A QUICK START TO TEXAS INSTRUMENTS TMS 320C6713 DSK

Rıfat Benveniste, Beril Sırmaçek, Cem Ünsalan

21.07.2010

This booklet is an introduction to TI TMS 320C6713 DSK through laboratory experiments. These experiments are covered on one semester at Yeditepe University in the Digital Signal Processing course. The main aim is getting familiar with TI TMS 320C6713 DSK. More advanced books on this board can be obtained from the reference list. Screen shot images and tables are taken from TI CCS.

Table of Contents

1. Introduction

In this chapter, we give a brief introduction to DSP processors by comparing them with microcontrollers and microprocessors. We also introduce the TI TMS 320C6713 DSK.

1.1 What is a Microprocessor?

Microprocessor is a silicon chip, designed to perform arithmetic and logic operations through its programs. Typical microprocessor operations include adding, subtracting, comparing two numbers, and fetching numbers from one area to another. These operations are the result of a set of instructions that are part of the microprocessor design. Typical layout of a microprocessor is as in Figure 1.1

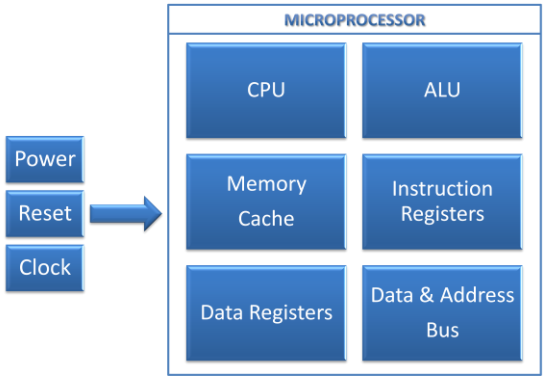


Figure 1.1 Typical layout of a microprocessor.

In this layout, the Central Processing Unit (CPU) is the basic logical structure that accomplishes instructions given by the program. Arithmetic Logic Unit (ALU) is the logical structure that accomplishes arithmetic operations like addition and subtraction. Memory is the structure that the data and the program are stored in. Common microprocessors do not have very large memory space inside the chip. Therefore, the main data and program is stored outside the microprocessor. Instruction registers are small memory blocks. They handle processes that the CPU can execute. The program sets the instruction registers in an order to execute each process required one at a time. The size of the instruction register defines the bit processing capability of the microprocessor. For example if the size of the register is 32-bit, then it is a 32-bit processor. Data registers are small memory blocks used by the CPU to store the data. Data and address busses are the highways that the data and instructions flow over. They also connect the CPU and the external peripheral units.

Microprocessors need regulated DC power supply in order to work. Typical processors work with 5V DC supply. As the silicon technology improves, these devices started working with lower power ratings. Therefore, nowadays there are processors working with 3.3V, 1.8V or lower DC voltage values.

Clock is the synchronization pulse of the processor. Instructions are executed in one or more clock pulses. Clock also provides the synchronization between the processing units of the processor. The speed of the processor is related to the clock pulse. The processor speeds are categorized in Million Instructions per Second (MIPS). This means how many instructions are processed in a second.

Reset circuitry is the most important part of a microprocessor system. If the microprocessor is locked in an unexpected infinite loop, the only way to recover the system is to apply reset to the processor. Reset brings the processor to the initial state that is defined in the reset vector inside processor's memory.

1.2 What is a Microcontroller?

Microprocessors can only perform data manipulation and computation. To interact with the outside world, there should be some peripherals connected to the microprocessor. Microcontrollers are silicon chips that contain both the microprocessor and peripherals. Typical layout of a microcontroller is as in Figure 1.2.

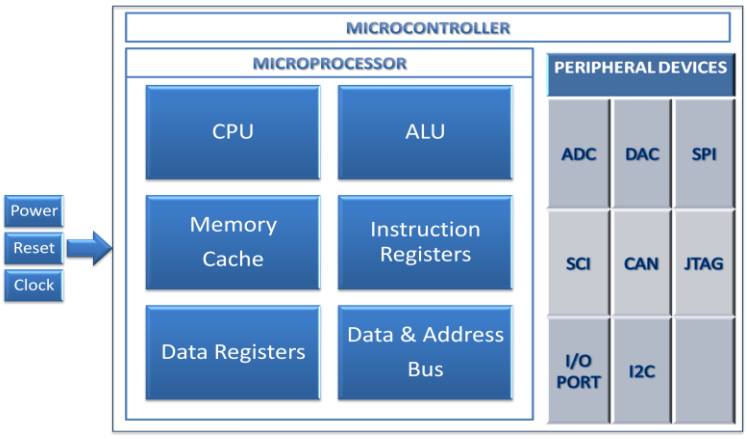


Figure 1.2 Typical layout of a microcontroller.

Microcontrollers may contain all or some of the peripherals according to their usage area. For example, to process an audio or a video signal, the microcontroller with an Analog to

Digital Converter (ADC) and Digital to Analog Converter (DAC) is needed. Serial Peripheral Interface (SPI), RS232 Serial Communication Interface (SCI), I2C (two wire communication protocol) or JTAG (A fast real-time communication protocol) interfaces will be needed to communicate with another processor or computer. The setup and usage of these peripheral devices are arranged by the control, status, and data registers. For example, to define the sampling rate of the ADC, the necessary bits in the ADC control register are set.

1.3 What is a Digital Signal Processor (DSP)?

Digital Signal Processor (DSP) is a microcontroller designed specifically for signal processing applications. This is achieved as follows. Commonly used operations in signal processing applications are convolution, filtering, and frequency-time domain conversions. These operations need recursive multiplication and additions. In other words, they need multiply and accumulate (MAC) operations. Standard microprocessors execute the multiplication operation as a recursive addition operation. This means for a standard microprocessor, the MAC operation is processed by excessive number of addition operations. This takes time. However, DSPs contain special MAC units that can execute the same operation in a single machine cycle. For example, a 150 MIPS DSP can process approximately 32 million data samples per second. For a standard 150 MIPS microprocessor, this reduces to 2 million data samples per second.

Like microcontrollers, DSPs are equipped with different peripheral devices according to their usage area. The DSP we are using throughout the book, TMS320C6713, does not contain any ADC or DAC but it contains SPI and I2C interfaces. Therefore, its development kit, TMS320C6713 DSK, contains an AIC23 codec chip working as an ADC and DAC interface. It communicates with the DSP over the SPI. Typical layout of a DSP is in Figure 1.3.

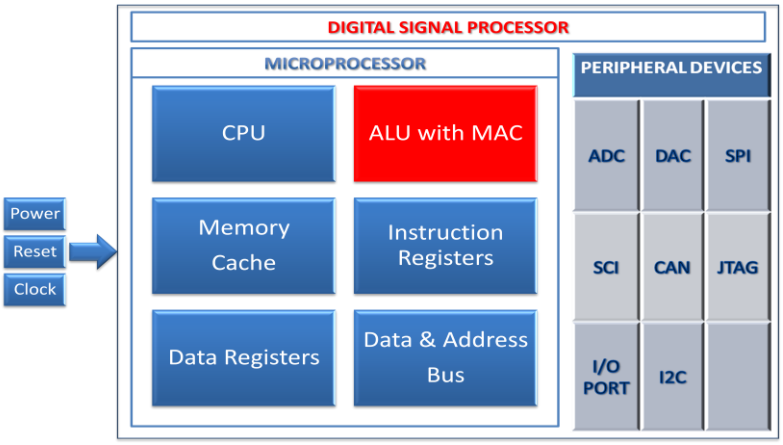


Figure 1.3 Typical layout of a DSP.

1.4 TMS320C6713 DSP Features

In this book, we focus on Texas Instrument TMS320C6713 DSP. It has the following properties:

- 32 bit instruction cycle.
- Typical 225 MHz, up to 300 MHz clock frequency.
- 2400 MIPS / 1800 MFLOPS instruction processing.
- 4xALU and 2x32-Bit MAC functional unit.
- 4KB Level1 program cache, 4KB Level1 data cache, 256KB Level2 data cache.
- 32 bit external memory interface up to 512 MB addressable external memory space.
- 2xI2C ports, 2xMCBSP (SPI) ports, 2x32-Bit timers.

The functional block diagram of TMS320C6713 DSP is given in Figure 1.4.

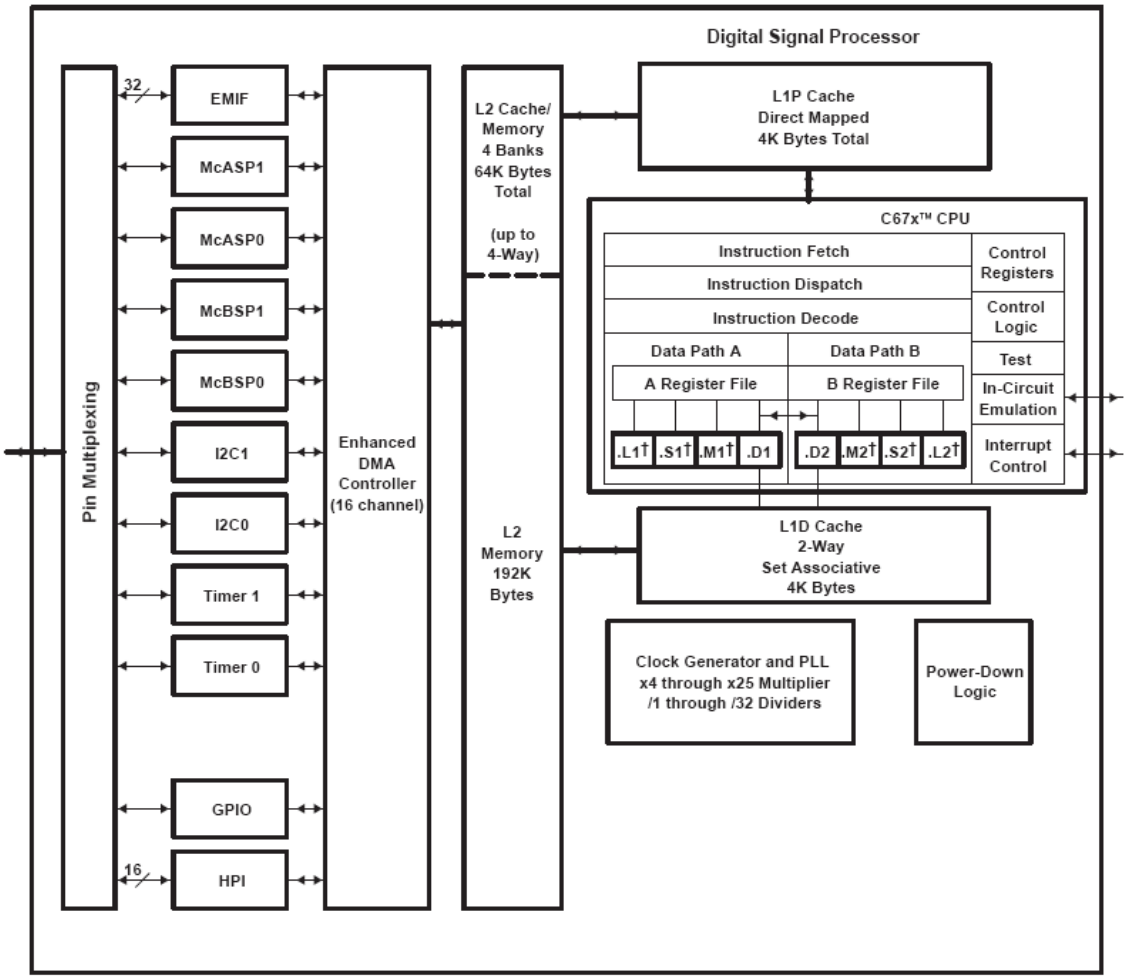


Figure 1.4 The functional block diagram of TMS320C6713 DSP.

1.5 TMS320C6713 Development Kit Features

We will not use the DSP processor alone. Instead, we will use the development kit for the mentioned DSP. This kit has the following properties:

- A Texas Instruments TMS320C6713 DSP operating at 225 MHz.
- An AIC23 stereo codec.
- 16 MB synchronous DRAM.
- 512 KB non-volatile Flash memory (256 KB usable in default configuration).
- 4 user accessible LEDs and DIP switches
- Software board configuration through registers implemented in Complex Programmable Logic Device (CPLD).
- Configurable boot options.
- Standard expansion connectors for daughter card usage.
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator.
- Single voltage power supply (+5V).

The functional block diagram of the software development kit TMS320C6713 DSK is given in Figure 1.5. We will call it as C6713 DSK from now on.

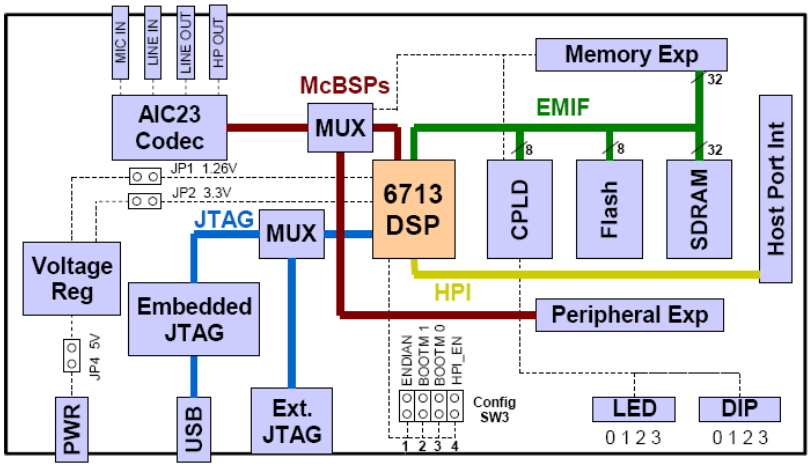


Figure 1.5 Functional block diagram of the software development kit C6713 DSK.

1.6 The Programming Language and the Number System

We use the C programming language to program the C6713 DSK. Therefore, it is utmost important to have a basic C programming knowledge. Although it is possible to code the same card with assembly language, it is not advisable to use it at beginner's level.

One of the most confusing parts of C programming the C6713 DSK is the number system. TMS320C6000 DSP has several data representations. We provide their properties in Table 1.1. This table is obtained from "TMS320C6000 Optimizing Compiler v 7.0 User's Guide". Please refer to this document for more and detailed information.

Table 1-1. TMS320C6000 C/C++ Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	8 bits	ASCII	-128	127
unsigned char	8 bits	ASCII	0	255
short	16 bits	2s complement	-32 768	32 767
unsigned short	16 bits	Binary	0	65 535
int, signed int	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
long, signed long	40 bits	2s complement	-549 755 813 888	549 755 813 887
unsigned long	40 bits	Binary	0	1 099 511 627 775
long long, signed long long	64 bits	2s complement	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum	32 bits	2s complement	-2 147 483 648	2 147 483 647
float	32 bits	IEEE 32-bit	1.175 494e-38	3.40 282 346e+38
double	64 bits	IEEE 64-bit	2.22 507 385e-308	1.79 769 313e+308
long double	64 bits	IEEE 64-bit	2.22 507 385e-308	1.79 769 313e+308
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

Exercises

1. Write a C code to calculate the first 20 elements of the Fibonacci series. First, form the algorithmic flowchart. Then, implement it.
2. What does "little Endian" and "big Endian" mean for TMS320C6000?
3. What does "pragma" and "extern" mean for TMS320C6000?

2. Code Composer Studio V3.3

The coding platform for TI's DSPs is the Code Composer Studio (CCS). In this chapter, we explore CCS V3.3 properties.

2.1 Basic Setup

Code Composer Studio (CCS) is the programming, building, and debugging interface of Texas Instruments DSPs. CCS communicates with the C6713 DSK board over embedded JTAG interface and can exchange real-time data with the DSK board.

Before starting the CCS V3.3, we have to select the platform that the CCS will use from the setup interface. To run the setup interface, click on the CCS Setup icon (similar to Figure 2.1) on your desktop.

Figure 2.1 CCS Setup shortcut icon.

2.1.1 Simulator Mode

To use the CCS V3.3 in the C6713 simulator mode, select “C6713 Device Cycle Accurate Simulator” from the board selection list. It should be added and seen on the left side of the setup screen as in Figure 2.2.

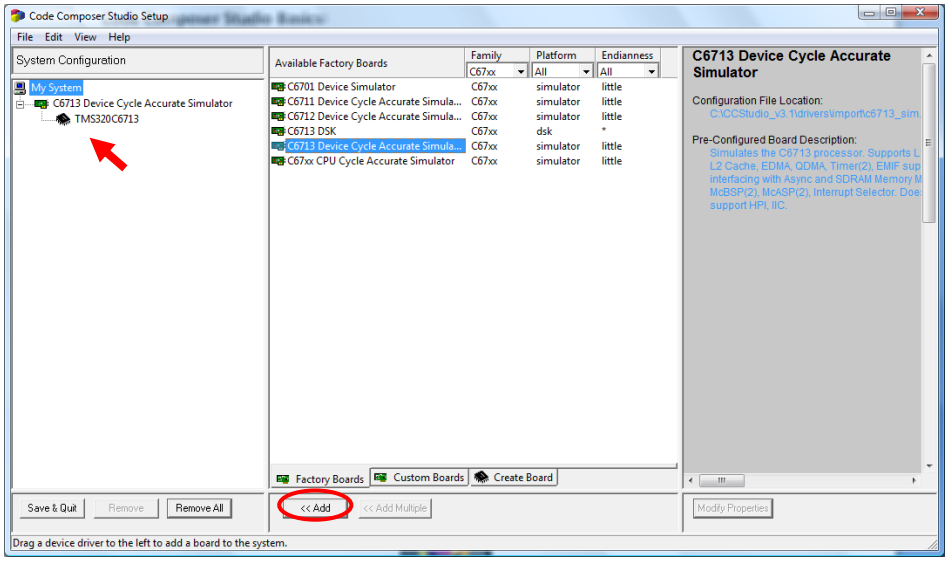


Figure 2.2 CCS V3.3 Setup Screen.

To adjust memory properties, click on the “TMS320C6713” icon as shown in Figure 2.2 with the red arrow and select “Properties”. The “Detect Reserved Memory Access” property should be set to “No” as in Figure 2.3.

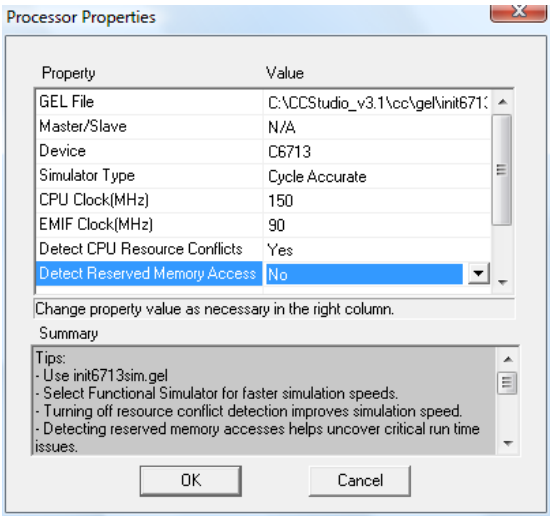


Figure 2.3 Processor Properties.

Press “OK” and return to the setup screen. Press “Save & Quit”. CCS is ready to run in the C6713 simulator mode.

2.1.2 C6713 DSK Board

If you will use the CCS V3.3 to code the C6713 DSK board, then you should select the “C6713 DSK” from the board list of the setup screen. Press the “Add” button. You should see the “C6713 DSK” on the left side of the setup screen as in Figure 2.4. Press “Save & Quit”. CCS V3.3 is now ready to program and run the C6713 DSK board.

