Using Code Composer Studio

Introduction

The importance of the C language in DSP systems has grown significantly over the past few years. TI has responded by creating an efficient silicon and compiler architecture to provide efficient C performance. Additionally, TI has worked hard to provide easy-to-use software development tools.

Using these tools, all it takes is a couple of minutes to get your C code running on the 'C6000. That's the goal of this module: compile, debug, and graph a simple C sine-wave routine.

Learning Objectives

Outline

- Code Composer Studio (CCS)
- Projects
- Build Options
- Build Configurations
- Configuration Tool
- C – Data Types and Header Files
- Lab 2
Module Topics

Using Code Composer Studio.................................................................................................................. 2-1

Code Composer Studio (CCS)................................................................................................................. 2-3

Projects................................................................................................................................................... 2-6

Build Options........................................................................................................................................ 2-8
CCS Graphical Interface for Build Options......................................................................................... 2-9
Linker Build Options.............................................................................................................................. 2-11
Configuration Tool................................................................................................................................. 2-12

C – Data Types and Header Files........................................................................................................ 2-13
C Data Types......................................................................................................................................... 2-13
C Header (.h) Files................................................................................................................................. 2-14

LAB 2: Using Code Composer Studio.................................................................................................. 2-15
main.c.................................................................................................................................................. 2-17
sine.h .................................................................................................................................................. 2-18
sine.c .................................................................................................................................................. 2-19
sine.c (continued)............................................................................................................................... 2-20
sine.c (continued)............................................................................................................................... 2-21
Start CCS........................................................................................................................................... 2-22
Create the Lab2 project....................................................................................................................... 2-22
Create a CDB file................................................................................................................................. 2-23
Adding files to the project..................................................................................................................... 2-24
Examine the C Code............................................................................................................................. 2-25
Examine/Modify the Build Options...................................................................................................... 2-25
Building the program (.OUT)............................................................................................................... 2-26
Watch Variables.................................................................................................................................. 2-27
Viewing and Filling Memory.................................................................................................................. 2-27
Setting Breakpoints............................................................................................................................... 2-28
Running Code..................................................................................................................................... 2-29
Windows and Workspaces................................................................................................................... 2-30
Graphing Data....................................................................................................................................... 2-31
Shut Down and Close......................................................................................................................... 2-32

Optional Exercises............................................................................................................................... 2-33
Lab2a – Customize CCS...................................................................................................................... 2-33
Lab2b – Using GEL Scripts.................................................................................................................. 2-35
Using GEL Scripts................................................................................................................................. 2-35
Lab2c – Fixed vs Floating Point.......................................................................................................... 2-38

Optional Topics..................................................................................................................................... 2-40
Optional Topic: CCS Automation........................................................................................................... 2-40
GEL Scripting......................................................................................................................................... 2-40
Command Line Window.......................................................................................................................... 2-41
CCS Scripting....................................................................................................................................... 2-42
TCONF Scripting (Textual Configuration)............................................................................................. 2-43
The Code Composer Studio (CCS) application provides all the necessary software tools for DSP development. At the heart of CCS you’ll find the original Code Composer IDE (integrated development environment). The IDE provides a single application window in which you can perform all your code development; from entering and editing your program code, to compilation and building an executable file, and finally, to debugging your program code.

When TI developed Code Composer Studio, it added a number of capabilities to the environment. First of all, the code generation tools (compiler, assembler, and linker) were added so that you wouldn’t have to purchase them separately. Secondly, the simulator was included (only in the full version of CCS, though). Third, TI has included DSP/BIOS. DSP/BIOS is a real-time kernel consisting of three main features: a real-time, pre-emptive scheduler; real-time capture and analysis; and finally, real-time I/O.

Finally, CCS has been built around an extensible software architecture which allows third-parties to build new functionality via plug-ins. See the TI website for a listing of 3rd parties already developing for CCS.
Here’s a snapshot of the CCS screen:
A Short Review of CCS File Extensions

Using Code Composer Studio (CCS) you may not need to know all these file extension names, but we included a basic review of them for your reference:

- C and C++ use the standard .C and .CPP file extensions.
- Linear Assembly is written in a .SA file.
- You can either write standard assembly directly, or it can be created by the compiler and Assembly Optimizer. In all cases, standard assembly uses .ASM.
- Object files (.OBJ), created by the assembler, are linked together to create the DSP’s executable output (.OUT) file. The map (.MAP) file is an output report of the linker.
- The .OUT file can be loaded into your system by the debugger portion of CCS.

If you want to use your own extensions for file names, they can be redefined with code generation tool options. Please refer to the TMS320C6000 Assembly Tools Users Guide for the appropriate options.
Projects

Code Composer works with a *project* paradigm. If you’ve done code development with most any sophisticated IDE (Microsoft, Borland, etc.), you’ve no doubt run across the concept of projects.

Essentially, within CCS you create a project for each executable program you wish to create. Projects store all the information required to build the executable. For example, it lists things like: the source files, the header files, the target system’s memory-map, and program build options.

![What is a Project?](image)

The project information is stored in a .PJT file, which is created and maintained by CCS. To create a new project, you need to select the `Project: New...` menu item.
Along with the main **Project** menu, you can also manage open projects using the right-click popup menu. Either of these menus allows you to **Add Files…** to a project. Of course, you can also drag-n-drop files onto the project from Windows Explorer.

---

**Right-Click Menu**

- **Set as Active Project**
  - Keep multiple projects open

- **Add files… to project**
  - Add drag-n-drop files onto .PJT

- **Open for Editing**
  - Opens PJT with text editor

- **Configurations…**
  - Keep multiple sets of build options

- **Options…**
  - Set build options

---

There are many other project management options. In the preceding graphic we’ve listed a few of the most commonly used actions:

- If your project team builds code outside the CCS environment, you may find **Export Makefile** (and/or Source Control) useful.

