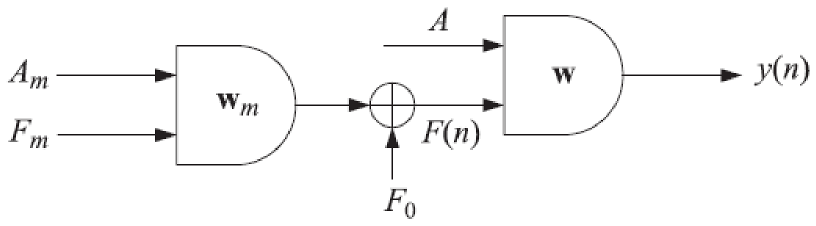


Figure 2: Wavetable implementation of FM modulation

1) Code:

```
// fmex.c - FM example
// -----------------------------------------------------------------------------------
#include "dsplab.h" // DSK initialization declarations and function prototypes
#include <math.h>
//#define PI 3.141592653589793
short xL, xR, yL, yR; // left and right input and output samples from/to codec
#define D 8000 // fmin = fs/D = 8000/8000 = 1 Hz
float w[D]; // wavetable buffer
short fs=8;
float A=5000, f=0.5;
float Am=0.3, fm=0.001;
int q, qm;
float wavgen(int, float *, float, float, int *);
// -----------------------------------------------------------------------------------
void main()
{
int i;
float PI = 4*atan(1);
q = qm = 0;
for (i=0; i<D; i++) w[i] = sin(2*PI*i/D); // load sinusoidal wavetable
//for (i=0; i<D; i++) w[i] = (i<D/2)? 1 : -1; // square wavetable
initialize();
sampling_rate(fs);
audio_source(LINE);
while(1);
}
// -----------------------------------------------------------------------------------
```

```
interrupt void isr()
{
float y, F;
// read_inputs(&xL, &xR); // inputs not used
F = (1 + wavgen(D, w, Am, fm/fs, &qm)) * f/fs; // modulated frequency
y = wavgen(D, w, A, F, &q); // FM signal
yL = yR = (short) y;
write_outputs(yL,yR);
return;
}
// --------------------------------------------------------------------------------
```

2) Procedure:

- Compile, run, and hear the program with the following three choices of the modulation depth: Am = 0.3f_0, Am = 0.8 f_0, Am = f0, Am = 0.1 f_0. Repeat these cases when the center frequency is changed to f_0 = 1000 Hz.

- Replace the sinusoidal wavetable with a square one and repeat the case f_0 = 500 Hz, Am = 0.3 f_0. You will hear a square wave whose frequency switches between a high and a low value in each second.

- Keep the square wavetable that generates the alternating frequency, but generate the signal by a sinusoidal wavetable. To do this, generate a second sinusoidal wavetable and define a circular buffer for it in *main()*. Then generate your FM modulated sinusoid using this table. The generated signal will be of the form:

$$x(t) = \sin[2\pi f(t)t], \qquad f(t) = 1 \text{ Hz square wave}$$

6.4 Ring Modulators and Tremolo:

Interesting audio effects can be obtained by feeding the audio input to the amplitude of a wavetable generator and combining the resulting output with the input, as shown in Fig. 3.

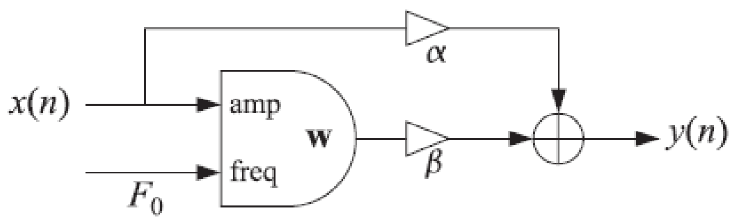


Figure 3: The Tremolo effect in audio signals

For example, for a sinusoidal generator of frequency $F_0 = f_0/fs$, we have:

$$y(n) = \alpha x(n) + \beta x(n)\cos(2\pi F_0 n) = x(n)\left[\alpha + \beta\cos(2\pi F_0 n)\right]$$

The ring modulator effect is obtained by setting $\alpha = 0$ and $\beta = 1$, so that:

$$y(n) = x(n)\cos(2\pi F_0 n)$$

whereas, the tremolo effect corresponds to $\alpha = 1$ and $\beta \neq 0$:

$$y(n) = x(n) + \beta x(n)\cos(2\pi F_0 n) = x(n)\left[1 + \beta\cos(2\pi F_0 n)\right]$$

The following ISR function implements either effect:

1) **Code:**

```
// ------------------------------------------------------------------------------------
interrupt void isr()
{
float x, y;
read_inputs(&xL, &xR);
x = (float) xL;
y = alpha * x + beta * wavgen(D, w, x, f/fs, &q);
yL = yR = (short) y;
write_outputs(yL,yR);
return;
}
// ----------------------------------------------------------------------------------
```

2) **Procedure:**

- Modify the amex.c project to implement the ring modulator/tremolo effect. Set the carrier frequency to f_0 = 400 Hz and $\alpha = \beta = 1$. Compile, run, and play a wavefile with voice in it (e.g., dsummer.)
- Experiment with higher and lower values of f_0
- Repeat first part when $\alpha = 0$ and $\beta = 1$ to hear the ring-modulator effect.

Lab Tasks:

1) Complete all the previous tasks
2) Write a code for AM modulation of the following signal: $x(t) = e^{-t} \; for \;\; 0 \leq t \leq 10$