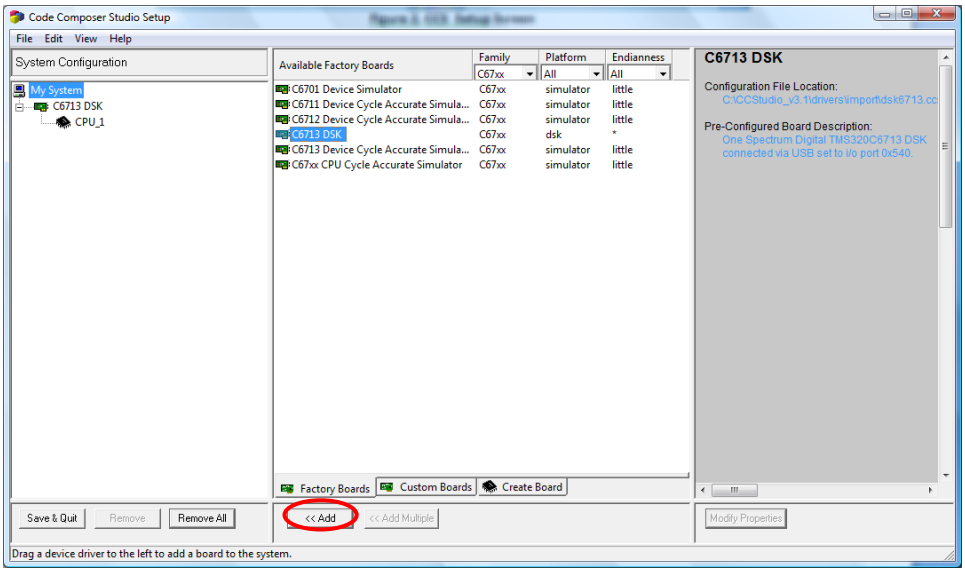


Figure 2.4 C6713 DSK selection.

To start the CCS, click on the CCS icon (similar to Figure 2.5) on your desktop with administrative privileges.

Figure 2.5 CCS shortcut icon.

2.2 Creating a New Project under CCS V3.3

You should press "project" and "new" to create a new project in CCS V3.3. To name the project, enter the "Project Name" tab as in Figure 2.6. As you click on "Finish", a blank project will be created.

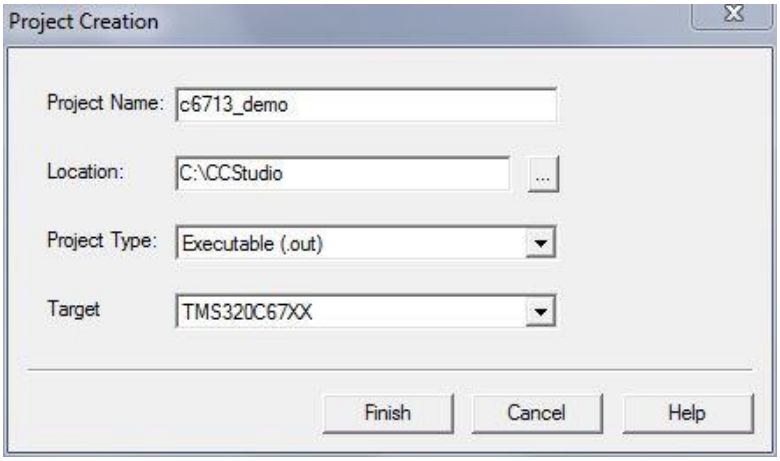


Figure 2.6 Project creation example.

2.2.1 Bios Configuration

To run the C6713 DSK board, you should adjust basic configurations in correct order. These configurations cover the memory map, interrupt handling, I/O controls, and etc. To manage these configurations, a real time operating system called DSP/BIOS is used. In the project, the operating system and the program (to run) are embedded together to the DSP. DSP/BIOS handles necessary operations on the background of the running program.

DSP/BIOS is setup using the configuration file “xxx.cdb”. To open a new DSP/BIOS configuration file, press “File→New→DSP/BIOS Configuration...” and select the “dsk6713.cdb” as in Figure 2.7.

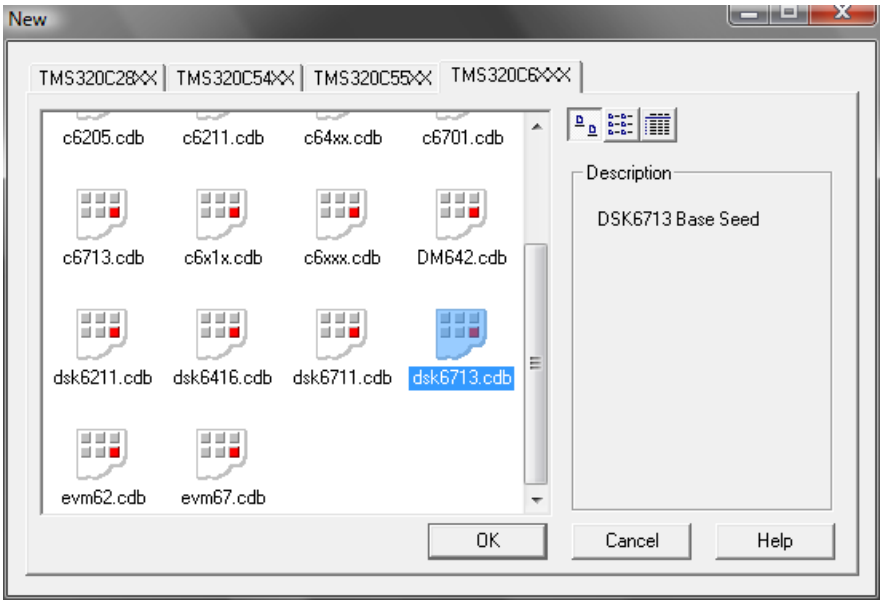


Figure 2.7 DSP/BIOS configuration.

If you will use the CCS V3.3 in the simulator mode, right click on the "RTDX-Real-Time Data Exchange Settings" as in Figure 2.8 and select its properties. Also, change the RTDX Mode to the “Simulator” as in Figure 2.9 to work in the simulator mode.

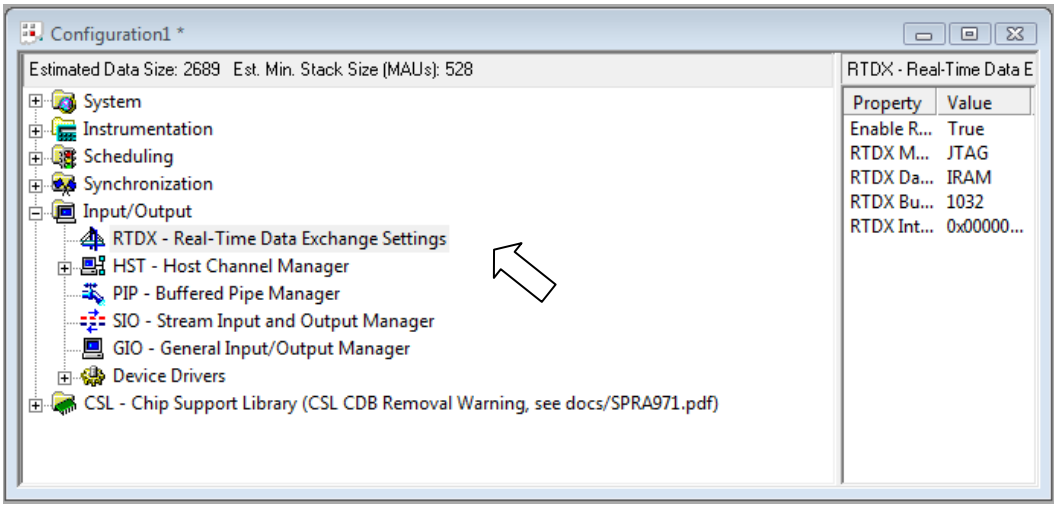


Figure 2.8 RTDX-Real-Time Data Exchange settings.

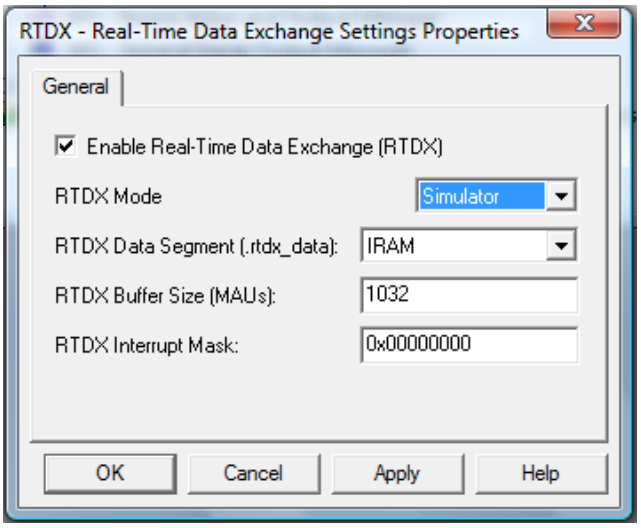


Figure 2.9 Selecting the simulator mode.

You should save the configuration file by "File→ Save: c6713_demo.cdb". Now, a new configuration file is created. However, it is not added to the project yet. It can be added to the project by "Project→ Add Files to Project..." and selecting the "c6713_demo.cdb" file as in Figure 2.10.

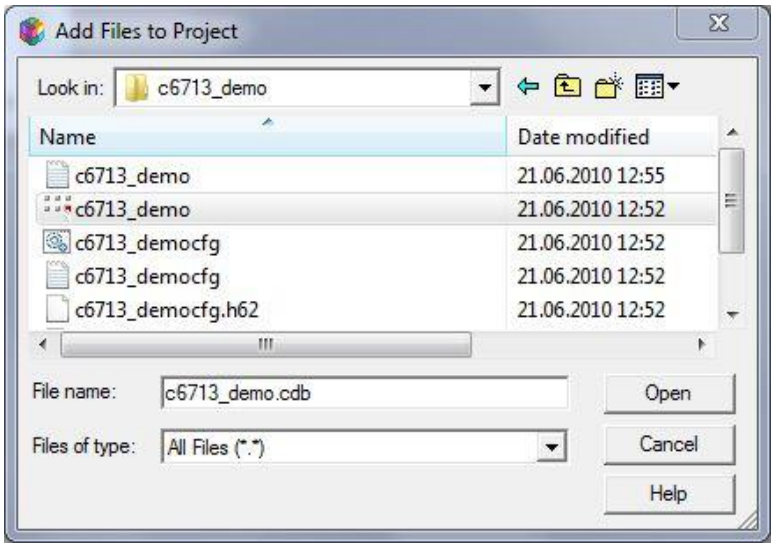


Figure 2.10 Adding cbd files to the project.

In the project tree, the configuration files that are automatically created by DSP/BIOS should be seen as in Figure 2.11.

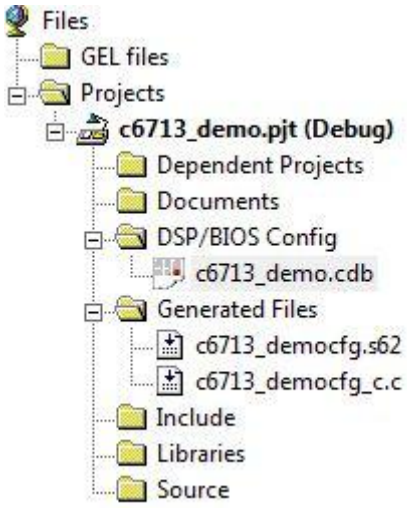


Figure 2.11 Project layout.

2.2.2 Creating the Source Code

To create a source code, you should follow the "File→ New→ Source File" steps. There is a simple source code in Figure 2.12. You should save this code using "File→ Save c6713_demo.c".

```
void main (void)
{
    int a[5]={1,2,3,4,5};
    int b[5]={1,2,3,4,5};
    int c[5];
    int i;

    for(i=0;i<5;i++)
    {
        c[i]=a[i]+b[i];
    }

}
```

Figure 2.12 The first project source code.

Again, you should add this file to the project by "Project→ Add Files to Project...". It should be seen in the project tree as in Figure 2.13.

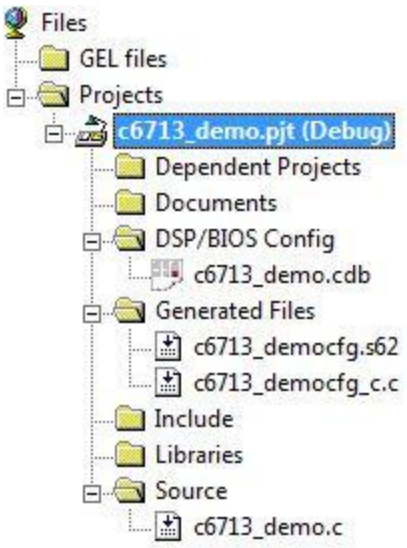


Figure 2.13 Source file added to the project.

2.3 Running the Source Code

Your first code is ready to be built and run. To build the code, follow steps "Project→Build". If there are no errors in the code, you should see a message similar to Figure 2.14.

```
Build Complete,
  0 Errors, 2 Warnings, 0 Remarks.
```

Figure 2.14 Successful code building.

To run the code, follow the steps:

i. File→ Load Program ...: Select the “experiment2_demo.out” under debug folder.
ii. Debug→ Reset Cpu
iii. Debug→ Restart
iv. Debug→ Go Main
v. Debug→ Run

2.3.1 Inserting a Break Point

To insert a break point in the code, you should move the cursor to the desired point. Click on the icon on top of the screen as in Figure 2.15. This will add a breakpoint there.

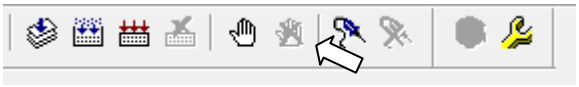


Figure 2.15 Inserting a breakpoint.

2.3.2 Adding a Watch Window

To see or change the values in variables, you can use the watch window. To add a variable to the watch window, right-click on the variable and select “Add to watch window” as in Figure 2.16.

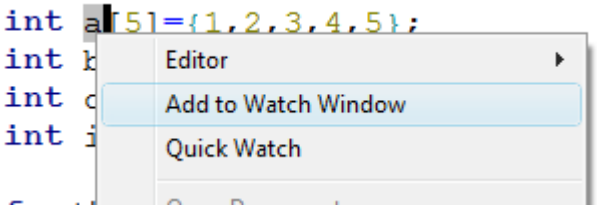


Figure 2.16 Adding a watch window.

You can see the values and changes in the watch window at the bottom right of the CCS V3.3 screen as in Figure 2.17.

Name	Value	Type	Radix
a	0x0000F034	int[5]	hex
[0]	1	int	dec
[1]	2	int	dec
[2]	3	int	dec
[3]	4	int	dec
[4]	5	int	dec

Figure 2.17 Observing and changing values in the watch window.

2.3.3 Plots

To see an array in a graph, go to "View→ Graph→ Time/Frequency...". On the pop up window (as in Figure 2.18), fill:

i. **Start Address:** Name of the array (to be plotted).
ii. **Acquisition Buffer Size:** Size of the array.
iii. **Display Data Size:** Size of the data to be displayed on the graph.
iv. **DSP Data Type:** Type of the array (integer or float).
v. **Sampling Rate:** Sampling rate of the signal in the array.

For example, to observe the array 'c' in the above demo code, adjust the parameters as in Figure 2.18. The resulting graph will be as in Figure 2.19.

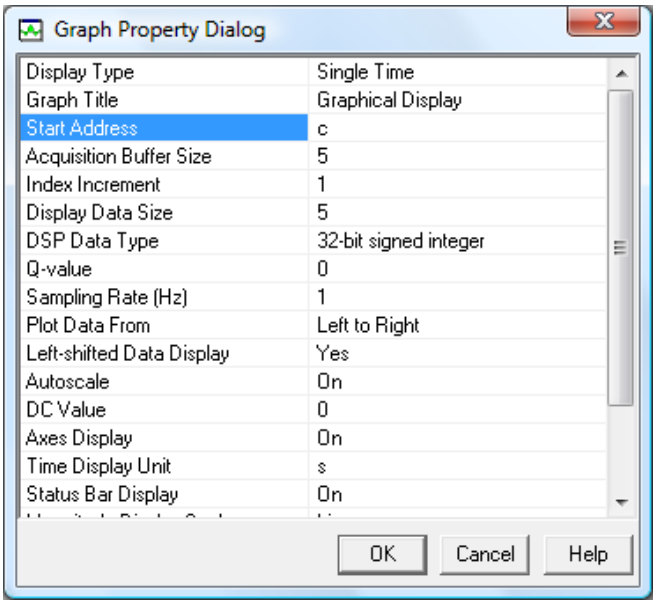


Figure 2.18 Graph setup.

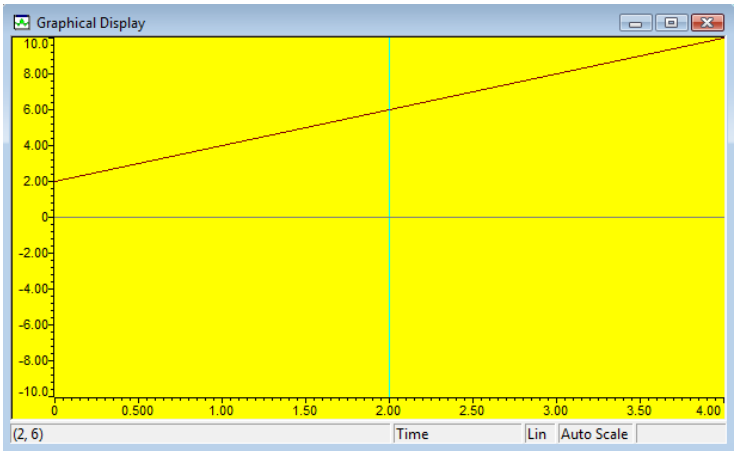


Figure 2.19 Graph output.

2.3.4 Images

Similar to plotting an array, to see an image under CCS V3.3 you should go to "View→ Graph→ image". Fill in the form in the pop up window as in Figure 2.20. In this figure, a hypothetical 256x256 grayscale image is assumed.

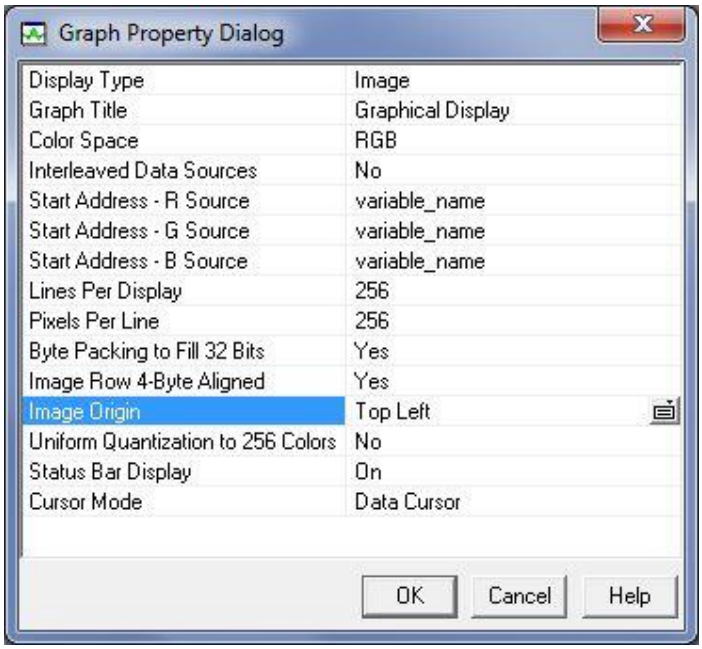


Figure 2.20 Graph setup for images.

2.3.5 Saving the Data

To save the array in a data file, you should go to "File→ Data→ Save". Fill in the form as in Figure 2.21. In this form, you can select the format for saving the data.