- CCS now allows you to keep multiple projects open simultaneously. Use the **Set as Active Project** menu option or the project drop-down to choose which one is active.

- If you like digging below the surface, you’ll find that the .PJT file is simply an ASCII text file. **Open for Editing** opens this file within the CCS text editor.

- **Configurations…** and **Options…** are covered in detail, next.
Build Options

Project options direct the code generation tools (i.e. compiler, assembler, linker) to create code according to your system’s needs. Do you need to logically debug your system, improve performance, and/or minimize code size? Your C6000 results can be dramatically affected by compiler options.

Compiler Build Options

- Nearly one-hundred compiler options available to tune your code’s performance, size, etc.
- Following table lists most commonly used options:

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mv6700</td>
<td>Generate 'C67x code ('C62x is default)</td>
</tr>
<tr>
<td>-mv67p</td>
<td>Generate 'C672x code</td>
</tr>
<tr>
<td>-mv6400</td>
<td>Generate 'C64x code</td>
</tr>
<tr>
<td>-mv6400+</td>
<td>Generate 'C64x+ code</td>
</tr>
<tr>
<td>-fr &lt;dir&gt;</td>
<td>Directory for object/output files</td>
</tr>
<tr>
<td>-fs &lt;dir&gt;</td>
<td>Directory for assembly files</td>
</tr>
<tr>
<td>-g</td>
<td>Enables src-level symbolic debugging</td>
</tr>
<tr>
<td>-ss</td>
<td>Interlist C statements into assembly listing</td>
</tr>
<tr>
<td>-o3</td>
<td>Invoke optimizer (-o0, -o1, -o2/-o, -o3)</td>
</tr>
<tr>
<td>-k</td>
<td>Keep asm files, but don’t interlist</td>
</tr>
</tbody>
</table>

- Debug and Optimize options conflict with each other, therefore they should be not be used together

There are probably about 100 options available for the compiler alone. Usually, this is a bit intimidating to wade through. To that end, we’ve provided a condensed set of options. These few options cover about 80% of most users needs.

As you probably learned in college programming courses, you should probably follow a two-step process when creating code:

- Write your code and debug its logical correctness (without optimization).
- Next, optimize your code and verify it still performs as expected.

As demonstrated above, certain options are ideal for debugging, but others work best to create highly optimized code. When you create a new project, CCS creates two sets of build options – called Configurations: one called Debug, the other Release (you might think of as Optimize). Configurations will be explored in the next section.

Note: As with any compiler or toolset, learning the various options requires a bit of experimentation, but it pays off in the tremendous performance gains that can be achieved by the compiler.
CCS Graphical Interface for Build Options

To make it easier to choose build options, CCS provides a graphical user interface (GUI) for the various compiler options. Here’s a sample of the Debug configuration options.

- **GUI has 8 pages of options for code generation tools**
- **Default build options for a new project are shown**
- **Basic page defaults are -g -mv6700**

There is a one-to-one relationship between the items in the text box and the GUI check and drop-down box selections. Once you have mastered the various options, you’ll probably find yourself just typing in the options.
**Build Option Configurations** (Sets of Build Options)

To help make sense of the many compiler and linker options, you can create sets of build options. These sets of options are called **configurations**. TI provides two default configurations in each new project you create. For example, if you created a project called `modem.pjt`, it would contain:

<table>
<thead>
<tr>
<th>Build Option Configuration</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Debug</strong></td>
<td><code>-g -fr&quot;$(Proj_dir)\Debug&quot; -d&quot;_DEBUG&quot; -mv6700</code></td>
</tr>
<tr>
<td><strong>Release</strong></td>
<td><code>-o3 -fr&quot;$(Proj_dir)\Release&quot; -mv6700</code></td>
</tr>
</tbody>
</table>

The two main differences between the **Debug** and **Release** configurations:
- **Debug** uses the `–g` option to enable source-level debugging
- **Release** invokes the optimizer with `–o3` (and doesn’t use `–g`)

**Note:** `$\langle$Proj_dir$\rangle$` indicates the current project directory. This aids in project portability. See SPRA913 (**Portable CCS Projects**) for more information.

The following graphic summarizes the default configurations for a project called “modem”. Additionally, it shows how to:
- Select the configuration before building your project
- Add or Remove configurations from a project (Project → Configurations… menu)

Steps to edit a configuration:

1. Set it active
2. Modify build options (shown previously)
3. Save project

**Note:** The examples shown here are for a C67x DSP, hence the `–mv6700` option.
Linker Build Options

There are many linker options but these four handle all of the basic needs.

- `-o filename` specifies the output (executable) filename.
- `-m filename` creates a map file. This file reports the linker’s results.
- `-c` tells the compiler to autoinitialize your global and static variables.
- `-x` tells the compiler to exhaustively read the libraries. Without this option libraries are searched only once, and therefore backwards references may not be resolved.

### Linker Options

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-o filename</code></td>
<td>Output file name</td>
</tr>
<tr>
<td><code>-m filename</code></td>
<td>Map file name</td>
</tr>
<tr>
<td><code>-c</code></td>
<td>Auto-initialize global/static C variables</td>
</tr>
<tr>
<td><code>-x</code></td>
<td>Exhaustively read libs (resolve back ref's)</td>
</tr>
</tbody>
</table>

- By default, linker options include the `-o` option
- We recommend you add the `-m` option
- 
- $(Proj_dir)\Debug\lab.map" indicates one subfolder level below project (.pjt) location
- Run-time Autoinit (-c) tells compiler to initialize global/static variables before calling main()
- Autoinit discussed in Ch 3
Configuration Tool

The DSP/BIOS Configuration Tool (often called Config Tool or GUI Tool or GUI) creates and modifies a system file called the Configuration DataBase (.CDB). If we talk about using CDB files, we’re also talking about using the Config Tool.

The following figure shows a CDB file opened within the configuration tool:

![CDB file opened within the configuration tool](image)

The GUI (graphical user interface) simplifies system design by:

- Automatically including the appropriate runtime support libraries
- Automatically handles interrupt vectors and system reset
- Handles system memory configuration (builds CMD file)
- When a CDB file is saved, the Config Tool generates 5 additional files:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename.cdb</td>
<td>Configuration Database</td>
</tr>
<tr>
<td>filename.cfg_c.c</td>
<td>C code created by Config Tool</td>
</tr>
<tr>
<td>filename.cfg.s62</td>
<td>ASM code created by Config Tool</td>
</tr>
<tr>
<td>filename.cfg.cmd</td>
<td>Linker commands</td>
</tr>
<tr>
<td>filename.cfg.h</td>
<td>header file for *cfg_c.c</td>
</tr>
<tr>
<td>filename.cfg.h62</td>
<td>header file for *cfg.s62</td>
</tr>
</tbody>
</table>

When you add a CDB file to your project, CCS automatically adds the C and assembly (.S62) files to the project under the Generated Files folder. (You must manually add the CMD file, yourself.)

- Many of the CDB objects will be discussed in this workshop. To get all the details on this tool, though, we recommend you attend the 4-day DSP/BIOS Workshop.
C – Data Types and Header Files

C Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>ASCII</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>Binary, 2’s complement</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>Binary, 2’s complement</td>
</tr>
<tr>
<td>long</td>
<td>40</td>
<td>Binary, 2’s complement</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>Binary, 2’s complement</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>IEEE 32-bit</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>IEEE 64-bit</td>
</tr>
<tr>
<td>long double</td>
<td>64</td>
<td>IEEE 64-bit</td>
</tr>
<tr>
<td>pointers</td>
<td>32</td>
<td>Binary</td>
</tr>
</tbody>
</table>

Here are a few guidelines to keep in mind regarding C data types on the C6000:

1. Use short types for integer multiplication. As with most fixed-point DSPs, our ‘C62x devices use a 16x16 integer multiplier. If you specify an int multiply, a software function in the runtime support library will be called. (Note, the ‘C67x devices do have a 32x32 → 64-bit multiply instruction, MPYID.)

2. Use int types for counters and indexes. As we examine during the next chapter, all registers and data paths are 32-bits wide.

3. Avoid accidentally mixing long and int variables. Many compilers allocate 32-bits for both types, thus some users interchange these types. The ‘C6000 allocates longs at 40-bits to take advantage of 40-bit hardware within the CPU. If you mix types, the compiler may be forced to manage this – which will most likely cost you some performance.

   Why 40-bits? The extra 8-bits are often used to provide headroom in integer operations. Also, they can act like an 8-bit “carry bit”.

4. On ‘C67x devices, 32-bit float operations are performed in hardware. The ‘C6000 supports IEEE 32-bit floating-point math.

5. The double precision floating-point hardware supports IEEE 64-bit floating-point math.

6. Pointers, at 32-bits, can reach across the entire ‘C6000 memory-map.
C Header (.h) Files

Including Header Files in C

```c
/*
 * ======== Include files ========
*/
#include <csl.h>
#include <csl_edma.h>
#include "sine.h"
#include "edma.h"
```

1. What is #include used for?
   - It adds the contents of the header file to your C file at the point of the #include statement.

2. What do header (.h) files contain?
   - They can contain any C statements. Usually, they contain code that would otherwise need to be entered into every C file. They're a shortcut.

3. What is the difference between <.h> and “.h”?
   - Angle brackets <.h> tell the compiler to look in the specified include path.
   - Quotes “.h” indicate the file is located in the same location as the file which includes it.

Example Header Files

```c
/*
 * ======== sine.h ========
 * This file contains prototypes for all
 * functions and global datatypes
 * contained in sine.c
 */
#ifndef SINE_Obj
typedef struct {
  float freqTone;
  float freqSampRate;
  float a;
  float b;
  float y0;
  float y1;
  float y2;
  ...
} SINE_Obj;
#endif

void copyData(short *inbuf, ...);
void SINE_init(SINE_Obj *sineObj, ...);
...
```

```c
/*
 * ======== edma.h ========
 * This file contains references for all
 * functions contained in edma.c
 */
void initEdma(void);
void edmaHwi(int tcc);
extern EDMA_Handle hEdma;
```

- Header files can contain any C code to be used over and over again
- Usually a header file is paired with a C file or library object file. Essentially, the header file provides a description of the global items in the “paired” file.
- Most commonly, header files contain:
  - Function prototypes
  - Global data references, such as new type definitions
LAB 2: Using Code Composer Studio

This lab has four goals:

- Build a project using C source files.
- Load a program onto the DSK.
- Run the program and view the results.
- Use the CCS graphing feature to verify the results.

### Lab 2 – Creating/Graphing a Sine Wave

**Introduction to Code Composer Studio (CCS)**
- Create and build a project
- Examine variables, memory, code
- Run, halt, step, breakpoint your program
- Graph results in memory (to see the sine wave)

**Note:** You will find that in this lab, the code is working VERY inefficiently. Using the proper optimization techniques (later in the workshop), you will experience vast improvements in the code’s performance.
A block sine-wave generator function creates data samples which we can then graph. The block sine-wave generator function is a basic for loop that uses the following routine to generate individual sine values:

```c
float y[3] = {0.0, 0.02, 0.0};
float A = 1.9993146;
short sineGen() {
    y[0] = y[1] * A - y[2];
    y[2] = y[1];
    y[1] = y[0];
    return((short)(28000*y[0]));
}
```

The algorithm used in the workshop is similar to that shown above. It uses a monostable IIR filter to generate a sine wave.