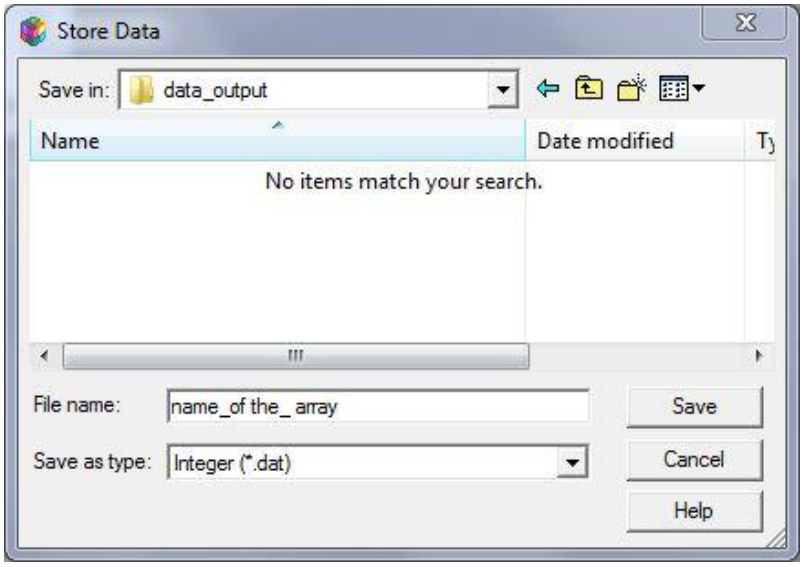


Figure 2.21 Saving the data-file properties.

As you press "save" a new window opens as in Figure 2.22. In this window, enter the “name of the array” to the “Address” field and enter the “length of the array” to the “length” field. CCS V3.3 will save the data in a “.dat” file. This file can be opened by other programs like Matlab.

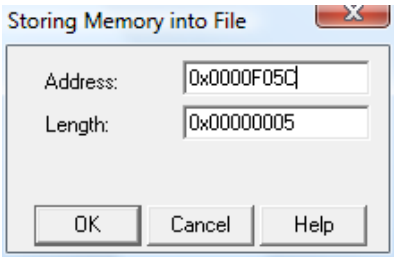


Figure 2.22 Saving the data-variable properties.

2.4 CCS GEL Scripts

Code composer studio can be customized by using General Extension Language (GEL) Scripts. GEL is an interpretive language that enables writing functions to configure the Integrated Development Environment (IDE) and access the target processor. Below is a sample script file. We will use it after designing filters.

```
menuitem "Gain Slider";
slider FilterGain1(1, 4, 1, 1, volume1)
{
 /* initialize the target variable with the Parameters passed by the slider object. */
gain1 = volume1;
}
slider FilterGain2(1, 4, 1, 1, volume2)
{
 /* initialize the target variable with the Parameters passed by the slider object. */
gain2 = volume2;
}
slider FilterGain3(1, 4, 1, 1, volume3)
{
 /* initialize the target variable with the Parameters passed by the slider object. */
gain3 = volume3;
}
```

To load a GEL file, select “File > Load GEL...” as in Figure 2.23. Load the “volume_slider.gel” file.

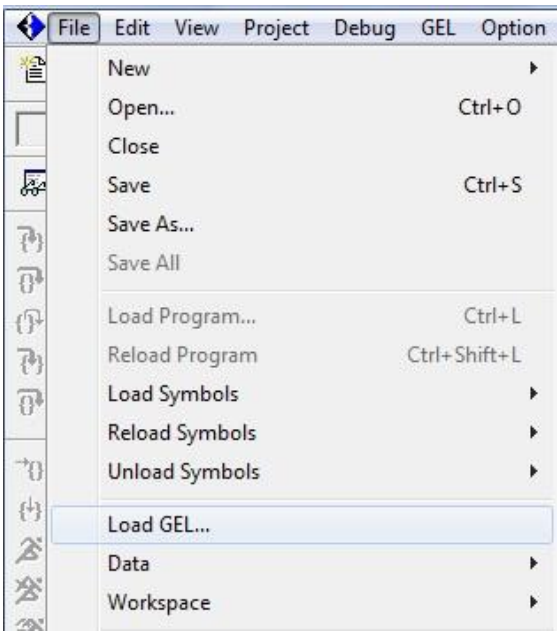


Figure 2.23 Loading the Gel file.

To open the volume sliders select “GEL> Gain Slider > FilterGain1”. If three sliders are loaded, they should be as in Figure 2.24.

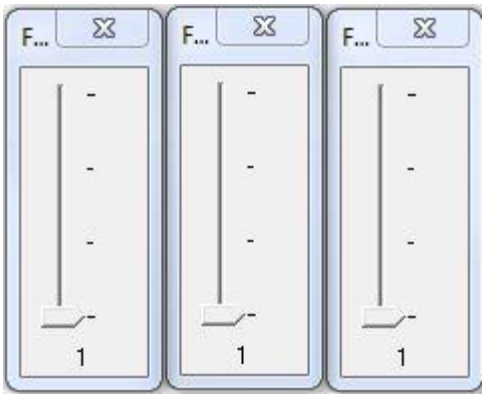


Figure 2.24 Loaded sliders by the Gel file.

2.5 Working with DIP Switches and LEDs

C6713 DSK has four user accessible DIP switches and LEDs. To reach these peripheral devices, you should include two header files “dsk6713_dip.h” and "dsk6713_led.h" to your code. For the CCS V3.3 you should apply the following procedure to locate these files. Select “Project→Build options→compiler”. Select the "processor" under the compiler tab. Include "C:\code composer studio setup folder\C6000\dsk6713\include" to search path, as in Figure 2.25.

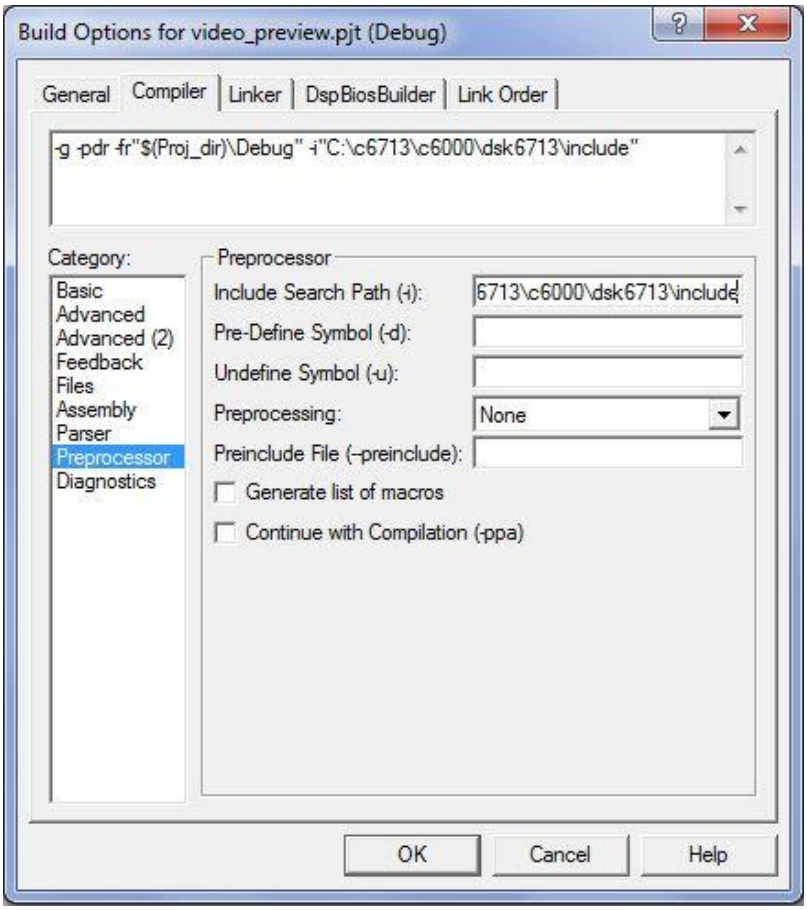


Figure 2.25 Including the search path.

You should also add the "dsk6713bsl.lib" file which is under "C:\code composer studio setup folder \C6000\dsk6713\lib" by “Add Files to Project” option as in Figure 2.26.

Figure 2.26 Adding the dsk6713bsl.lib file to the project.

Now, we are ready to reach DIP switches and LEDs on the C6713 DSK. The following code is an example of DIP switch and LED application.

```
/*********************************************************/
/* Read the status of the DIP switches and control LEDs*/
/*********************************************************/
/* All header files starting with dsk6713 belong to the board support library */
#include<dsk6713.h> /* General functions */
#include<dsk6713_dip.h> /* Functions for DIP */
#include<dsk6713_led.h> /* Functions for LED */
void main()
{
DSK6713_init();
DSK6713_DIP_init();/*Initialize DIP switches */
DSK6713_LED_init(); /*Initialize LEDs*/
while(1){
 /*DSK6713_DIP_get('switch number');  Retrieve the DIP switch value */
if(DSK6713_DIP_get(1)==1){
/* DSK6713_LED_on('LED number');   Set the LED */
  DSK6713_LED_on(1);
}else{
/* DSK6713_LED_off('LED number');   Clear the LED */
DSK6713_LED_off(1);
}
}
}
```

Exercises

1. Create an empty project named as "exercise21.pjt" in CCS V3.3 (in the simulation mode).
    a. Create a new DSP/BIOS configuration file. Set the RTDX to simulator mode. Save your configuration file to your project folder. Add this file to your project.

b. Create a new C source code file. Your code should calculate and save the first 10 elements of the Fibonacci series in an array. Save and add this file to your project folder.
c. Build and Run your project. Observe the graph of the Fibonacci array by using the graph property of CCS V3.3.
d. Save the Fibonacci array in a data file called “Result.dat”.

2. Repeat Exercise 1 using the DSP board in real time.

3. Create an empty project named as “exercise22.pjt” in CCS V3.3.
a. Include all necessary files to your project.
b. Create a new DSP/BIOS configuration file. Save your configuration file to your project folder. Add this file to your project. (Note: we will use the DSP in real time. Therefore, **do not set** the RTDX to the simulator mode).
c. Create a new C source code file. This code should set the LED on the C6713 DSK board corresponding to the switch being pressed. Add this file to your project.

3. Code Composer Studio V4.1

In this chapter, we explore the CCS V4.1 properties. We follow the same layout as in the previous chapter. To note here, we will use CCS V4.1 in the following chapters.

3.1 Creating a New Project under CCS

You should press "File", "New", and "CCS Project" to create a new project under CCS V4.1 as shown in Figure 3.1.

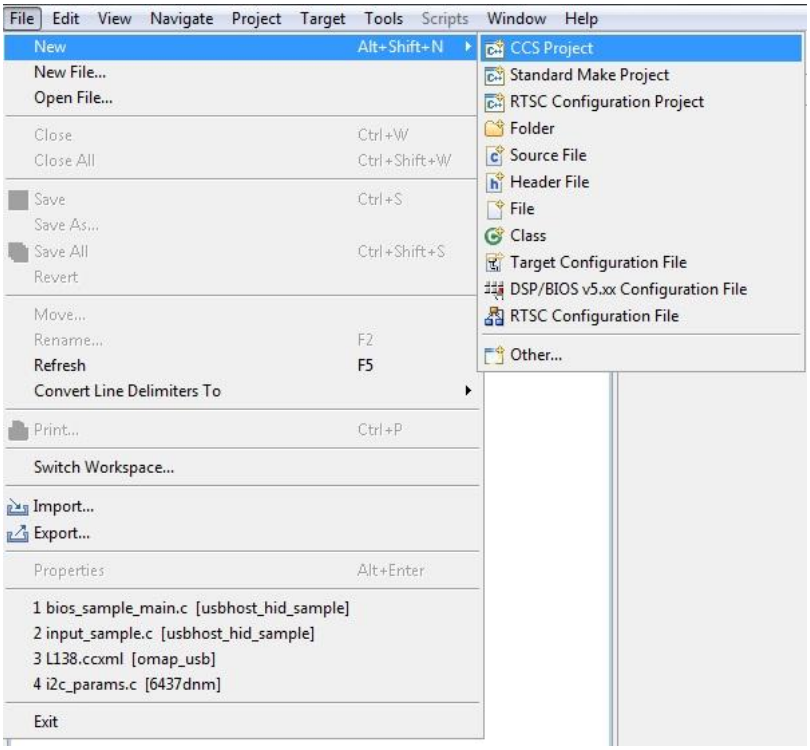


Figure 3.1 Creating a new project under CCS V4.1.

To name the project, enter the "Project Name" tab as in Figure 3.2.

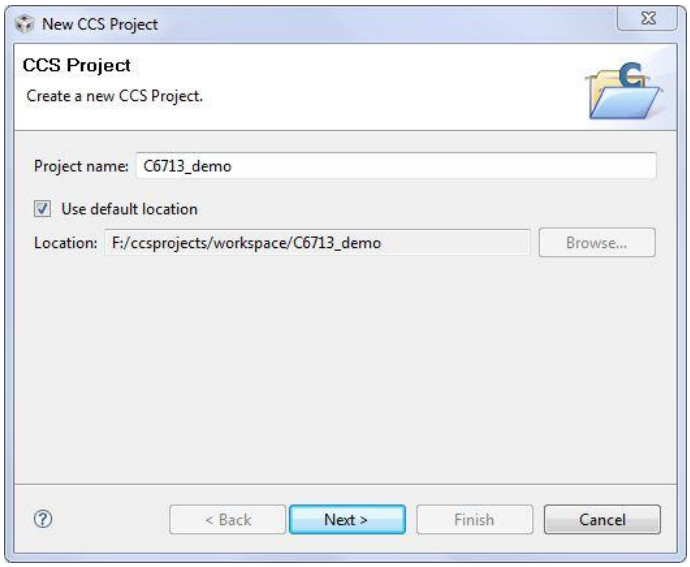


Figure 3.2 Defining the project name.

Click "Next", to select the project type as C6000 and select "debug" and "release" configurations as in Figure 3.3. Click "Next".

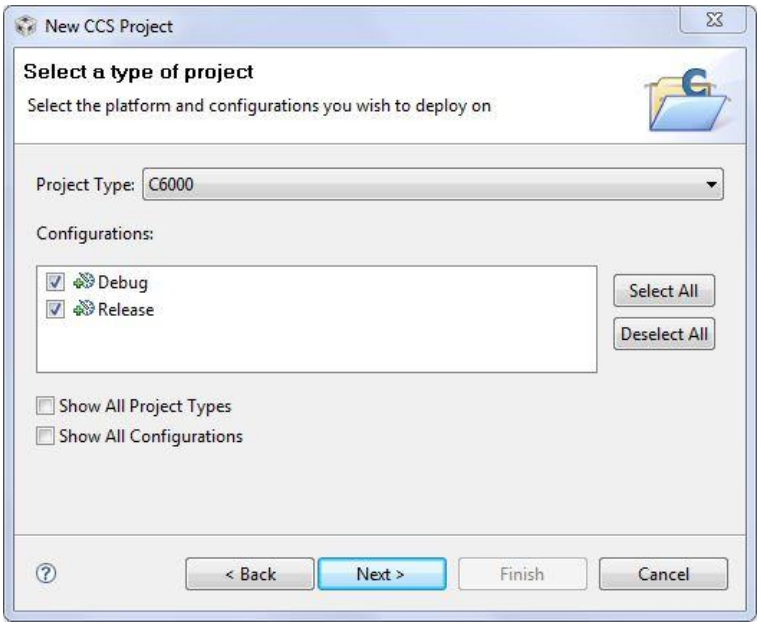


Figure 3.3 Selecting the project type.

If you have a previous project, you can use it as a reference as in Figure 3.4. Otherwise, click "Next" to continue.

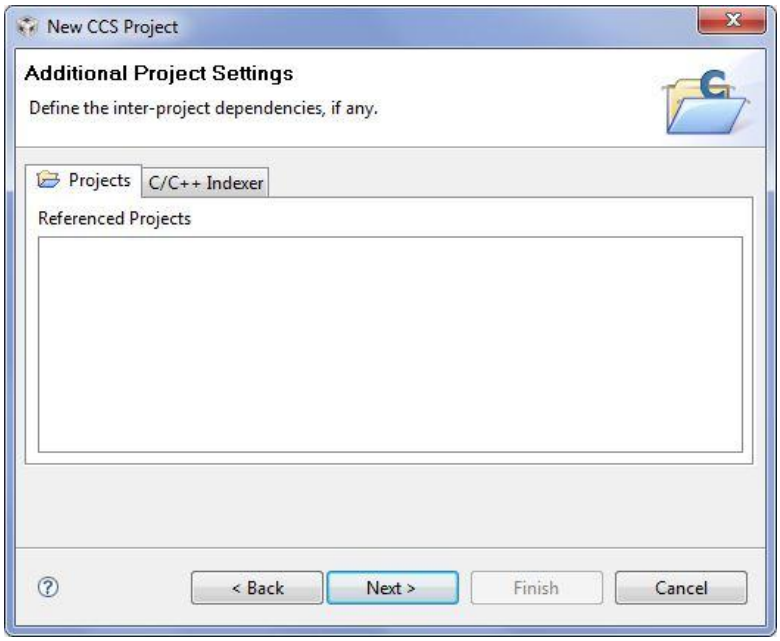


Figure 3.4 Selecting the reference project.

To finalize the project creation, select the "Output type" as "executable"; "Device Variant" as "Generic devices, Generic C67xx device"; "Device Endianness" as "little"; "Code Generation tools" as "TIv6.1.12"; "Runtime Support Library" as "automatic"; "Use DSP/BIOS v5.xx" as "5.41.02.14". Click "Finish" as in Figure 3.5.

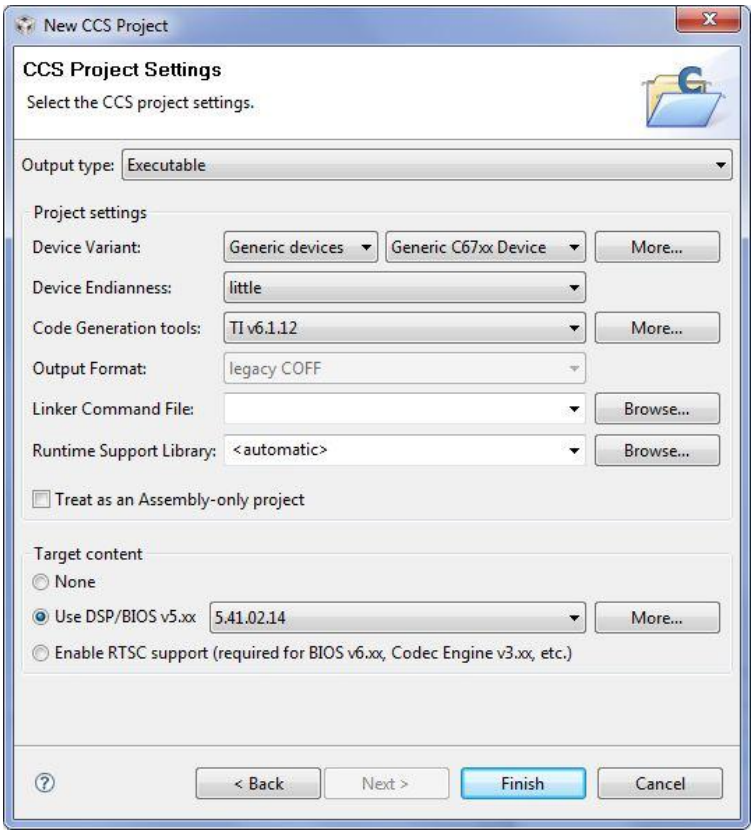


Figure 3.5 Final project settings.

You should see the project tree in the CCS V4.1 workspace window as in Figure 3.6.

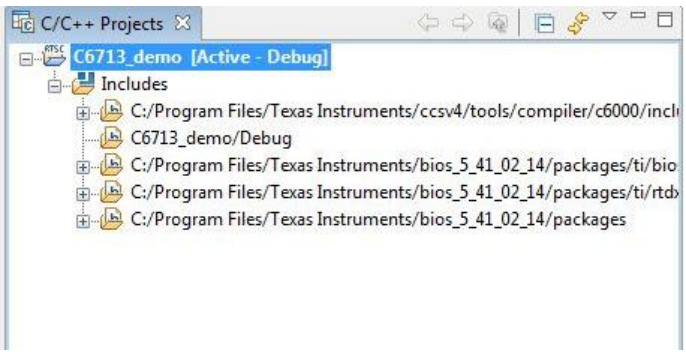


Figure 3.6 Project tree in CCS V4.1.