The lab’s version of the sine-wave generator, though, provides an sine initialization function which calculates the value for A and y[1] based on the tone & sampling frequencies.

There are many ways to create sine values, we have chosen this simple IIR based model. While generating a sine wave using a table is probably more MIPs efficient, this method is more memory efficient. Also, since this function calculates each sine wave value, it gives the processor some “work” to perform.
main.c

For your convenience, we've provided a print out of the code that you will be starting with on the next few pages.

```c
/*
 * ======== main.c ========
 * This file contains all the functions for Lab2 except
 * SINE_init() and SINE_blockFill().
 */

/*
 * ======== Include files ========
 */
#include "sine.h"

/*
 * ======== Declarations ========
 */
#define BUFFSIZE 32

/*
 * ======== Prototypes ========
 */

/*
 * ======== Global Variables ========
 */
short gBuf[BUFFSIZE];
SINE_Obj sineObj;

/*
 * ======== main ========
 */
void main()
{
    SINE_init(&sineObj, 256, 8 * 1024);

    SINE_blockFill(&sineObj, gBuf, BUFFSIZE); // Fill the buffer with sine data

    while (1) { // Loop Forever
        }
    }
}
sine.h

/*
 * ======== sine.h ========
 * This file contains prototypes for all functions
 * contained in sine.c
 */
#endif
#define SINE_Init
#define SINE_blockFill
#define SINE_addPacked
#define SINE_add

void copyData(short *inbuf, short *outbuf, int length);
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampRate);
void SINE_blockFill(SINE_Obj *myObj, short *buf, int len);
void SINE_addPacked(SINE_Obj *myObj, short *inbuf, int length);
void SINE_add(SINE_Obj *myObj, short *inbuf, int length);
sine.c

// ======== sine.c ========
// The coefficient A and the three initial values
// generate a 200Hz tone (sine wave) when running
// at a sample rate of 48KHz.
//
// Even though the calculations are done in floating
// point, this function returns a short value since
// this is what's needed by a 16-bit codec (DAC).

// ======== Includes ========
#include "sine.h"
#include <std.h>
#include <math.h>

// ======== Definitions ========
#define PI 3.1415927

// ======== Prototypes ========
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampRate);
void SINE_blockFill(SINE_Obj *sineObj, short *buf, int len);
void SINE_addPacked(SINE_Obj *sineObj, short *inbuf, int length);
void SINE_add(SINE_Obj *sineObj, short *inbuf, int length);
static short sineGen(SINE_Obj *sineObj);
static float degreesToRadiansF(float d);
void copyData(short *inbuf, short *outbuf ,int length );

// ======== Globals ========

// ======== SINE_init ========
// Initializes the sine wave generation algorithm
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampRate)
{
    float rad = 0;
    if(freqTone == NULL)
        sineObj->freqTone = 200;
    else
        sineObj->freqTone = freqTone;
    if(freqSampRate == NULL)
        sineObj->freqSampRate = 48 * 1024;
    else
        sineObj->freqSampRate = freqSampRate;

    rad = sineObj->freqTone / sineObj->freqSampRate;
    rad = rad * 360.0;
    rad = degreesToRadiansF(rad);

    sineObj->a  = 2 * cosf(rad);
    sineObj->b  = -1;
    sineObj->y0 = 0;
    sineObj->y1 = sinf(rad);
    sineObj->y2 = 0;
    sineObj->count = sineObj->freqTone * sineObj->freqSampRate;
```c
sine.c (continued)

sineObj->aInitVal = sineObj->a;
sineObj->bInitVal = sineObj->b;
sineObj->y0InitVal = sineObj->y0;
sineObj->y1InitVal = sineObj->y1;
sineObj->y2InitVal = sineObj->y2;
sineObj->countInitVal = sineObj->count;
}

// ======== SINE_blockFill ========
// Generate a block of sine data using sineGen
void SINE_blockFill(SINE_Obj *sineObj, short *buf, int len)
{
    int i = 0;
    for (i = 0; i < len; i++) {
        buf[i] = sineGen(sineObj);
    }
}

// ======== SINE_addPacked ========
// add the sine wave to the indicated buffer of packed
// left/right data
// divide the sine wave signal by 8 and add it
void SINE_addPacked(SINE_Obj *sineObj, short *inbuf, int length)
{
    int i = 0;
    static short temp;
    for (i = 0; i < length; i+=2) {
        temp = sineGen(sineObj);
        inbuf[i] = (inbuf[i]) + (temp>>4);
        inbuf[i+1] = (inbuf[i+1]) + (temp>>4);
    }
}

// ======== SINE_add ========
// add the sine wave to the indicated buffer
void SINE_add(SINE_Obj *sineObj, short *inbuf, int length)
{
    int i = 0;
    short temp;
    for (i = 0; i < length; i++) {
        temp = sineGen(sineObj);
        inbuf[i] = (inbuf[i]) + (temp>>4);
    }
}
```
static short sineGen(SINE_Obj *sineObj) {
    float result; if (sineObj->count > 0) {
        sineObj->count = sineObj->count - 1;
    }
    else {
        sineObj->a = sineObj->aInitVal;
        sineObj->b = sineObj->bInitVal;
        sineObj->y0 = sineObj->y0InitVal;
        sineObj->y1 = sineObj->y1InitVal;
        sineObj->y2 = sineObj->y2InitVal;
        sineObj->count = sineObj->countInitVal;
    }

    sineObj->y0 = (sineObj->a * sineObj->y1) + (sineObj->b * sineObj->y2);
    sineObj->y2 = sineObj->y1;
    sineObj->y1 = sineObj->y0;

    // To scale full 16-bit range we would multiply y[0]
    // by 32768 using a number slightly less than this
    // (such as 28000) helps to prevent overflow.
    result = sineObj->y0 * 28000;

    // We recast the result to a short value upon returning it
    // since the D/A converter is programmed to accept 16-bit
    // signed values.
    return((short)result);
}

static float degreesToRadiansF(float d) {
    return(d * PI / 180);
}

void copyData(short *inbuf, short *outbuf, int length) {
    int i = 0;

    for (i = 0; i < length; i++) {
        outbuf[i] = inbuf[i];
    }
}
Lab 2 Procedure

Start CCS
1. Start CCS using the desktop icon

Create the Lab2 project
2. Create a new project
   Create a new project: c:\iw6000\labs\audioapp\audioapp.pjt by choosing:
   Project → New
   It should look like this:

   ![Project Creation](image)

   **Note:** Make sure that the location is correct. If you need to change it, you can either type it in or browse to it by clicking on the box next to it.

3. Verify that the new project was created correctly.
   Verify the newly created project is open in CCS by clicking on the + sign next to the Projects folder in the Project View window. Click again on the + sign next to audioapp.pjt. If you don’t see the new project, notify your instructor.

67
If using the C6713 DSK, this should say TMS320C67XX
Create a CDB file

As mentioned during the discussion, configuration database files (*.CDB), created by the Config Tool, control a range of CCS capabilities. In this lab, the CDB file will be used to automatically create the reset vector and perform memory management.


Create a new CDB file (DSP/BIOS Configuration…) as shown:

When the dialog box appears, select the dsk6416.cdb (or dsk6713.cdb) template and click OK.

Hint: In some TI classrooms you may see two or more tabs of CDB templates; e.g. TMS62xx, TMS54xx, etc. If you experience this, just choose the ‘C6x tab.
5. **Save your CDB file.**

   File → Save As

   C:\iw6000\labs\audioapp\audioapp.cdb

Then, close the CDB Config Tool.

The CDB files shown in the aforementioned dialog box are called “seed” CDB files. CDB files are used to configure a great many objects. Of these, quite a few are board specific; e.g. type of DSP, MHz, etc. To make life easier, TI provides a seed file with all boards it ships.

**Adding files to the project**

You can add files to a project in one of three ways:

- Project → Add Files to Project
- Right-click the project icon in the Project Explorer window and select *Add files*…
- Drag and drop files from Windows Explorer onto the project icon

6. **Add files to your project.**

   Using one of these methods, add the following files from C:\iw6000\labs\audioapp\ to your project:

   - main.c
   - audioapp.cdb
   - sine.c

**Note:** You may need to change the "Files of Type" box at the bottom of the Open Dialog Box to see all of the files. We recommend that you choose "All Files" so that you can add everything at once.

Click the + sign next to Source in the Project Window to make sure your source (*.c) files were added successfully. Also, click the + sign next to DSP/BIOS Config to make sure the .CDB file is displayed.
Examine the C Code

7. Open and inspect main.c

Open main.c (double-click on the file in the Project window) and inspect its contents. You’ll notice that we’ve set up a buffer in memory that is of length 32. This buffer will hold the values that are generated by the sine wave generator routine. Look at the main() routine. We simply initialize the sine generator, call the sine function (fill a buffer with sine values), and then go into an infinite while loop.

8. Examine the sine routine

Open the sine.c file. Here, we have coded an IIR filter using some initial conditions that are provided by the call to SINE_init(). We pass in the tone that we want to generate and the rate that we will sample that tone at. We also pass in a SINE_Obj structure that is used by the sine generator to keep track of the information that it needs. To actually generate sine values, we call the SINE_blockFill() function. Each time this routine is called, it will fill up the buffer with 32 new sine data values. Toward the end of this lab, we will graph this buffer to see if it looks like a sine wave. Some other functions, used in later labs, are also located in this file.

Examine/Modify the Build Options

9. Review the Build Options

Select:

Project → Build Options

Using the settings under the 4 tabs shown, you can control the compilation and linking of your project to any degree you like. For example, you can choose various levels of code optimization, but leave optimization off (None) for now. Note: if you’re using the C6713DSK, the Target Version will be (-mv6700):

Notice that the text box at the top of the Build Options window reflects all of the currently selected options. Click OK to close the Build Options dialog when you’re finished.
**Building the program (.OUT)**

Now that all the files have been added to our project, it’s time to create the executable output program (that is, the .OUT file). Our executable file will be named: `audioapp.out`.

10. **Select Debug configuration.**

   The build configurations are shown to the right of the project name near the upper LH corner of CCS. For easy debugging, use the `Debug` configuration; this should be the default. Verify that `Debug` is in the `Project Configurations` drop-down box.

11. **Build the program.**

   There are two ways to build (compile and link) your program:

   - Use the `REBUILD ALL` toolbar icon:

   ![REBUILD ALL icon]

   - Select **Project → Rebuild All**