3.1.1 Creating a Target Configuration File

To create a target configuration file for the project, right click on the name of your project on the project tree. Select "New→Target Configuration File" as in Figure 3.7.

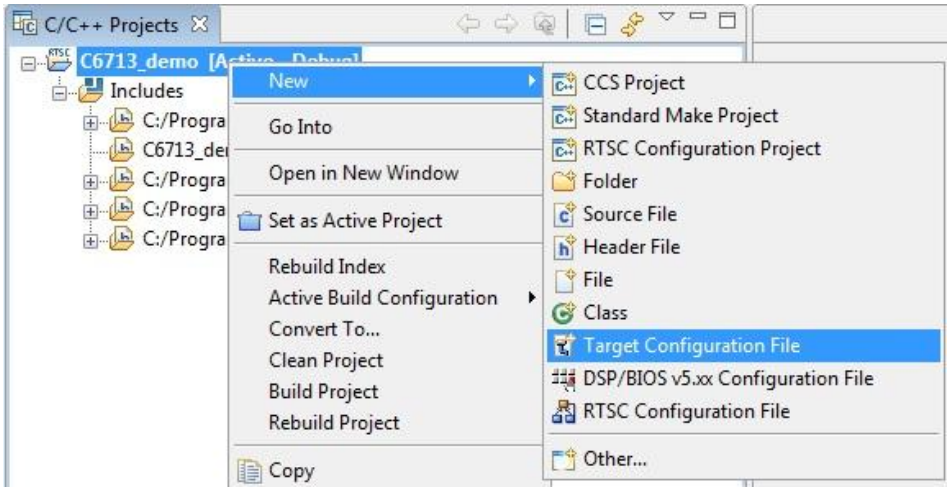


Figure 3.7 Creating a target configuration file.

On the configuration window, enter the name of your configuration file with an extension “xxx.ccxml” as in Figure 3.8. Click on the "Finish" button.

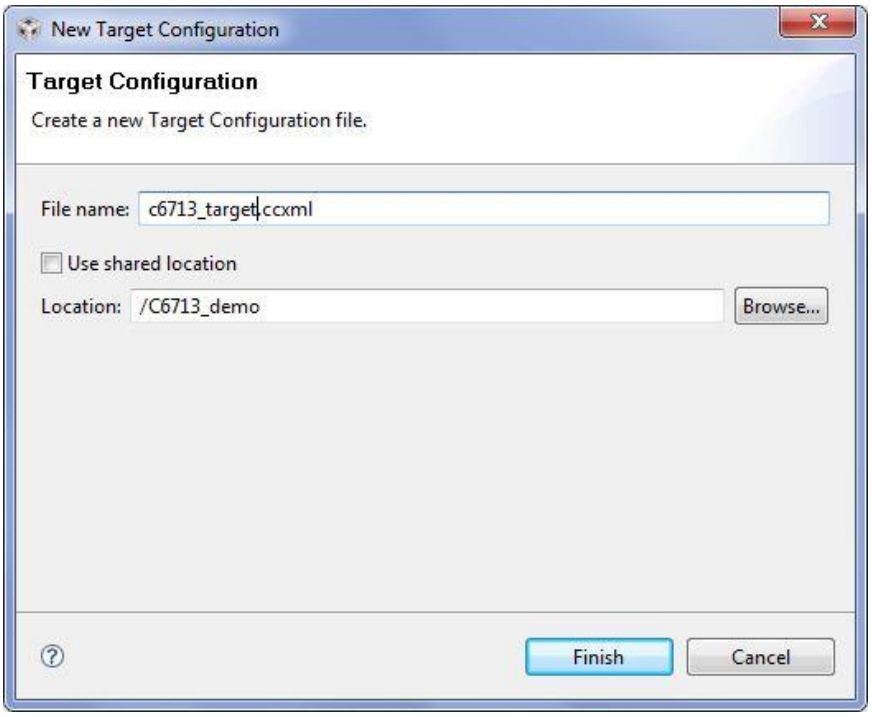


Figure 3.8 Entering the name of the configuration file.

Now, select "Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator" on the connection tab. Select "DSK6713" as the device and click save on the right hand side of the window as in Figure 3.9.

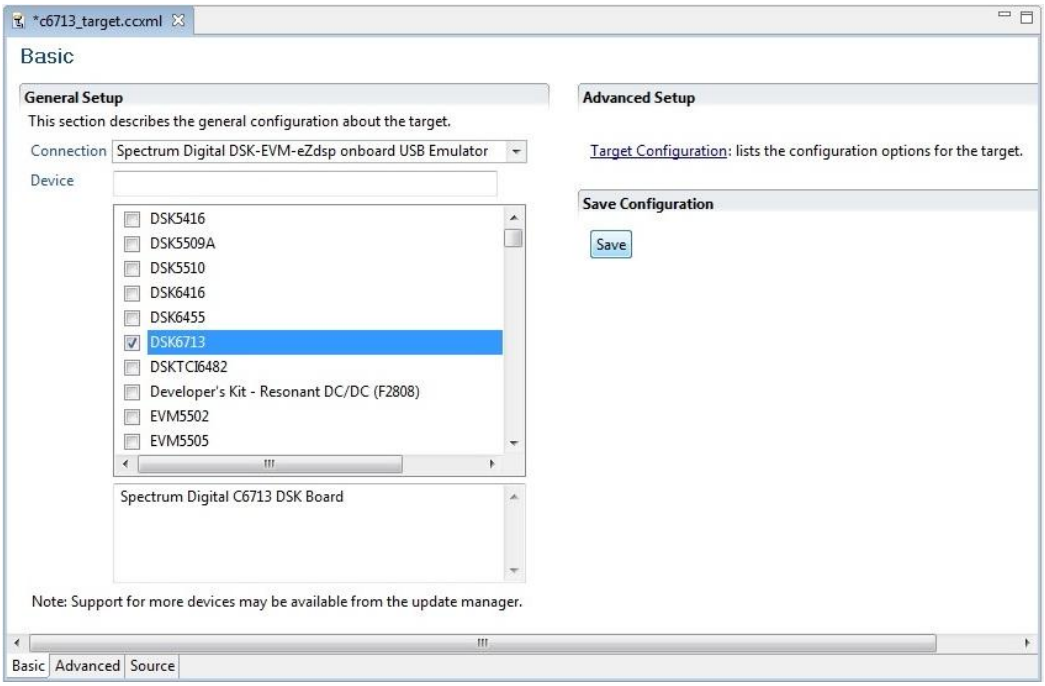


Figure 3.9 C6713 target configurations.

3.1.2 Creating a DSP/BIOS Configuration File

To create a configuration file, right click on the name of your project on the project tree. Select "New→DSP/BIOSv5.xx Configuration File" as in Figure 3.10.

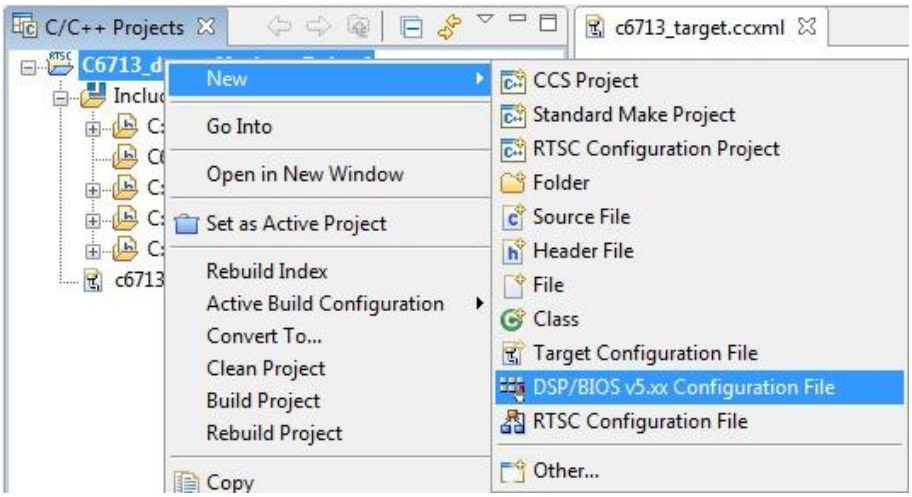


Figure 3.10 Creating the configuration file.

On the configuration window, enter a name to your configuration file with an extension ".tcf" as in Figure 3.11. Click on "Next".

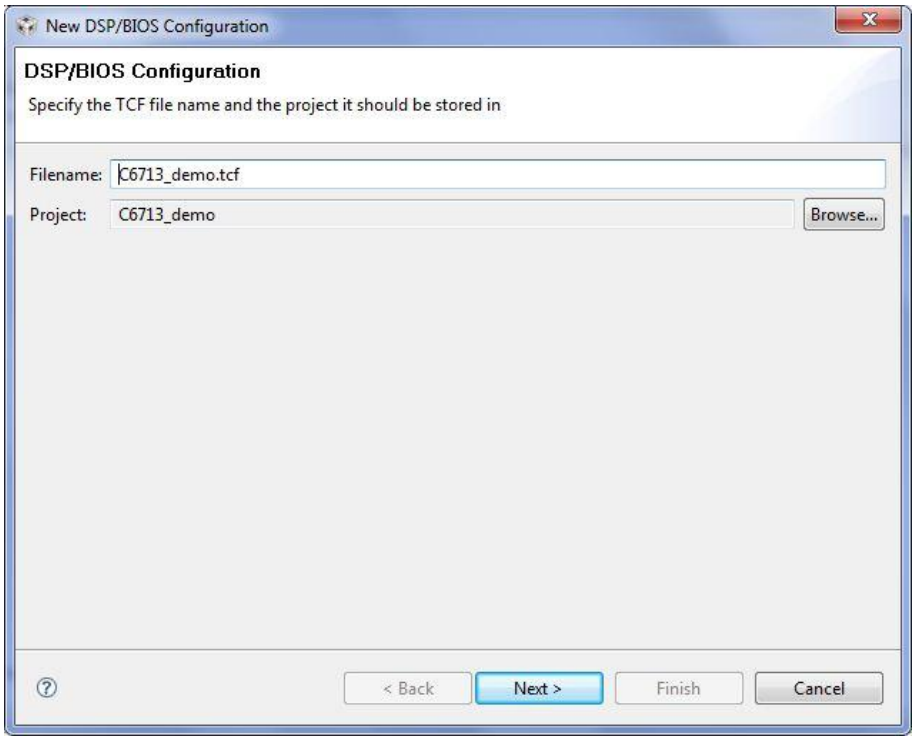


Figure 3.11 Entering the name of the configuration file.

On the opening list, select "ti.platforms.dsk6713" as in Figure 3.12. Click "Finish".

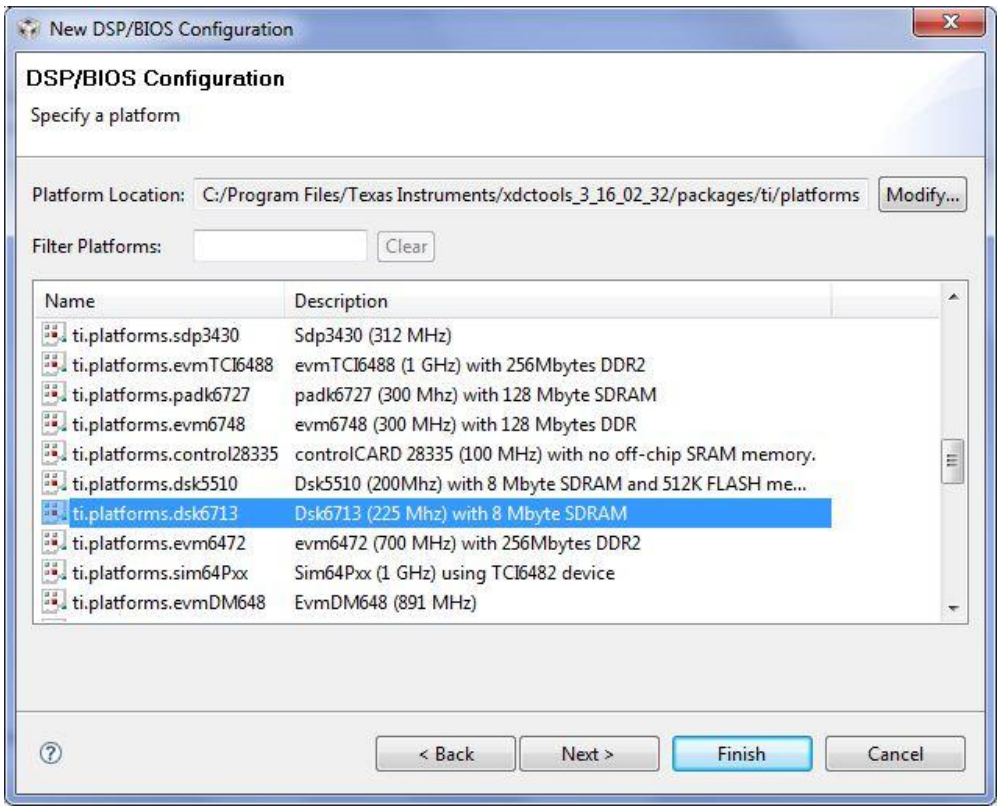


Figure 3.12 Selecting the platform on DSP/BIOS configuration.

If you do not want to change other configuration preferences, you can close the configuration tool as in Figure 3.13.

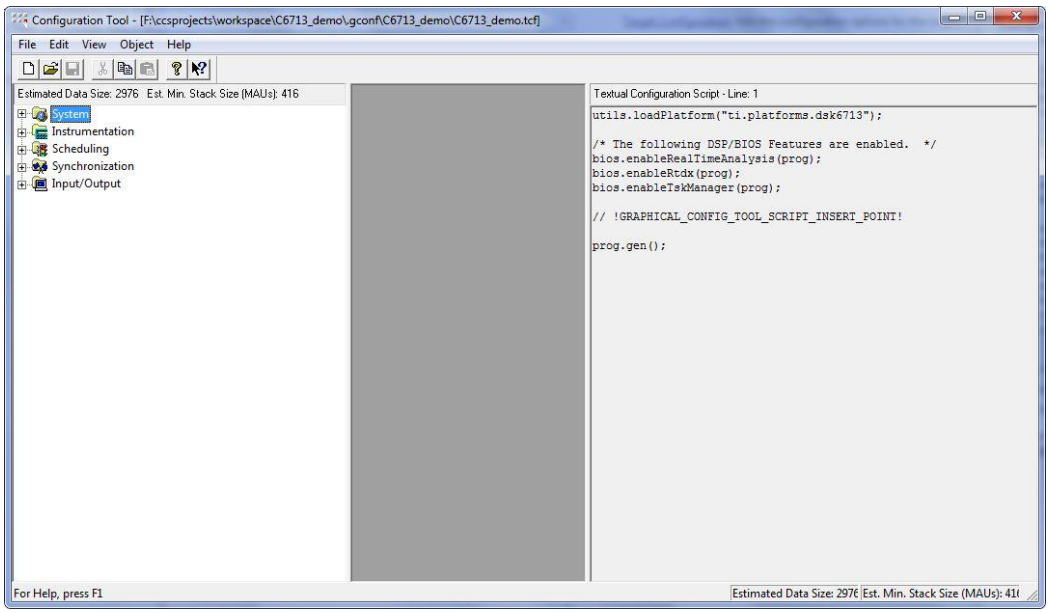


Figure 3.13 Closing the configuration tool.

3.1.3 Creating the Source File

To create the source file, right click on the name of your project on the project tree. Select "New→Source File" as in Figure 3.14.

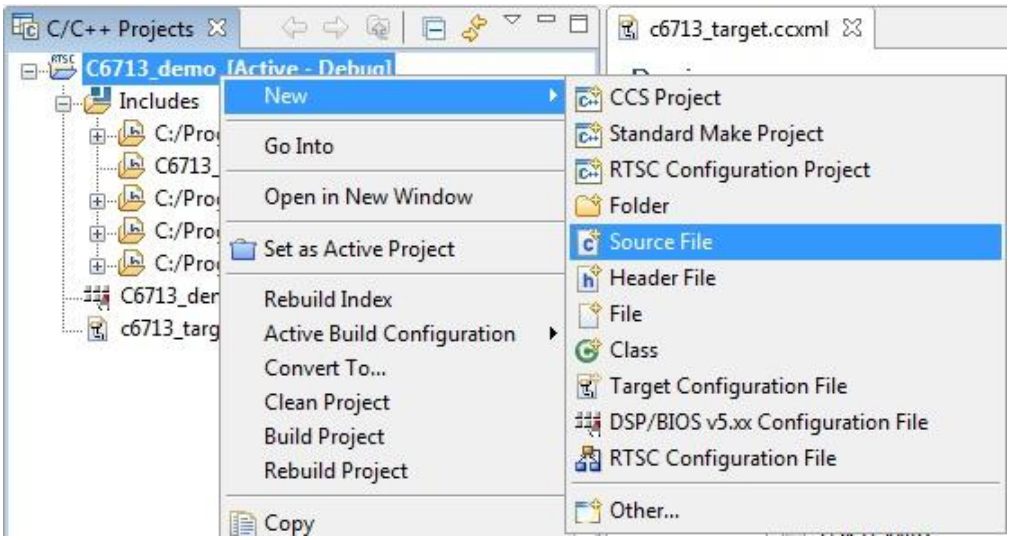


Figure 3.14 Creating the source file.

On the opening configuration window, enter a name to your source file with an extension ".c" as in Figure 3.15. Click on "Finish".

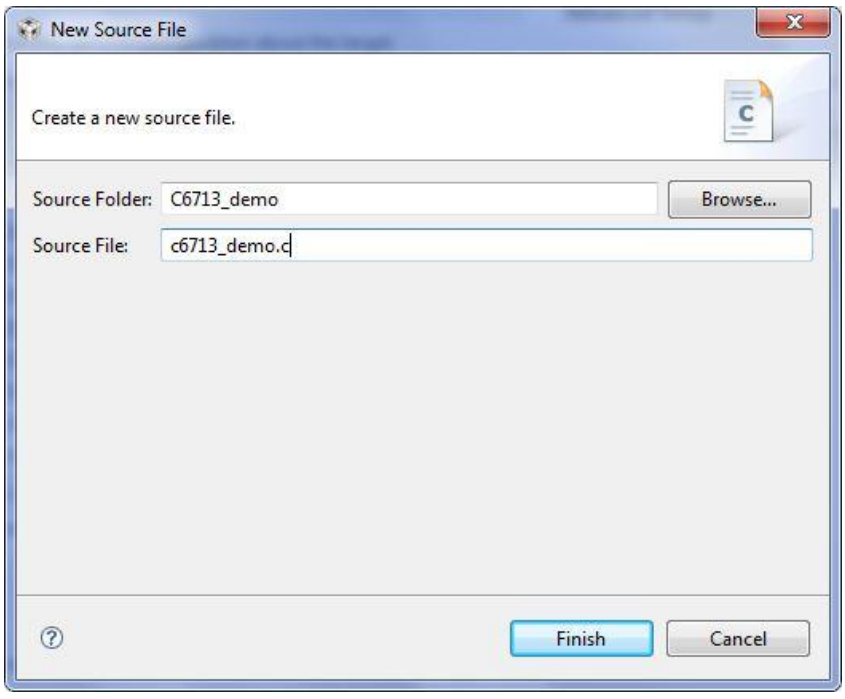


Figure 3.15 Entering the name of the source file.

As in the previous chapter, there is a simple source code in Figure 3.16.