   Choose one of the above methods and build your program. The `Build Output` window appears in the lower part of the CCS window. Note the build progress information. If you don’t see “0 Errors, 0 Warnings, 0 Remarks”, please ask your instructor for help.

12. **Load your program to the DSK.**

   Since you previously enabled the `Program Load after Build` option, the program should automatically have been downloaded to the DSK by the CCS debugger. If your program did not load, select:

   **File → Load Program**

   and browse to the Debug folder `c:\iw6000\labs\audioapp\debug\` and select `audioapp.out`.

13. **Run to Main (if not there already)**

   In lab 1, we also enabled the option to automatically go to main when a program is loaded. So your program should be sitting at `main( )` right now. **If you are not, and the program has been loaded, run to the main function using:**

   **Debug → Go Main**

   The debugger should run past the system initialization code, until `main( )` is reached. Since main is in `main.c`, this file should appear in CCS’s main work area. Many initialization steps occur between reset and your main program. These issues will be explained and investigated later in this workshop.
Watch Variables

Now that we have the program built and loaded, let's take a closer look at it using the tools provided by CCS.

14. Add gBuf to the Watch window.

Select and highlight the variable gBuf in the main.c window. Right-click on gBuf and choose Add to Watch Window.

Note: the value shown for gBuf may differ from that shown below.

After adding a variable, the Watch window automatically opens and gBuf is added to it. Alternatively, you could have opened the watch window, selected gBuf, and drag-n-dropped it onto the Watch 1 window.

Click on the + sign next to gBuf to see the individual elements of the array.

Note: At some point, if the Watch window shows an error “unknown identifier” for a variable, don’t worry, it's probably due to the variable’s scope. Local variables do not exist (and don’t have a value) until their function is called. If requested, Code Composer will add local variables to the Watch window, but will indicate they aren’t valid until the appropriate function is reached.

Viewing and Filling Memory

15. View the memory contents at the address gBuf.

Another way to view values in memory is to use a memory window. Select View → Memory

and type in the following:

- Title = gBuf
- Address = gBuf
- Q-Value = 0
- Format = 16-Bit Hex-TI Style

Click OK and resize the window so that you can see your code and the buffer. Because we have just come out of reset and this memory area was not initialized, you should see random garbage at this location. Let’s initialize it….
16. **Record the address of the gBuf array.**

   There are many ways to find this address. Two of them are:
   - The address shown for the +gBuf value in the *Watch Window*; or
   - The address associated with gBuf in the *Memory View* window

   Address of gBuf: ___________________________________________________________

17. **Initialize the gBuf array to zero.**

   Select:
   
   **Edit → Memory → Fill**

   and fill in the following:
   - Address = gBuf
   - Length = 16
   - Fill Pattern = 0

   Click OK. The buffer was 32 16-bit values in length (they were defined as “shorts” in the C file). The fill memory function fills integer, or 32-bit, locations. So, we only need to fill 16 32-bit locations to zero out the 32x16 array. Keep this in mind when you want to initialize an area of memory. You might end up stomping on something you shouldn’t. In a few moments, we’ll create a GEL file that does this fill automatically.

**Setting Breakpoints**

18. **Set a break point.**

   Set a break point on the while loop in main(). Breakpoints can be set in 3 different ways. Choose the one you like best and set the breakpoint:

   - Place the cursor on the end brace of the while() loop and click on the:
   - Right-click on the line with the end brace and choose *Toggle Breakpoint*
   - Double-click in the grey area next to the end brace (as shown below):

   ![Image of code with break point set on the end brace]
Running Code

19. Run your code.

Run the code up to the breakpoint. There are 3 different ways to cause CCS to run your code:

- Use toolbar icon: 
- Select: Debug → Run
- Press F5

The processor will halt at the breakpoint that you’ve set. Notice that this line is inside an infinite while loop. Notice that the watch window changes to show the new values of gBuff[]. You may have to click on the + sign next to buffer to see the values. Code Composer allows you to collapse and expand aggregate data types (structures, arrays, etc.).

The values that are red are the values that have changed with the last update, which occurred when your code hit the breakpoint.
**Windows and Workspaces**

20. **Save your Workspace**

As long as a window is not maximized in CCS, it can be moved around to any location you prefer. Windows can float or be docked. Select the watch window, right-click on the upper portion, and select **Float In Main Window**. Then, move it around. Try docking it again.

When you have the windows exactly where you want them, save your workspace by choosing:

File → Workspace → Save Workspace As

Pick a filename and save it in any location you prefer (typically your /audioapp directory).

---

**Note:** The workspace includes the current open project. So, when you retrieve the workspace, it will retrieve the project. If you don’t wish to save the project info with the workspace, close the project before saving your workspace.

If you want to retrieve a previously saved workspace, select:

File → Workspace → Load Workspace
**Graphing Data**

21. Graph your sine data.

The watch window is a great way to view data in CCS. But, can you tell if this is really a sine wave? Wouldn’t it be better to see this data graphed? Well, CCS allows us to do this. Select:

View → Graph → Time/Frequency

Modify the following values:

- Graph Title: gBuf
- Start Address: gBuf
- Acquisition Buffer Size: 32
- Display Data Size: 32
- DSP Data Type: 16-bit signed integer
- Sampling Rate: 8000

Click OK when finished.

Your graph should look something like this:
22. Other graphing features

CCS supports many different graphing features: time frequency, FFT magnitude, dual-time, constellation, etc. The sine wave that we generated was a 256Hz wave sampled at 8KHz.
Let’s see if we can use the FFT magnitude plot to see the fundamental frequency of the sine wave.

Right click on the graphical display of gBuf and select Properties. Change the display type to FFT Magnitude and click OK. You can now see the 256Hz wave. It should look something like this:

![FFT Magnitude Graph](image)

23. Remove Breakpoints.

Clear any breakpoints you set in the lab. You can use two different methods:

- Debug → Breakpoints → Delete All
- Use the toolbar icon:

24. Close the project and CCS.

Select:

Project → Close

Save changes if necessary and close Code Composer.

25. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp\*.* TO c:\iw6000\labs\lab2

Using Windows Explorer, open up a window to c:\iw6000\labs. Right-click on the audioapp folder and drag it to an open spot in the window. Click copy here. Rename the “copy of audioapp to” lab2. You will do this at the end of every lab. You also might want to leave the window open to c:\iw6000\labs for future saves of your work.

You’re Done with the main lab. Please inform your facilitator before moving on to the optional labs
Optional Exercises

If you still have some more time, give these simple exercises a try.

- Lab 2a – Customize CCS
- Lab 2b – Using GEL Scripts
- Lab 2c – Fixed vs. Float

Lab2a – Customize CCS

Add Custom Keyboard Assignment

While most CCS commands are available via hotkeys, you may find yourself wanting to modify CCS to better suit your personal needs. For example, to restart the processor, the default hotkey(s) are:

Debugger → Restart

CCS lets you remap many of these functions. Let’s try remapping Restart.

1. Start CCS if it isn’t already open.
2. Open the CCS customization dialog.

Option → Customize...
3. Choose the Keyboard tab in the customize dialog box.
4. Scroll down in the Commands list box to find Debugger → Restart and select it.
5. Click the Add button.

When asked to, “Press new shortcut key”, press:

F4

We already checked and this one isn’t assigned within CCS, by default.
6. Click OK twice to close the dialog boxes.
7. From now on, to Restart and Run the CPU, all you need to do is push F4 then F5.
Optional Exercises

Customize your Workspace

You may not find the default workspace for CCS as convenient as you’d like. If that’s the case, you can modify as needed.

8. **Close CCS if it’s open, and then open CCS.**
   
   This is forces CCS back to its default states (i.e. no breakpoints, profiling, etc.).

9. **Move the toolbars around as you’d like them to be.**
   
   For example, you may want to close the BIOS and PBC toolbars and then move the Watch toolbar upwards so that you free up another ½ inch of screen space.

10. **If you want the Project and File open dialogs to default to a specific path, you need to open a project or file from that path.**

11. Make sure you close any project or file from the previous step.

12. **Save the current workspace.**

   \[\text{File} \rightarrow \text{Workspace} \rightarrow \text{Save Workspace As...}\]

   Save this file to a location you can remember. For example, you might want to save it to:

   \[\text{c:}\backslash\text{iw6000}\backslash\text{labs}\]

13. **Close CCS.**

14. **Change the properties of the CCS desktop icon.**

   \[\text{Right-click on the CCS desktop icon}\]
   
   \[\text{Add your workspace path to the Target, as shown below:}\]

   ![CCS 2 ('C6000) Properties](image)

   This should be the path and name of your workspace.

15. **Open up CCS and verify it worked.**
Lab2b – Using GEL Scripts

GEL stands for General Extension Language, a fancy name for a scripting tool. You can use GEL scripts to automate processes as you see necessary. We’ll be using a few of them in the lab in just a few minutes….

Using GEL Scripts

When debugging, you often need to fill memory with a known value prior to building and running some new code. Instead of constantly using the menu commands, let’s create a GEL (General Extension Language) file that automates the process. GEL files can be used to execute a string of commands that the user specifies. They are quite handy.

1. Start CCS and open your project (lab2.pjt) and load the program (lab.out), if they’re not already open and loaded.

2. Create a GEL file (GEL files are just text files)

   File → New → Source File

3. Save the GEL file

   Save this file in the lab2 folder. Pick any name you want that ends in *.gel.

   File → Save

   We chose the name mygel.gel.
4. **Create a new menu item**

   In the new gel file, let’s create a new menu item (that will appear in CCS menu “GEL”) called “My GEL Functions”. Type the following into the file:

   ```
   menuitem "My GEL Functions";
   ```

   You can access all of the pre-defined GEL commands by accessing:

   ![Help → Contents](image)

   Select the Index tab and type the word “GEL”.

5. **Create a submenu item to clear our arrays**

   The `menuitem` command that we used in the previous step will place the title “My GEL Functions” under the GEL menu in CCS. When you select this menu item, we want to be able to select different operations. Submenu items are created with the `hotmenu` command.

   Enter the following into your GEL file to create a submenu item to clear the memory array:

   (Don’t forget the semicolon – as with C, it’s important!)

   ```
   hotmenu ClearArray()
   {
     GEL_MemoryFill(gBuf, 0, 16, 0x0);
   }
   ```

   The MemoryFill command requires the following info:
   - Address
   - Type of memory (data memory = 0)
   - Length (# of words)
   - Memory fill pattern.

   This example will fill our array (gBuf) with zeros. For more info on GEL and GEL commands, please refer to the CCS help file.

6. **Add a second menu item to fill the array**

   In this example, we want to ask the user to enter a value to write to each location in memory. Rather than using the `hotmenu` command, the `dialog` command allows us to query the user.

   Enter the following:

   ```
   dialog FillArrays(fillVal "Fill Array with:"
   {
     GEL_MemoryFill(gBuf, 0, 16, fillVal);
   }
   ```

7. **Save then Load your new GEL file**

   To use a GEL file, it must be loaded into CCS. When loaded, it shows up in the CCS Explorer window in the GEL folder.

   ![File → Save](image)

   ![File → Load GEL](image) and select your GEL file
8. **Before trying our GEL scripts, let’s show the gBuf array in Memory window.**

Without looking at the arrays, it will be hard to see the effect of our scripts. Let’s open a Memory window to view **gBuf**.