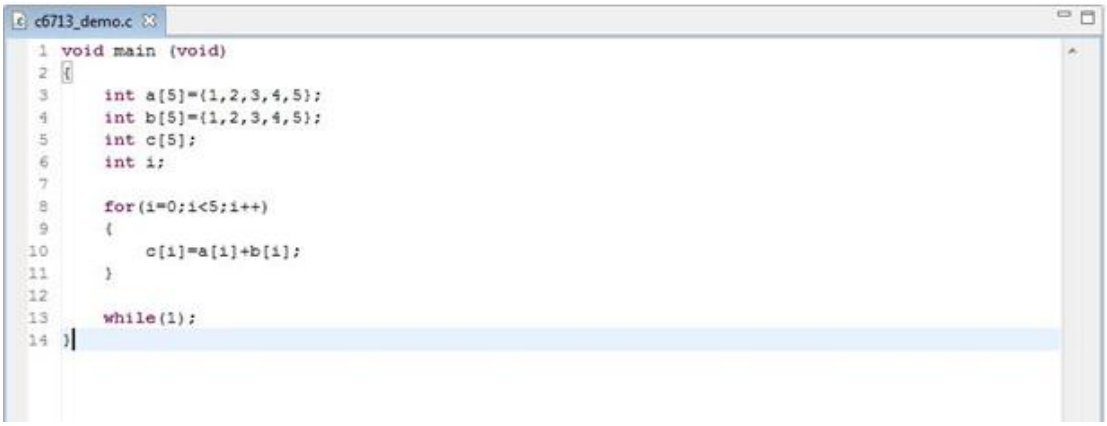


Figure 3.16 Simple source code.

3.2 Running the Source Code

To test these steps, you can start writing the source code given in Figure 3.16. To run this code, select "Project→Build Active Project" as in Figure 3.17.

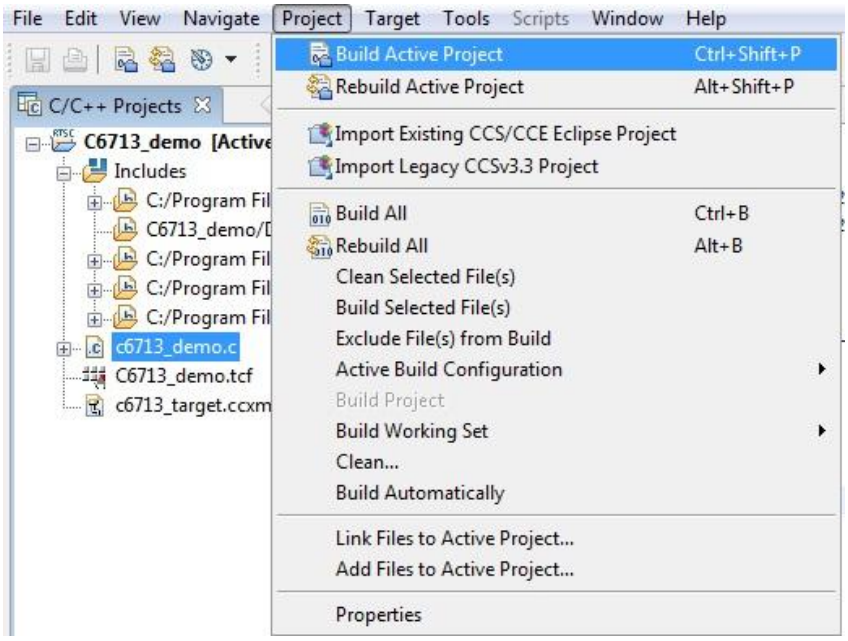


Figure 3.17 Building the active project.

If there are no errors in your code, you should see "Build complete sign" in the "Console" window as in Figure 3.18. Any errors or warnings will be listed in the companion "Problems" window.

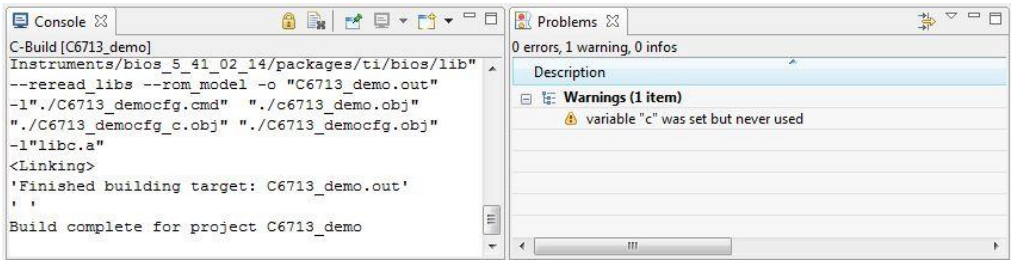


Figure 3.18 Console and Problems windows.

After building the project, select "Target→Debug Active Project" as in Figure 3.19.

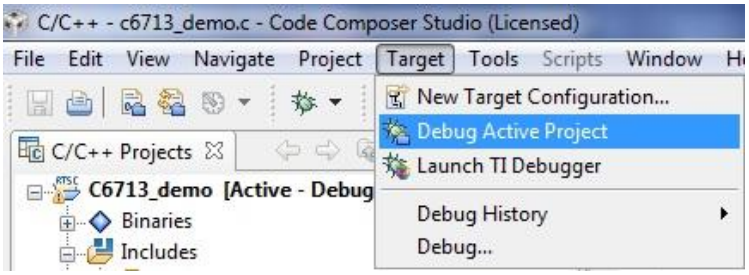


Figure 3.19 Debugging and active project.

Now, you should see the "Debug" window in the CCS V4.1 workspace as in Figure 3.20.

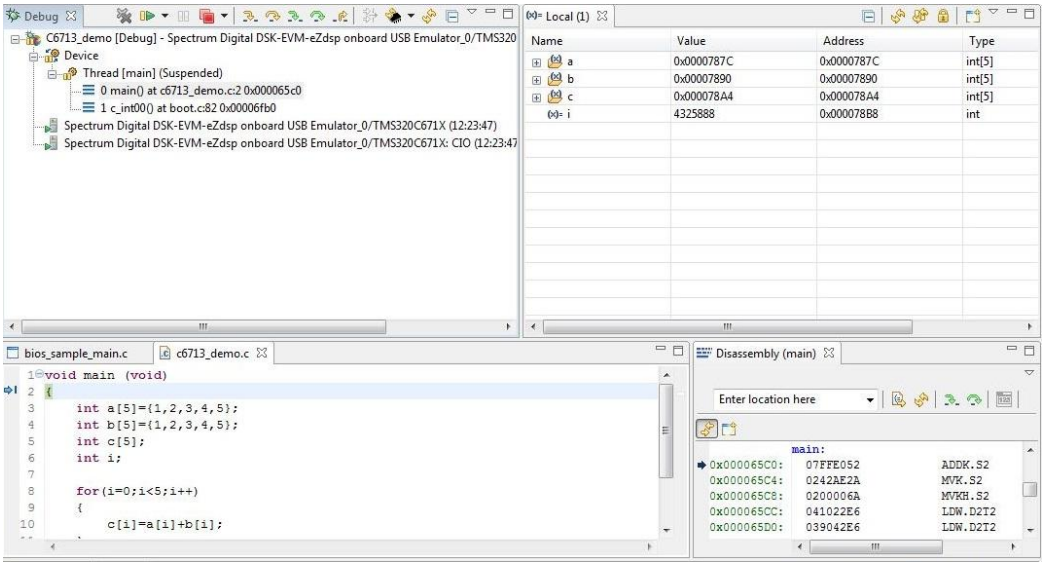


Figure 3.20 Debug window in CCS V4.1 workspace.

Press the green play button on the "Debug" window and select "Run" to start running your project as in Figure 3.21.

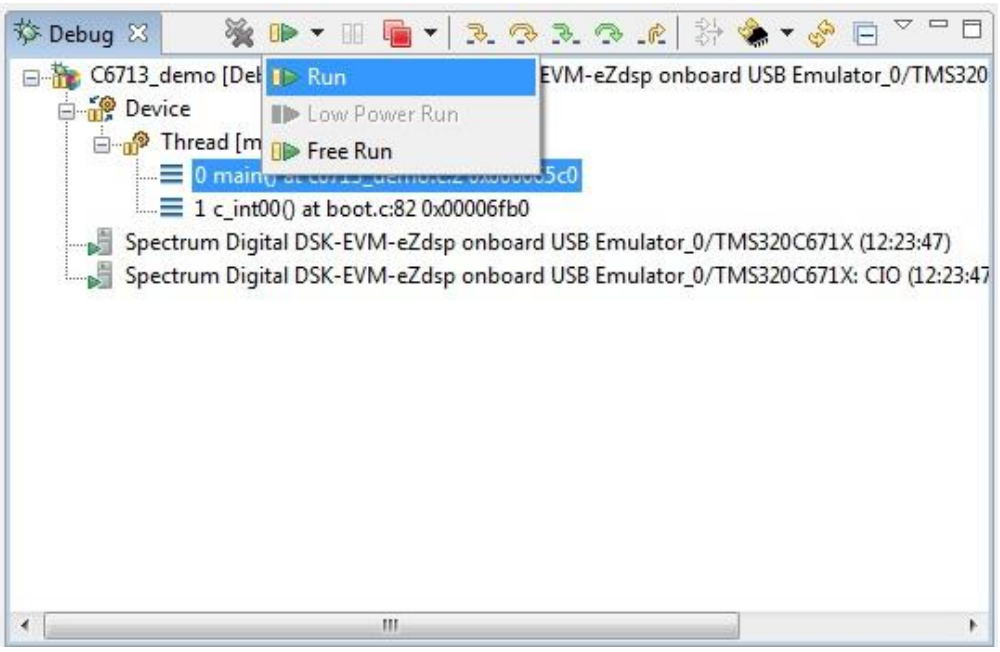


Figure 3.21 Running the project.

3.2.1 Inserting a Break Point

Right click on the row that you want to place a break point. Select "Toggle Breakpoint" as in Figure 3.22.

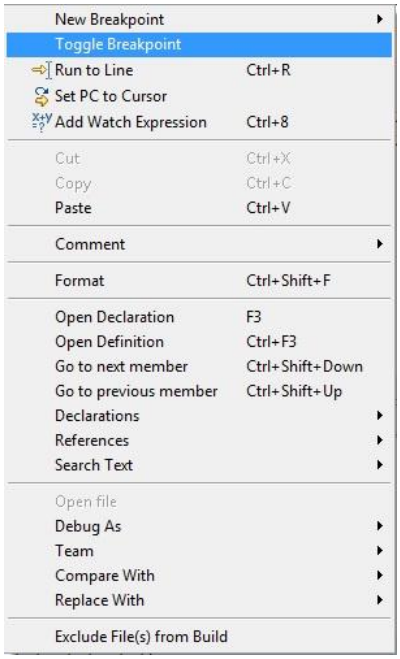


Figure 3.22 Inserting a breakpoint.

3.2.2 Adding a Watch Window

Select the variable you want to observe. Right click on it. Select "Add Watch Expression" as in Figure 3.23. You can observe the variable in the watch window as shown in Figure 3.24.

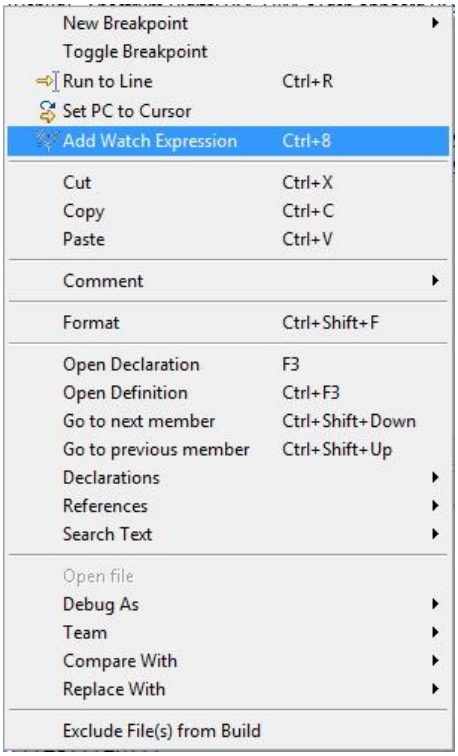


Figure 3.23 Adding a watch window.

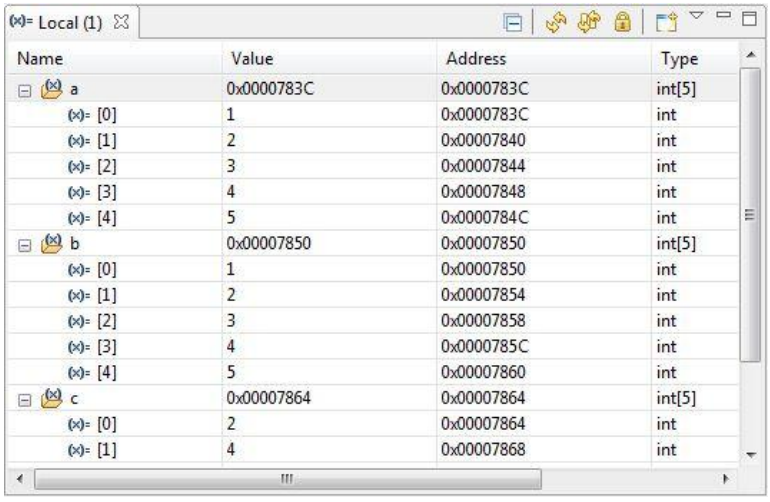


Figure 3.24 Observing the variable in the watch window.

3.2.3 Plots

To see an array in a graph, you should go to "Tools→Graph→Single Time" as in Figure 3.25.

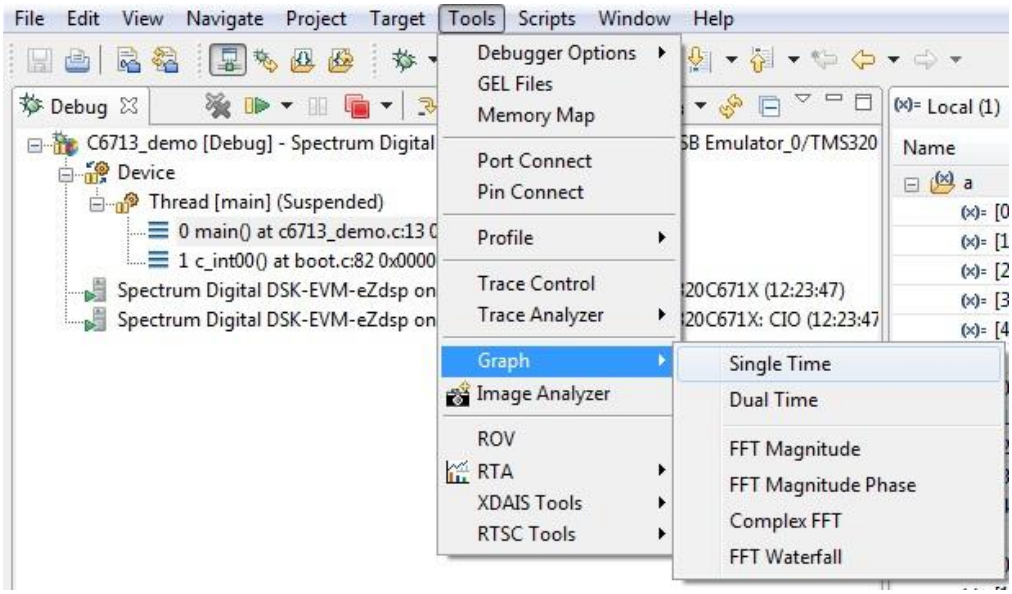


Figure 3.25 Plotting an array.

On the “Graph” properties window (as in Figure 3.26), fill:

i. **Start Address:** Name of the array (to be plotted).

ii. **Acquisition Buffer Size:** Size of the array.

iii. **Display Data Size:** Size of the data to be displayed on the graph.

iv. **DSP Data Type:** Type of the array (integer or float).

v. **Sampling Rate:** Sampling rate of the signal in the array.

For example, to observe the array 'c' in the above demo code, adjust the parameters as in Figure 3.26. The resulting graph will be as in Figure 3.27.

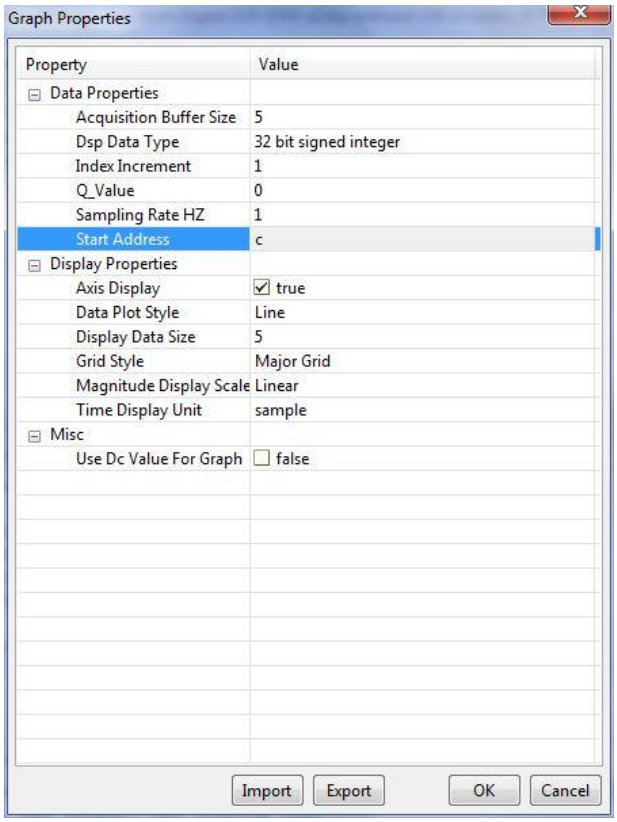


Figure 3.26 Setting graph properties.

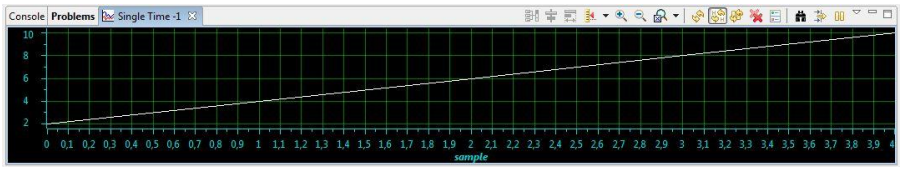


Figure 3.27 Graph of the array 'c'.

3.2.4 Images

To observe a grayscale image in size of 256x256, select "Tools→ Graph→ Image Analyzer" and make necessary configurations as shown in Figure 3.28 below.

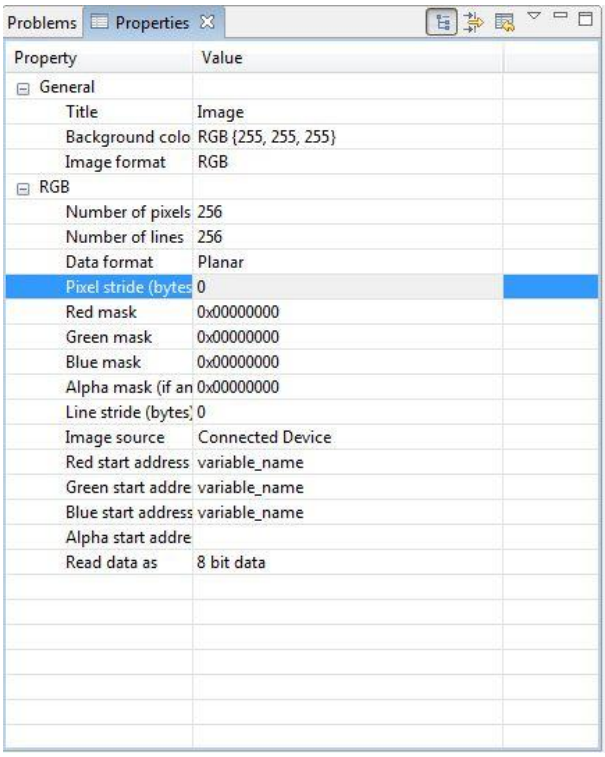


Figure 3.28 Displaying an image.