```
View → Memory...
```

- **Title:** gBuf
- **Address:** gBuf
- **Q-Value:** 0
- **Format:** 16-bit hex – TI style

A couple notes about memory windows:
- C Style adds 0x in front of the number, TI Style doesn’t.
- Select the Format based on the data type you are interested in viewing. This will make it easier to ‘see’ your data.

9. **Now, try the two GEL functions.**

```
GEL → My GEL Functions → ClearArray
GEL → My GEL Functions → FillArray
```

You can actually use this GEL script throughout the rest of the workshop. It is a very handy tool. Feel free to add or delete commands from your new GEL file as you do the labs.

10. **Review loaded GEL files.**

Within the CCS Explorer window (on the left), locate and expand the GEL files folder. CCS lists all loaded GEL files here.

**Hint:** If you modify a loaded GEL file, before you can use the modifications you must reload it. The easiest way to reload a GEL file:

1. Right-click the GEL file in the CCS Project Explorer window
2. Pick **Reload** from the right-click popup menu
Lab2c – Fixed vs Floating Point

We included a functioning integer sinewave routine for comparison to the float routine used throughout the workshop. Notice the additional effort required make integer math routines work correctly. This extra work is required so that the 16-bit integer values do not overflow and cause data corruption.

The method used to solve overflow in this application is often called Q-math. Maybe a better name for it is fractional, fixed-point math. The beauty of fractions is that when multiplied together, their value gets smaller. Hence the result is always bounded (i.e. no overflow).

The problem with integer math is not confined to TI DSPs (or DSPs in general), rather it is a side affect between the fact that integer numbers get bigger when add or multiply them and that the C language provides no means of handling overflow for signed numbers. In fact, the C language leaves signed math that overflows undefined – every compiler writer can handle it however they want (so much for portability).

The dynamic range of floating-point variables sure makes life easier. It’s why many folks choose floating-point to decrease their engineering time (and get to market more quickly). Of course, this is why the C6713 is so popular – as it’s designed to do floating-point math in hardware.

We have provided a project for you to compare different versions of sineGen:

- Standard fixed-point math
- Q-math (fractional, fixed-point)
- Floating-point math

You will find LAB2c_6416.PJT or LAB2c_6713.PJT already built in the LAB2c folder:

C:\iw6000\labs\lab2c\n
Try running the project and comparing all three results in three different graphs. To simplify setting up the graph windows, try using one of the provided workspaces: C6416.wks or C6713.wks located in C:\iw6000\labs\lab2c\.
Lab Debrief

Lab 2 Debrief

1. What differences are there in Lab2 between the C6713 and C6416 solutions?
2. What do we need CCS Setup for?
3. Did you find the “clearArrays” GEL menu command useful?
Optional Topics

Optional Topic: CCS Automation

As evidenced by the optional lab exercise, CCS ships provides scripting/automation tools. They are mentioned here to make you aware of their presence. To explore them further, please examine the online documentation.

**GEL Scripting**

Notice the GEL folder in the Project View window. You can load/unload GEL scripts by right-clicking this window.

GEL syntax is very C-like. Notice that QuickTest() calls LED_cycle(), defined earlier in the file. (This happens to be a C6711 DSK GEL script.)

You can add items to the GEL menu. An example is shown in the above graphic.

Finally, a GEL file can be loaded upon starting CCS. The startup GEL script is specified using the CCS Setup application.
Optional Topics

**Command Line Window**

Provides a convenient way to type in CCS commands, rather than using the pull-down menus.

---

**Command Window**

Command Window Plug-in, Version 1.24  (Mar 7 20
Copyright (c) 1986-2000 Texas Instruments
TMS320C67xx Simulator
Enter "HELP" for information
Board Version 00.00.00
Target Silicon Version 00.00.00
Device Driver Version 02.80.00

Some frequently used commands:

- help
- dlog <filename>,a
- dlog close
- alias ...
- take <filename.txt>
- load <filename.out>
- reload
- reset
- restart
- ba <label>
- wa <label>
- run <cond>
- run <label>
- go <label>
- step <number>
- cstep <number>
- halt

For those of you ‘ol timers, who remember the old command line debugging tools, you can use the same commands you’ve used for years.

The Command Window is available inside CCS under Tools → Command Window.
**CCS Scripting**

CCS Scripting is a CCS plug-in. After installing CCS on your PC, you should use the *Update Advisor* feature (available from the Help menu) to download and add the CCS Scripting plug-in.

| Hint: You may find other useful tools, application notes, and plug-ins available via the CCS Update Advisor.

CCS scripting provides a method of controlling the CCS debugger from another scripting language. Any Microsoft COM (i.e. OLE) compliant language should be able to use the CCS Scripting library, but *VB Script* and *Perl* are the two languages for which examples are provided.

The graphic below is an example of a VB Script using CCS Scripting:

![VB Script Example](image)

Among other things, CCS Scripting is very useful for testing purposes. For example, if you have a number of test vectors you would like to run against your system, you can use CCS Scripting to automate this process. Your script could then:

- Build
- Run
- Capture data, memory values, benchmarks
- And compare the results against what you expect (or hope)
- Over and over again …

At this time, the CCS Scripting Plug-in (v1.2) only ships with C5000 based examples. For your convenience, we have written and included some C6000 based examples along with the workshop lab files.
CCS now provides a textual scripting method for creating and editing CDB files.

**TCONF Scripting (Textual Configuration)**

TCONF Scripting (CDB vs. TCF)

- Tconf Script (.tcf)
  
  ```
  /* load platform */
  utils.loadPlatform("ti.platforms.dsk6416");
  config.board("dsk6416").cpu("cpu0").clockOscillator = 600.0;
  /* make all prog objects JavaScript global vars */
  utils.getProgObjs(prog);
  
  /* Create Memory Object */
  var myMem = MEM.create("myMem");
  myMem.base = 0x00000000;
  myMem.len = 0x00100000;
  myMem.space = "data";
  
  /* generate cfg files (and CDB file) */
  prog.gen();
  ```

- Textual way to create and configure CDB files
- Runs on both PC and Unix
- Create #include type files (.tci)
- More flexible than Config Tool

Some users find ‘writing code’ preferable to using the Graphical User Interface (GUI) of the Configuration Tool. This is especially true for users who build their code in the Unix environment, as there is no Unix version of the GUI.
*** we’re not sure why this page is blank – please inform your instructor ***