3.2.5 Saving the Data

To save your data, click "main" in the debug window as in Figure 3.29.

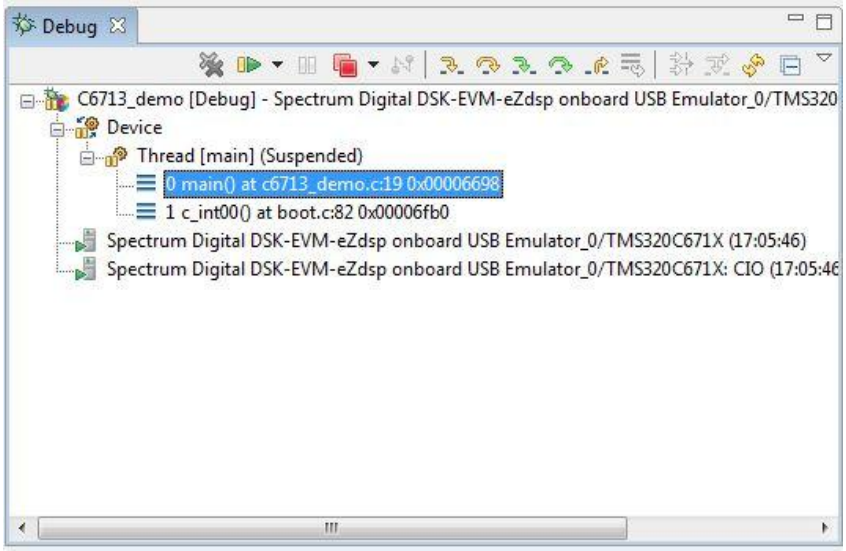


Figure 3.29 Saving the data.

Then, go to the "Memory" window. Click on "the green button" as shown in Figure 3.30. Select "save".

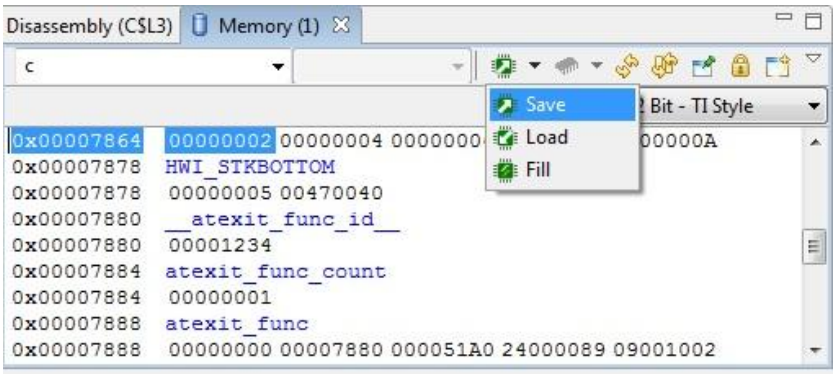


Figure 3.30 Selecting the memory window.

In the opening popup window, enter the name of the output data file as in Figure 3.31.

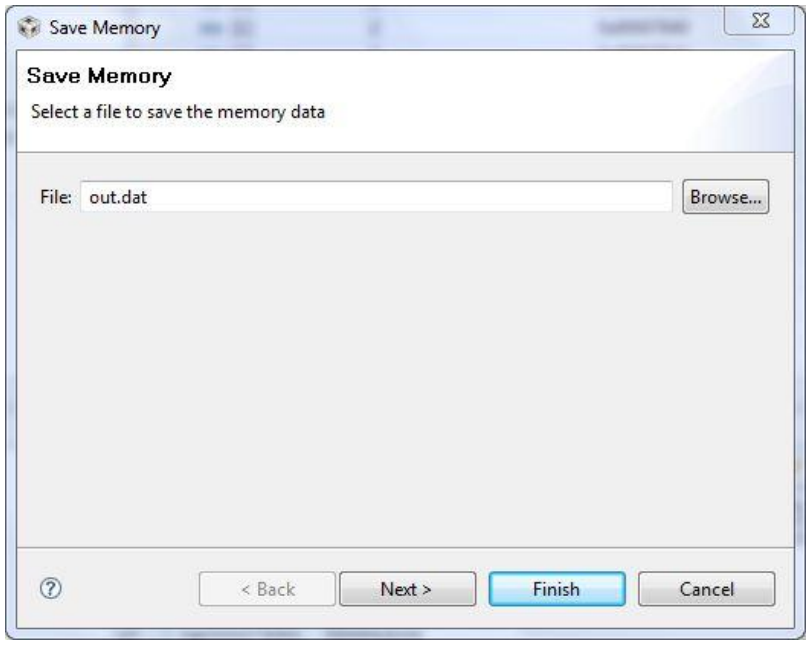


Figure 3.31 Entering the name of the output file.

Select the format to be saved as "integer". Enter "the name of the array" as the "start address". For the "length", enter the "length of the array". Click "Finish" to save the data as in Figure 3.32.

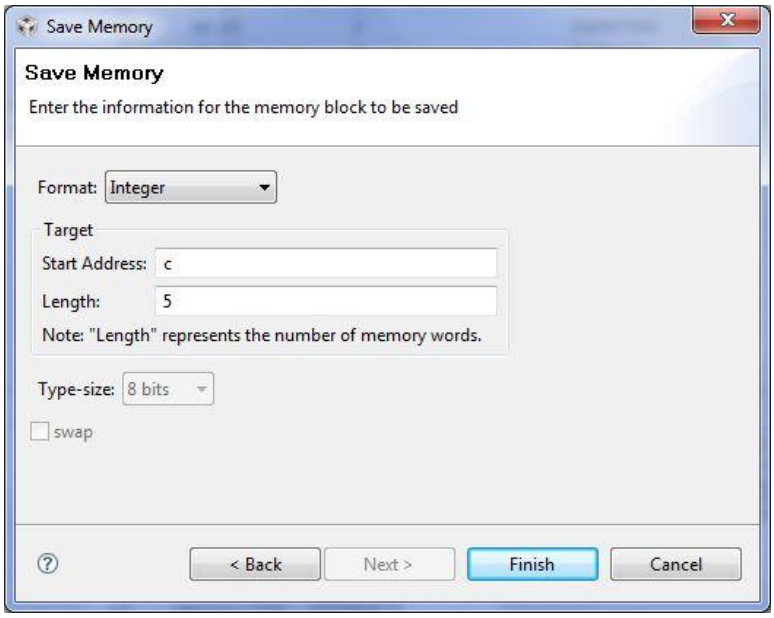


Figure 3.32 Selecting the properties of the output file.

3.3 CCS Gel Scripts

Code composer studio can be customized by using General Extension Language (GEL) Scripts. GEL is an interpretive language that enables writing functions to configure the IDE and access the target processor. Below is a sample script file we will use in Chapter 6 after designing filters.

```
menuitem "Gain Slider";
slider FilterGain1(1, 4, 1, 1, volume1)
{
 /* initialize the target variable with the Parameters passed by the slider object. */
   gain1 = volume1;
}
slider FilterGain2(1, 4, 1, 1, volume2)
{
 /* initialize the target variable with the Parameters passed by the slider object. */
   gain2 = volume2;
}
slider FilterGain3(1, 4, 1, 1, volume3)
{
 /* initialize the target variable with the Parameters passed by the slider object. */
   gain3 = volume3; }
```

To load a GEL file, select “Tools > GEL Files” as in Figure 3.33.

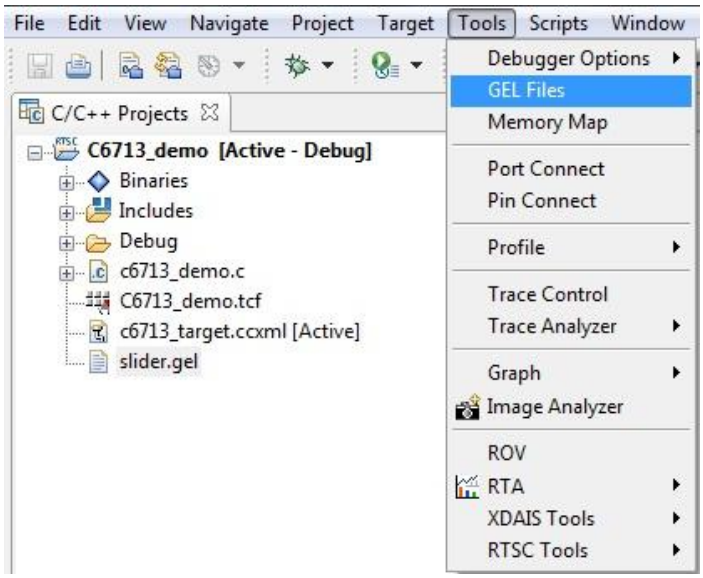


Figure 3.33 Loading a GEL file in CCS V4.1.

On the opening window, right click and select "Load GEL…". Load the **“volume_slider.gel”** file as in Figure 3.34.

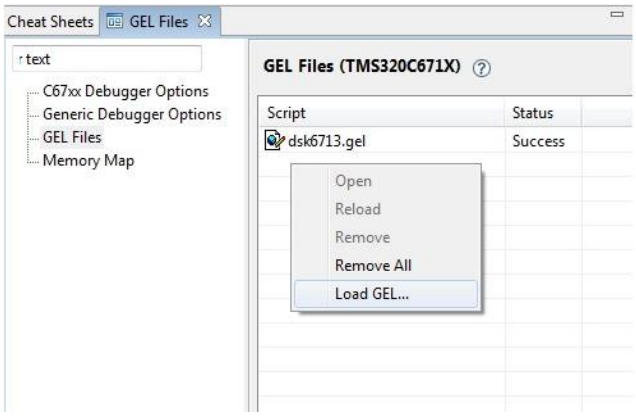


Figure 3.34 Loading the gel file.

To open the sliders, select "Scripts→Gain Slider→FilterGain1" as in Figure 3.35.

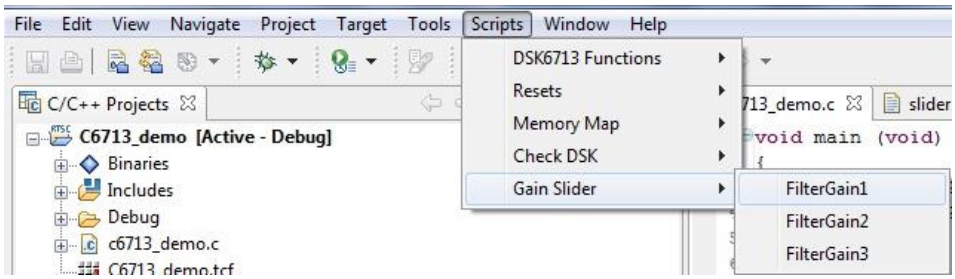


Figure 3.35 Opening gain sliders.

If all three gain sliders are loaded correctly, you should see them as in Figure 3.36.

Figure 3.36 Loaded gain sliders.

3.4 Working with DIP Switches and LEDs

C6713 DSK has four user accessible DIP switches and LEDs. To reach these peripheral devices, you should include two header files “dsk6713_dip.h” and "dsk6713_led.h" to your code. To add their paths to CCS V4.1, right click on "the name of your project" and select "Build Properties…" as in Figure 3.37.

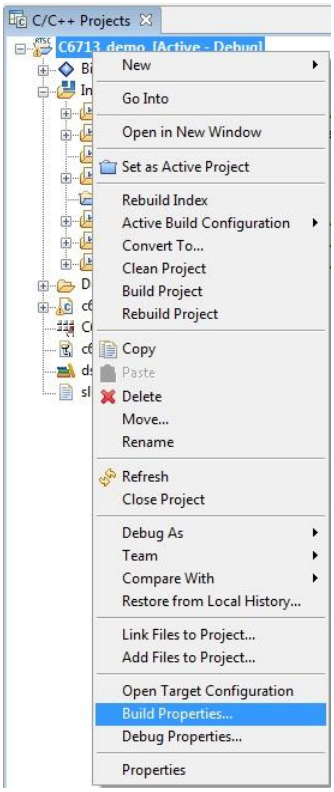


Figure 3.37 Build properties.

On the opening window, select “Include options”. Click on the  button to add the path of these files as in Figures 3.38 and 3.39.

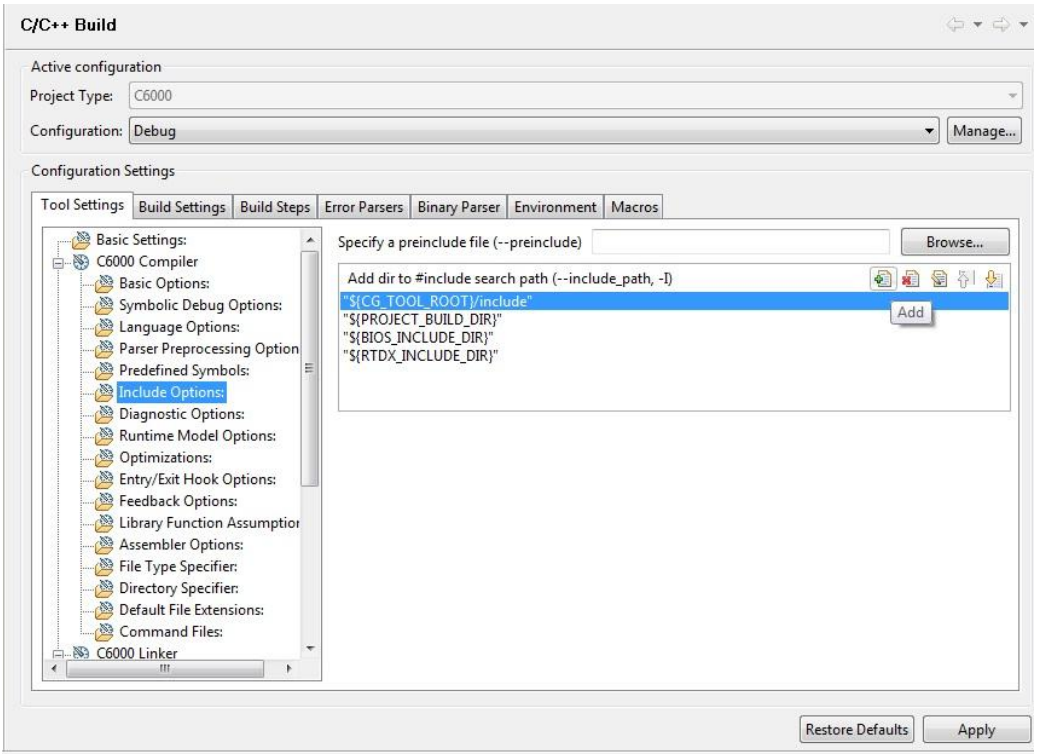


Figure 3.38 Active configuration.

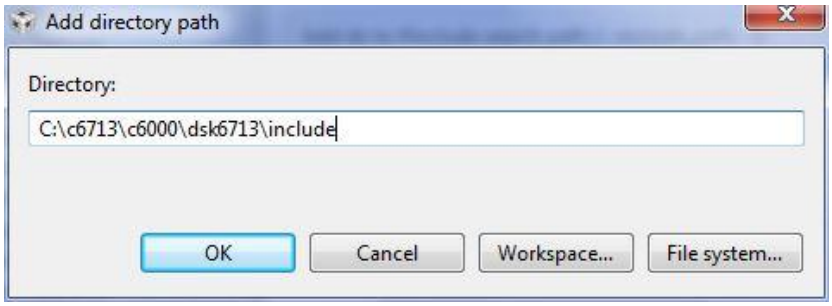


Figure 3.39 Adding directory path.

Exercises

1. Create an empty project named as “Fibonacci” in CCS V4.1.

a. Create a new target configuration file.

b. Create a new DSP/BIOS configuration file.

c. Create a new C source code file. Your code should calculate and save the first 10 elements of the Fibonacci series in an array.

d. Build and Run your project. Observe the graph of the Fibonacci array by using the graph property of CCS V4.1.

e. Save the Fibonacci array in a data file called “Result.dat”.

2. Create an empty project named as “Led_Switch” in CCS V4.1.

a. Include the path of all necessary header files to your project.

b. Add “dsk6713bsl.lib” file to your project.

c. Create a new target configuration file.

d. Create a new DSP/BIOS configuration file.

e. Create a new C source code file. This code should set the LED on the C6713 DSK board corresponding to the switch being pressed.

4. Basic Audio Effects

The aim in this chapter is implementing basic audio effects. This will lead to more complex code writing under CCS. We start with the basic audio signal processing with the C6713 DSK next.

4.1 Basic Audio Processing

This section is about the onboard TLV320AIC23 codec on the C6713 DSK. We will use this codec to input/output audio signals to the DSK.

4.1.1 TLV320AIC23 Codec

The TLV320AIC23 32-bit stereo codec is used for analog to digital (A/D) as well as digital to analog (D/A) conversion on the C6713 DSK board. This codec uses the sigma-delta conversion method for this purpose. Hereafter, we call this codec as AIC23.

There are four connectors on the C6713 DSK board to provide input and output to the system. These are: **MIC IN** for microphone input, **LINE IN** for line input, **LINE OUT** for line output, and **HEADPHONE** for headphone output (multiplexed with line output). The codec can select the microphone or the line as the active input. The analog output is driven to both the line out (fixed gain) and headphone (adjustable gain) connectors. The onboard codec and its input/output connections can be seen in Figure 4.1.

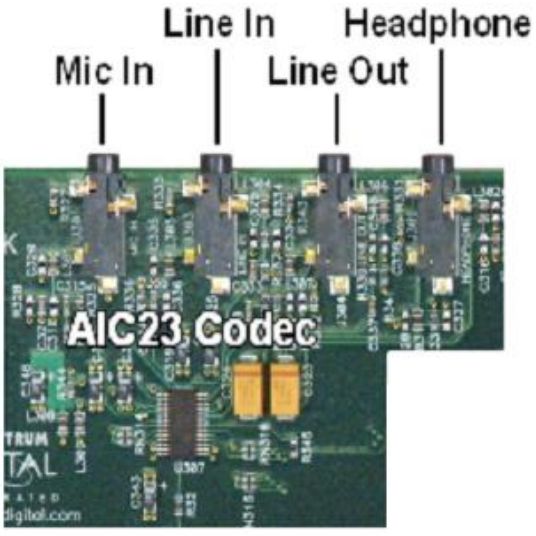


Figure 4.1 The AIC23 Codec on the C6713 DSK board.

TMS320C6713 DSP uses two Multi Channel Buffered Serial Port (McBSP) registers to communicate with the AIC23 codec. McBSP0 is used to send commands to the codec control interface. McBSP1 is used for digital audio data. It is possible to reach and change control registers. AIC23 codec peripherals and control registers can be seen in Figure 4.2.

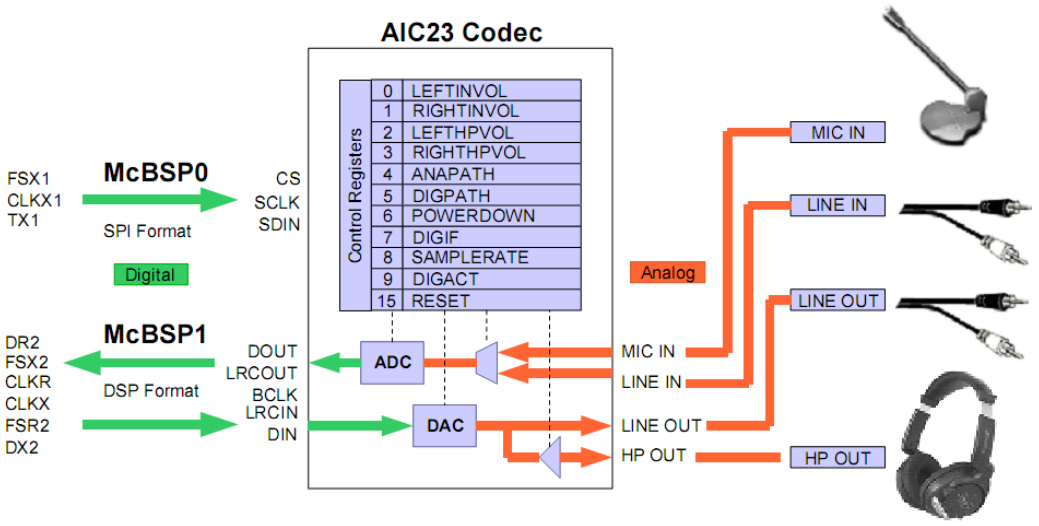


Figure 4.2 AIC23 codec peripherals and control registers.

AIC23 codec is optimized for audio applications and is highly configurable. It is connected to a 12 MHz. system clock. Variable sampling rates from 8 kHz to 96 kHz can be set to sample the input signal. Also quantization levels can be set into four different values as 16, 20, 24, and 32 bits by using control registers.

Input type (microphone or line) of the codec can be selected by setting appropriate bits of the *"Analog Audio Path Control"* register. Each bit of this register is described below.

Analog Audio Path Control (Address: 0000100)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	STA2	STA1	STA0	STE	DAC	BYP	INSEL	MICM	MICB
Default	0	0	0	0	1	1	0	1	0

STA[2:0] and STE

STE	STA2	STA1	STA0	ADDED SIDETONE
1	1	X	X	0 dB
1	0	0	0	–6 dB
1	0	0	1	–9 dB
1	0	1	0	–12 dB
1	0	1	1	–18 dB
0	X	X	X	Disabled

DAC	DAC select	0 = DAC off	1 = DAC selected
BYP	Bypass	0 = Disabled	1 = Enabled
INSEL	Input select for ADC	0 = Line	1 = Microphone
MICM	Microphone mute	0 = Normal	1 = Muted
MICB	Microphone boost	0=dB	1 = 20dB

The sampling rate can be selected using SR (Sampling Rate Control) bits of the *"Sample Rate Control"* register. Each control bit of this register is described below.

Sample Rate Control (Address: 0001000)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	CLKOUT	CLKIN	SR3	SR2	SR1	SR0	BOSR	USB/Normal
Default	0	0	0	1	0	0	0	0	0

CLKIN	Clock input divider	0 = MCLK	1 = MCLK/2
CLKOUT	Clock output divider	0 = MCLK	1 = MCLK/2
SR[3:0]	Sampling rate control (see Sections 3.3.2.1 AND 3.3.2.2)		
BOSR	Base oversampling rate		
	USB mode:	0 = 250 f_S	1 = 272 f_S
	Normal mode:	0 = 256 f_S	1 = 384 f_S
USB/Normal	Clock mode select:	0 = Normal	1 = USB
X	Reserved		

The quantization level can be selected using IWL (Input Bit Length) bits of the *"Digital Audio Interface Format"* register. Each bit of this register is described below.

Digital Audio Interface Format (Address: 0000111)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	X	MS	LRSWAP	LRP	IWL1	IWL0	FOR1	FOR0
Default	0	0	0	0	0	0	0	0	1

MS	Master/slave mode	0 = Slave 1 = Master
LRSWAP	DAC left/right swap	0 = Disabled 1 = Enabled
LRP	DAC left/right phase	0 = Right channel on, LRCIN high 1 = Right channel on, LRCIN low DSP mode 1 = MSB is available on 2nd BCLK rising edge after LRCIN rising edge 0 = MSB is available on 1st BCLK rising edge after LRCIN rising edge
IWL[1:0]	Input bit length	00 = 16 bit 01 = 20 bit 10 = 24 bit 11 = 32 bit
FOR[1:0]	Data format	11 = DSP format, frame sync followed by two data words 10 = I^2S format, MSB first, left – 1 aligned 01 = MSB first, left aligned 00 = MSB first, right aligned

More information about control registers can be found in theTLV320AIC23 datasheet.

4.1.2 Sampling an Audio Signal

The AIC23 audio codec will be used to sample the audio signal acquired from the microphone. In order to use this codec, first include the "dsk6713_aic23.h" header file to your code. Then, configure the control registers of the codec before the main program starts. If the register parameters are not set, the default parameters are used. In the next code block, all adjustments are done. Below, some parameters in this code block are intentionally left blank.

```
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "stdlib.h"
```

```
#include "math.h"

/* Configure Codec */
DSK6713_AIC23_Config config = {
0x0017,          /* 0 - DSK6713_AIC23_LEFTINVOL Left line input channel volume */
0x0017,          /* 1 - DSK6713_AIC23_RIGHTINVOL Right line input channel volume */
0x01f9,          /* 2 - DSK6713_AIC23_LEFTHPVOL Left channel headphone volume */
0x01f9,          /* 3 - DSK6713_AIC23_RIGHTHPVOL Right channel headphone volume */
0x????,          /* 4 - DSK6713_AIC23_ANAPATH Analog audio path control */
0x0000,          /* 5 - DSK6713_AIC23_DIGPATH Digital audio path control */
0x0000,          /* 6 - DSK6713_AIC23_POWERDOWN Power down control */
0x0043,          /* 7 - DSK6713_AIC23_DIGIF Digital audio interface format */
0x????,          /* 8 - DSK6713_AIC23_SAMPLERATE Sample rate control */
0x0001           /* 9 - DSK6713_AIC23_DIGACT Digital interface activation */ }

void main(){
   DSK6713_AIC23_CodecHandle hCodec;
   Int16 OUT_L,OUT_R;
   Uint32 IN_L;

   /* Initialize the board support library. It must be called first */
   DSK6713_init();

   /* Start the codec */
   hCodec = DSK6713_AIC23_openCodec(0, &config);

  /*  Set codec frequency to 48KHz */
 DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_48KHZ );

while (1) {
    /* Read sample from the left channel */
       while (!DSK6713_AIC23_read(hCodec, &IN_L));

    /* Feeding the input directly to output */
      OUT_L = IN_L;
```

```
      OUT_R = IN_L;

    /* Delay effect will be added here */
    /* Echo effect will be added here */
    /* Reverberation effect will be added here */

  /* Send the sample to the left channel */
       while (!DSK6713_AIC23_write(hCodec, OUT_L));

  /* Send  the sample to the right channel */
       while (!DSK6713_AIC23_write(hCodec, OUT_R));
           }

  /* Close the codec */
   DSK6713_AIC23_closeCodec(hCodec);
}
```

Now, you are ready to get the audio signal from the microphone. In the following chapters, we will use the same setting repeatedly to process audio signals. Therefore, it is important to understand the codec parameters and their values.

4.2 Audio Effects

In natural environments, sounds we perceive depend on the listening conditions and in particular on the acoustic environment. The same sound signal played in a concert hall; bathroom or a small room will not be “perceived” the same. Since these effects are important for musicians, the digital technology can be used to simulate them. In this section, we will implement some of these effects to our audio signal.

4.2.1 Delay

Delay is the simplest audio effect which holds input signal and then plays it back after a period of time. Delay is used very often by musicians. It is also the main block for other audio effects such as reverb, chorus, and flanging.

The difference equation for the delay operation is $y[n] = x[n-n_0]$ where 'n_0' is the delay amount. Since the difference equation between the output and the input is specified, it can

be directly coded in C language. To implement the delay operation, the best way is defining an array which stores input audio signals. In Figure 4.3, we demonstrate the delay operation using "n_0+1" length array which should be defined beforehand. To feed the delayed audio signal to output, first we should store the audio signal on the first entry of the array. At each operation cycle, each entry in the array should be shifted towards right, to open space to the new input. This way, an input which is taken at time "n" will reach to the end of the array "n_0" cycles later.

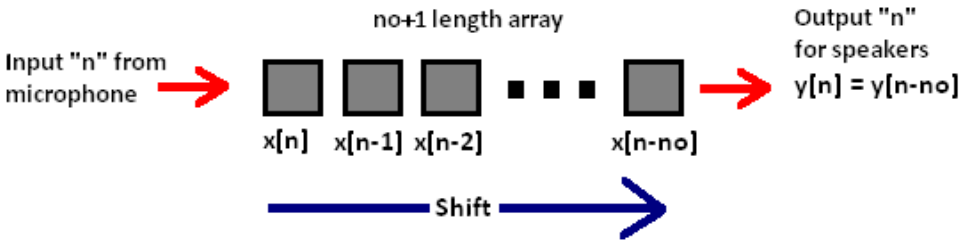


Figure 4.3 Demonstration of the delay operation.

4.2.2 Echo

The echo block simply takes an audio signal and plays it after summing with a delayed version of it. The delay time can range from several milliseconds to several seconds. The basic echo block is given in Figure 4.4.

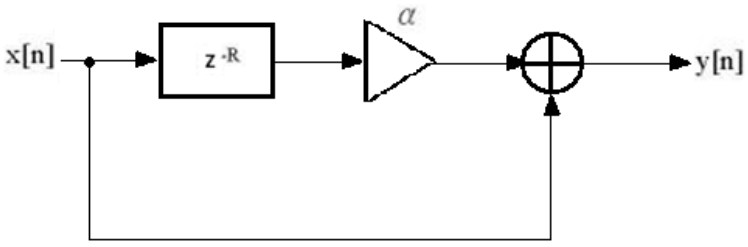


Figure 4.4 Echo block.

The difference equation for this system is $y[n] = x[n] + \alpha x[n-n_0]$; where '$\alpha$' is the delay mix parameter, 'n_0' is the delay amount. As can be seen in this equation, to generate the echo effect we need both the present and the delayed signal. In order to access to the delayed signal, we should store the input audio signal in an array as in previous section. Using this

array, we can process the audio signal $x[n-n_0]$. Summing this $x[n- n_0]$ value (after multiplying with α) with the present input x[n], we can generate the echo effect.

4.2.3 Reverberation

Reverberation is also one of the most heavily used effects in music. The effects of combining an original signal with a very short (<20ms) time-delayed version of itself results in reverberation. In fact, when a signal is delayed for a very short time and then added to the original, the ear perceives a fused sound rather than two separate sounds. If a number of very short delays (that are fed back) are mixed together, a crude form of reverberation can be achieved. The block diagram of a basic reverberation system is given in Figure 4.5.

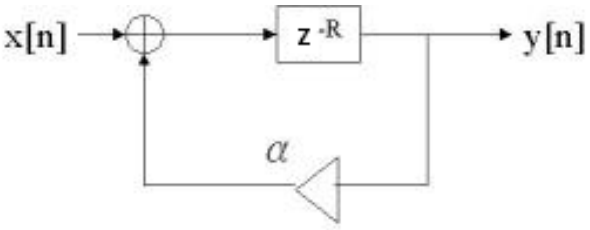


Figure 4.5 The block diagram of a multi-echo reverberation system.

The difference equation for this system is $y[n] = \alpha y[n-n_0] + x[n-n_0]$. As can be seen in Figure 4.5 and its difference equation, we need both the delayed input and output signals in order to implement a reverberation system. Therefore, different from "delay" and "echo" effects, in the reverberation system we need to store two signals. We can reach to $x[n- n_0]$ and $y[n-n_0]$ signals by storing input and output signals in an array. Then, the reverberation effect is obtained by summing the weighted delayed output ($\alpha y[n- n_0]$) with the delayed input $x[n-n_0]$. By increasing α, we can increase the effect of previous outputs in recent output signal. To prevent overflow at the output signal, "α" value should be chosen between [0, 1].

Exercises

1. Create an empty project named "Audio_in_out" in CCS V4.1.

   a. Create a new target configuration file.
   b. Create a new DSP/BIOS configuration file.
   c. Add "dsk6713bsl.lib" file to your project.
   d. Add "csl6713.lib" file to your project.

e. Create a new C source code file to input audio signals from the microphone and feed them to output (to the speaker). Apply necessary configurations in your source code to set AIC23 Audio Codec control registers in order to use the microphone as the input.
f. Set the sampling frequency to 48 kHz. Set the quantization level to 16 bits.
g. Build and run your project. If your code works correctly, you should clearly hear what you talk to microphone.

2. Add a buffer array to the source code in Exercise 1.
   a. Change your code to store the last 512 samples acquired from the input to this buffer array.
   b. Observe the signal in the buffer array by using the Graph property of CCS.
   c. Save the buffer array in a data file called “48khz.dat”.

3. Change the sampling frequency to 8 kHz and repeat Exercise 2.

4. Plot the audio samples in “48khz.dat “and “8khz.dat” using another program.

5. Change the quantization level into 20, 24, and 32 bits and repeat Exercise 2. What is the effect of quantization on the memory storage space?

6. Apply delay, echo, and reverberation effects to your audio signal separately.

7. Write a code to select the audio effect using DIP switches on the C6713 DSK.
   a. The user will select the desired audio effect by using the DIP switch.
   b. The corresponding LED will also show the selected DIP switch.
   c. The user can change the audio effect while the code is running.

5. Spectrum Analysis on CCS

Spectrum analysis is mandatory for filter design and analysis. Therefore, this chapter deals with Discrete Fourier Transform (DFT) and Fast Fourier transform (FFT) on C6713 DSK. Although the DFT is not used in practice, we consider it first for two main purposes. First, it is the basis for FFT. Second, while implementing DFT the reader can become familiar with mathematical operations on the C6713 DSK.

5.1 Discrete Fourier Transform (DFT)

The Discrete-time Fourier Transform (DTFT) of the sequence x[n] is defined by:

$$X(e^{jw}) = \sum_{n=-\infty}^{\infty} x[n]e^{-jwn}$$

The Discrete Fourier Transform is the sampled version of the DTFT. It is useful on only finite-length signals. Therefore, for a finite-length signal x[n], such that x[n]≠0 within the interval 0 ≤ n ≤ N-1, the DFT is defined by:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}nk} \qquad k = 0,1,\dots,N-1$$

In general, *X[k]* is a complex function and can be written in rectangular form as:

$$X[k] = X_r[k] + jX_i[k]$$

where $X_r[k]$ *and* $X_i[k]$ are real and imaginary parts respectively. In C programming libraries, the complex term *j* is not defined. Therefore, DFT is calculated in rectangular form using Euler's formula:

$$e^{-j\frac{2\pi}{N}nk} = \cos\left(\frac{2\pi}{N}nk\right) - jsin(\frac{2\pi}{N}nk)$$

Here, the real part of the input sequence is multiplied by the *cosine* term and the imaginary part is multiplied by the *sine* term. The magnitude and phase components of *X[k]* are calculated by:

$$|X[k]| = \sqrt{X_r^2[k] + X_i^2[k]}$$

$$\theta(w) = \tan^{-1}\left(\frac{X_i[k]}{X_r[k]}\right)$$

Based on the above formulation, we provide the C code to calculate the DFT below. Only the core DFT calculation function is intentionally left blank to the reader.

```
#include <math.h>
#include "sine.h"
#include "cosine.h"
#include "square.h"

/* sine and cosine header files are lookup tables. They should be prepared beforehand.*/

#define pi 3.14159

void dft(float xreal[],float ximag[],float yreal[],float yimag[]);

void main(void)
{
float xreal[N];
float ximag[N];
float yreal[N];
float yimag[N];
float mag[N];
int i;

for(i=0; i<N; i++)
{
xreal[i] = 0.0;
ximag[i] = 0.0;
yreal[i]=0.0;
yimag[i]=0.0;
}

/* Assign the real signal, we want to calculate its DFT, to the variable xreal. If the signal has
an imaginary part, assign it to ximag.*/

dft(xreal,ximag,yreal,yimag);

/* The DFT result will be at yreal and yimag variables. */

for(i=0;i<N;i++)
{
        mag[i] = sqrt(pow(yreal[i], 2) + pow(yimag[i], 2));
}

while(1);
```

```
}
void dft(float xreal[],float ximag[],float yreal[],float yimag[])
{
        /*Write a code that calculates the DFT here. */
}
```

5.2 The Fast Fourier Transform (FFT)

The fast Fourier transform (FFT) is a specific DFT algorithm that reduces the number of computations needed for N points from $2N^2$ to $2N\log_2 N$. TI's DSPLib library has a predefined function for FFT. Next, we will use it to calculate the frequency spectrum of a signal. First, we should add four files to the project as follows.

Right Click on "Active Project ➔ Add files to project":

C:\DSPLIB_INSTALLATION_DIRECTORY\c6700\dsplib\lib ➔ dsp67x.lib

C:\ DSPLIB_INSTALLATION_ DIRECTORY \c6700\dsplib\support\fft ➔ utility.c

C:\CCS_INSTALLATION_ DIRECTORY \C6000\dsk6713\lib ➔ dsk6713bsl.lib

They should also be added to the search path for compiling as follows.

Right Click on "Active Project ➔ Build Properties ➔Add dir to #include Search Path (-i)":

C:\ CCS_INSTALLATION_ DIRECTORY \C6000\dsk6713\include;

C:\ CCS_INSTALLATION_ DIRECTORY \C6000\cgtools\include;

C:\ CCS_INSTALLATION_ DIRECTORY \C6000\csl\include;

C:\ DSPLIB_INSTALLATION_ DIRECTORY \c6700\dsplib\support\fft;

C:\ DSPLIB_INSTALLATION_ DIRECTORY \c6700\dsplib\include;

To calculate the single-precision floating-point radix-2 FFT with complex input, we can use TI's predefined library function as:

void DSPF_sp_cfftr2_dit (float * x, float * w, short n)

Here,

'x' Pointer to complex input array.

'w' Pointer to complex twiddle factor in bit-reverse order.

'n' Length of the FFT in complex samples, power of 2 such that n ≥ 32.

This routine performs the decimation-in-time (DIT) radix-2 FFT of the input array. The input array 'x' contains N complex floating-point numbers (N*2 elements) arranged as successive

real and imaginary pairs. The coefficients for the FFT are passed to the function in array ‘w’ which contains N/2 complex numbers (N elements) as successive real and imaginary number pairs. The FFT coefficients w are in N/2 bit-reversed order. The elements of input array x are in normal order.

To calculate the inverse FFT, we can use TI's predefined function with single-precision inverse, complex, radix-2, decimation in frequency FFT as:

void DSPF_sp_icfftr2_dif (float* x, float* w, short n)

Here

'x' Input and output sequences. x has n complex numbers (2n single precision numbers). The real and imaginary values are interleaved in memory. The input is in bit-reversed order and output is in normal order.

'w' FFT coefficients. w has n/2 complex numbers (n single precision numbers). FFT coefficients must be in bit-reversed order. The real and imaginary values are interleaved in memory.

'n' Length of the FFT in complex samples (input).

This routine is used to compute the inverse, complex, radix-2, decimation-in frequency. Fast Fourier Transform of a single-precision complex sequence of size n and a power of 2. The routine requires bit-reversed input and bit-reversed coefficients (twiddle factors) and produces results that are in normal order.

TI's predefined FFT coefficient generator function must also be used to calculate the FFT as:

void tw_genr2fft(float*w, int n)

Here

'w' Pointer to complex twiddle factor in bit-reverse order.

'n' Length of the FFT.

This routine calculates the twiddle factor array for use in the tw_genr2fft routine. After generation, the array still needs to be bit-reversed. We should also use TI's predefined bit reversal function as:

void bit_rev(float* x, int n)

Here

'x' Pointer to the complex input array.

'n' Size of the array x.

This function is used to bit reverse the given array used in the tw_genr2fft routine to bit_rev the twiddle factor array as well as to bit_rev the output. Remember that the size of the twiddle factor array is N/2 complex floats, and the size of the output array is N complex floats (where N is the dimension of the FFT).

To explain the usage of these functions, we pick an FFT example as follows. We pick the input signal $x[n] = sin(2\pi 10n) + sin(2\pi 40n)$ having 512 (2*N) samples. We obtain the magnitude of the calculated FFT having size N complex numbers. We also calculate the inverse FFT on the same example. We provide the source code for this example below.

```
#define CHIP_6713  1
#include "dsk6713.h"
#include "stdlib.h"
#include "math.h"
#include "utility.h"

#define PI 3.14159265358979323846
#define N 512

float x[2 * N];
float w[N];

void divide(float *x,int n)
{
        int i;
        float inv = 1.0 / n;

        for(i=0;i<n;i++)
        {
                x[2 * i]=inv * x[2 * i];
                x[2 * i + 1]=inv * x[2 * i + 1];
        }
}

void main()
{
        int i;
        float mag[N];
```

```
        for(i=0;i<N;i++)
        {
                x[2 * i]= sin(2 * PI * 10 * i / N) + sin(2 * PI * 40 * i / N);
                x[2 * i +1]=0;
        }

        tw_genr2fft(w,N);
        bit_rev(w,N>>1);
        DSPF_sp_cfftr2_dit(x,w,N);
        bit_rev(x,N);

        for(i=0;i<N;i++)
        {
                mag[i]=sqrt((x[2 * i] * x[2 * i]) + (x[2 * i + 1] * x[2 * i + 1]));
        }

        bit_rev(x,N);
        DSPF_sp_icfftr2_dif(x,w,N);
        divide(x,N);

        while(1);
}
```

To visualize the input, we should put a break point to the code line “tw_genr2fft(w,N)”. We should also open the time graph and adjust its properties as in Figure 5.1.

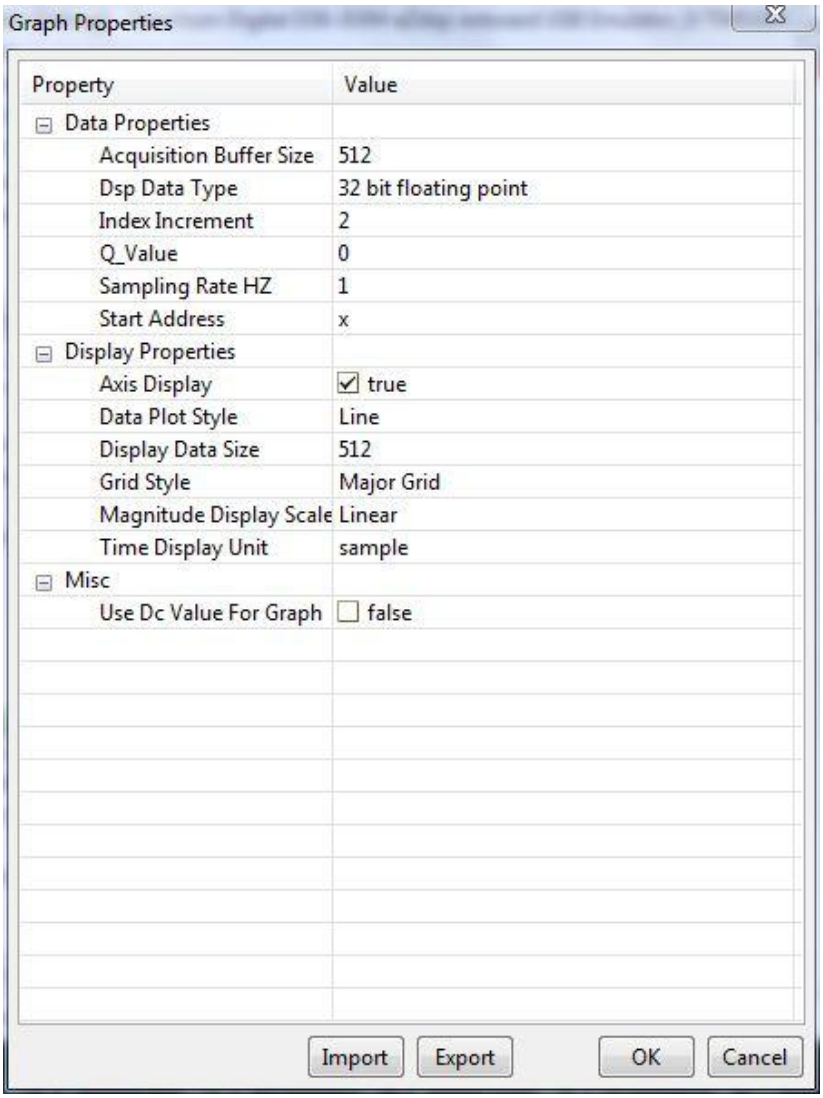


Figure 5.1 Setting graph properties for visualizing the input signal.

Then, we obtain the original signal in time domain as in Figure 5.2.

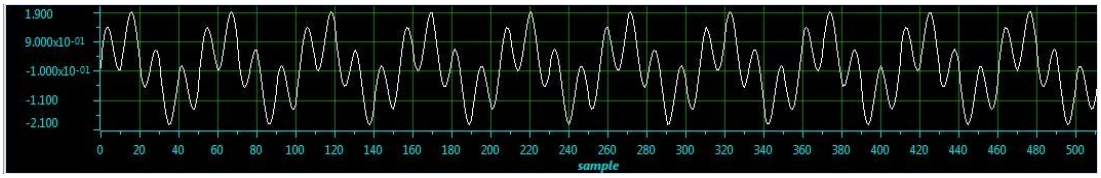


Figure 5.2 The original signal.

To visualize the FFT magnitude output, we should place a break point just after the for loop that calculates the magnitude. Then, we should open the time graph and adjust its properties as in Figure 5.3.

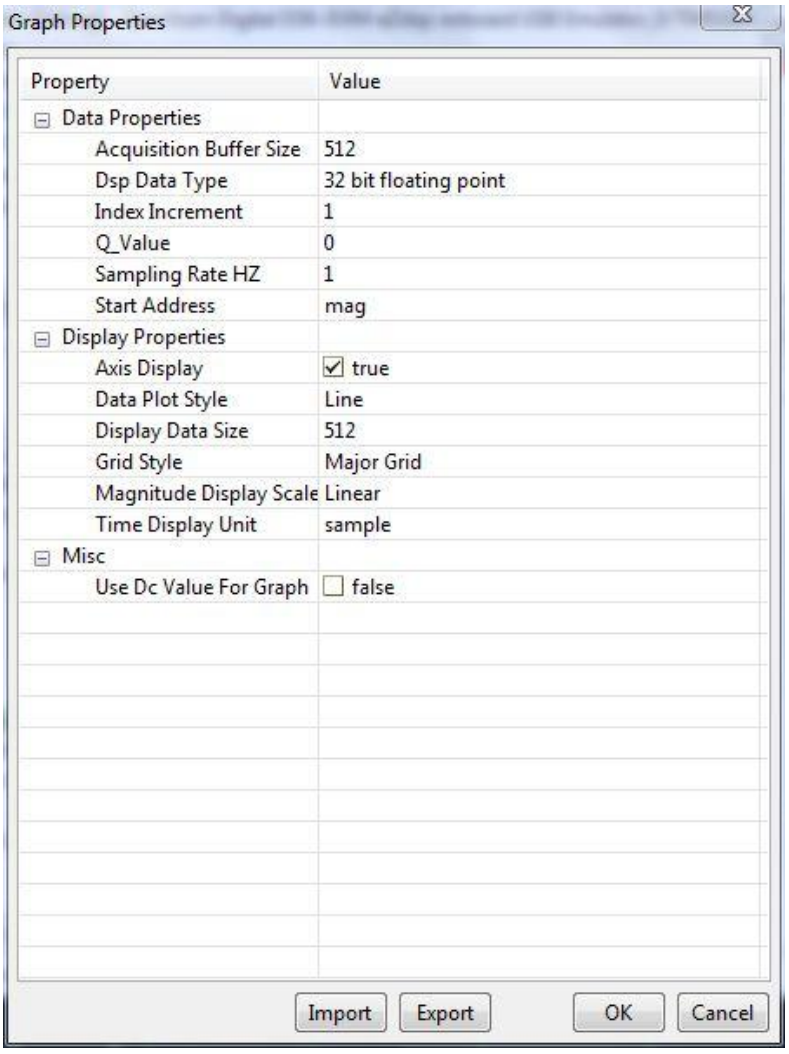


Figure 5.3 Setting graph properties for visualizing the FFT.

Then, we obtain the magnitude of the calculated FFT as in Figure 5.4.

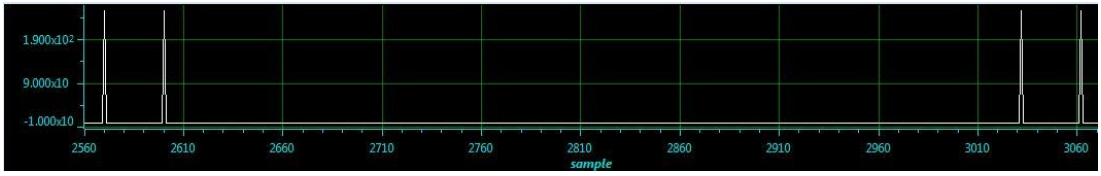


Figure 5.4 Magnitude of the calculated FFT.

We can observe the Inverse FFT (IFFT) output at the end of the program by opening the time graph and adjusting its properties as given in Figure 5.5

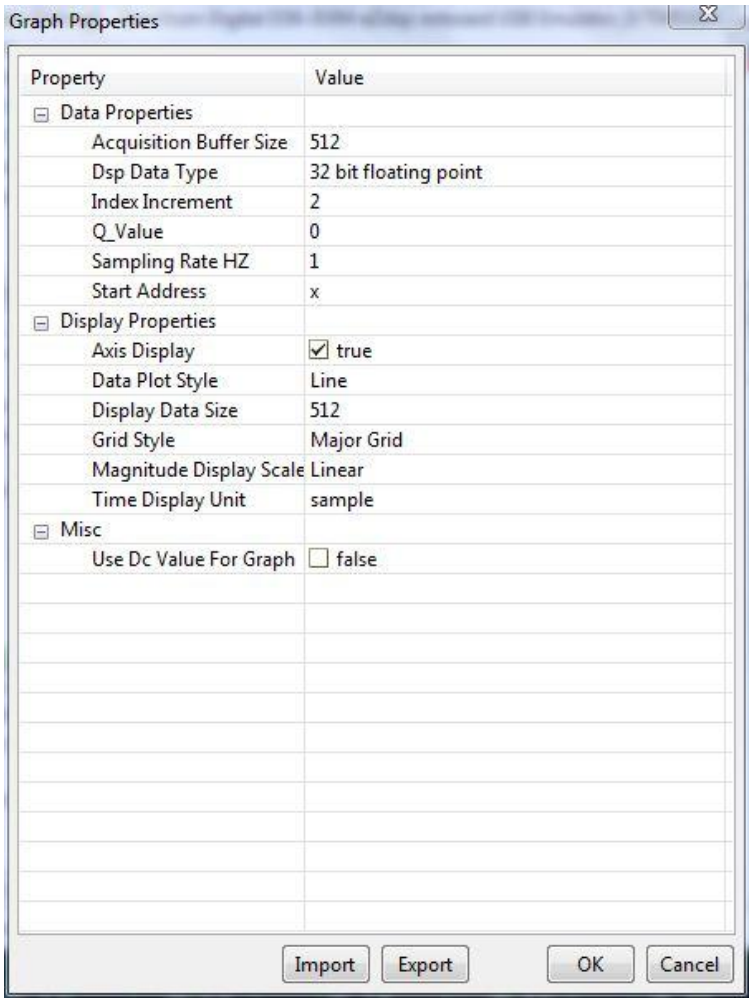


Figure 5.5 Setting graph properties for visualizing the IFFT.

Exercises

1. Create a sine signal with 256 samples having 20 Hz frequency.
    a. Plot the signal and its frequency spectrum using a suitable program.
    b. Create a header file “sine.h” and copy your samples in it.
2. Repeat Exercise 1 for a cosine signal having 30 Hz frequency.
    a. Plot the signal and its frequency spectrum using a suitable program.
    b. Create a header file “cosine.h” and copy your samples in it.
3. Use the above code (by temporarily disabling certain parts) to
    a. Read the sine and cosine signals in Exercises 1 and 2 from the header file.
    b. Plot these signals using the CCS V4.1 graph command.
4. Calculate the DFT of the signals generated in Exercises 1 and 2.
    a. Plot the magnitude of each signal separately using CCS graph commands.
    b. Save the magnitudes of the DFT results to dat files with extension “.dat” (float.dat).

c. Load the dat files on a suitable program and plot the magnitudes of DFT.

5. Repeat Exercise 4 using the FFT function.

6. Repeat Exercise 5 using your own audio signal as the input.

6. Filter Design and Implementation

The objective of this chapter is implementing digital filters on C6713 DSK. In implementation, we will choose two different ways provided by TI's DSPLib library. TI's DSPLib library can be used to implement filters in two different ways. First, the predefined filtering functions can be used. Second, the convolution operation under this library can be used. We consider both implementations next.

6.1 Filter Implementation using the Filtering Function

To implement an FIR filter, the easiest way is using TI's DSPLib library functions. Therefore, first the DSPLib should be included to the C code for filtering. The FIR filter prototype for the filtering function is as follows:

void DSPF_sp_fir_gen (const float *x, const float *h, float * restrict r, int nh, int nr)

Here

'x' is the input floating-point array.
'h' is the coefficient floating-point array.
'r' is the output array.
'nh' is the number of coefficients.
'nr' is the number of output values.

This function is an implementation of a block FIR filter. There are 'nh' filter coefficients, 'nr' output samples, and 'nh+nr-1' input samples. FIR filter coefficients should be placed in the 'h' array in reverse order {h(nh-1), ... , h(1), h(0)} . The array 'x' starts at x(-nh+1) and ends at x(nr-1). Below, you can find a section of the C code for filtering a signal.

```
#define nr 200
#define nh 240

float x[nr+nh-1];
```

```
const float r[nr];
int i;
float dummy;

/*Change the filter coefficient orders from left to right*/
for(i=0;i<nh/2;i++)
{
        dummy=h[nh-i];
        h[nh-i]=h[i];
        h[i]=dummy;
}

/*DSPLIB FIR filtering function*/
DSPF_sp_fir_gen(x,h,r,nh,nr);
```

6.2 Filter Implementation using the Convolution Function

The second method to implement an FIR filter is by using the discrete convolution operation. In time domain, the convolution of an input signal x[n] with a causal FIR filter of order *N,* represented by h[n], is given by:

$$y[n] = \sum_{k=0}^{N} h[k]x[n-k]$$

where *y[n]* is the output sequence. TI's DSPLib library can be used for this purpose also. The convolution function prototype is as follows:

void DSPF_sp_convol (float *x, float *h, float *r, int nh, int nr)

Here

'x' is the input floating-point array.

'h' is the coefficient floating-point array. Pointer to real input vector of size nh in forward order. h typically contains the filter coefficients.

'r' is the output array.

'nh' is the number of coefficients. nh ≤ nr. Nh must be a multiple of two.

'nr' is the number of output values. nr must be a multiple of four.

This function calculates the full-length convolution of real vectors 'x' and 'h' using time-domain techniques. The result is placed in real vector 'r'. It is assumed that input vector 'x' is padded with nh-1 zeros in the beginning and end. It is also assumed that the length of the input vector 'h', 'nh', is a multiple of two and the length of the output vector 'r', 'nr', is a multiple of 4. 'nh' is greater than or equal to four and 'n'r is greater than or equal to 'nh'. The routine computes four output samples at a time. 'x' and 'h' are assumed to be aligned on a double word boundary.

Exercises

1. Design 127th order FIR equiripple filters choosing filter parameters as
    a. Low-pass filter: Fs = 48 kHz, Fpass = 500 Hz, Fstop = 1 kHz
    b. Band-pass filter: Fs = 48 kHz, Fstop1 = 1 kHz, Fpass1 = 1.5 kHz, Fpass2 = 6.5 kHz, Fstop2 = 7 kHz.
    c. High-pass filter: Fs = 48 kHz, Fstop = 7 kHz, Fpass = 8.5 kHz
    d. Draw the magnitude and phase responses of your filters.
    e. Save your filter coefficients to a header file (to be called from C6713 DSK).
2. Create a project for audio signal filtering
    a. The code should take the input signal from the buffered array named “voice_in”. It should feed the filtered output signal of the three filters to the variables named “voice_filtered1”, “voice_filtered2” and “voice_filtered3” respectively.
    b. Use TI’s predefined filtering function to implement filters.
    c. Plot the FFT magnitude graphs of the “voice_in”, “voice_filtered1”, “voice_filtered2” and “voice_filtered3”.
3. Repeat Exercise 2 using TI’s predefined convolution function.
4. Repeat Exercise 2 by implementing your own convolution function.
5. Design an equalizer using your filters (in Exercise 2) and GEL files.

7. Image Processing using C6713 DSK

This chapter deals with external memory handling of the C6713 DSK. We will explore this property through basic image processing applications.

7.1 External Memory

A grayscale image is a two-dimensional array. Each element in the array is a number which represents the sampled intensity. The samples are named as pixels (picture element). Since the image is a two dimensional array, the memory usage is an important concept for the DSP implementation. For an image having a dimension of 256x256 pixels with each pixel quantized to 8 bits, the total size is 256x256x8 = 65.536 Kbytes. The internal memory of the TMS320C6713 DSP is not sufficient to store this data. Therefore, external memory space is needed. C6713 DSK has 16MB of external memory on board. To use this memory, we should setup DSP/BIOS Memory properties. Open the “DSP/BIOS configuration file” (xxx.cdb) for your project. Under the "system" tab, right click and select properties of the "MEM-Memory Section Manager". Under "compiler sections" tab, change "C Variables Section (.far)" to SDRAM. Also change the "Data Initialization Section (.cinit)" to SDRAM as in Figure 7.1.

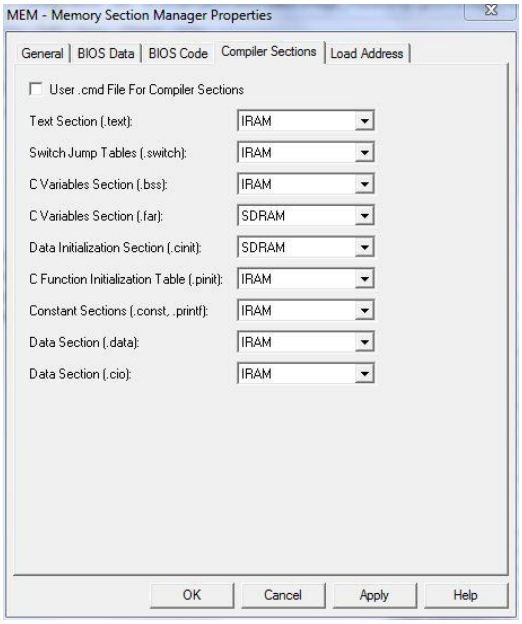


Figure 7.1 Memory configuration.

7.2 Image Processing on C6713 DSK

The C6713 DSK does not have a peripheral to directly acquire images from a camera. Therefore, we feed the image to the DSK as a header file. As we mentioned previously, we should also use the external memory to process the image. Next, we provide a section of a C code to process an image.

```
#include "image.h"

/* reading the image from the header file */

#define ROW 256
#define COLUMN 256
#define NSIZE ROW*COLUMN

/* use the far definition to use the external memory */
far int imtresh[ROW][COLUMN];
far int imres[128][128];
int hist[256];

void main(void)
{
/* add your histogram calculation function here */
/* add your image thresholding function here */
/* add your image resizing function here */

while(1);
}
```

7.2.1 Image Histogram Formation

An image histogram is the representation of the intensity distribution in a digital image. It indicates the number of pixels for each intensity value. To implement the histogram function as a C code, we should first form an array of integers with the size of grayscale levels in the image. Then, each pixel should be checked in the image. The array element indexed by the pixel's grayscale value should be increased by one. This process should be repeated until all the pixels are checked in the image. The calculated array will be the histogram of the image.

7.2.2 Image Segmentation by Thresholding

Thresholding is the simplest image segmentation method. It is often used to see which areas of the image consists of pixels above (or below) specific values. To implement thresholding on a C code, each pixel should be checked. If it has a value higher than the threshold level, it should be set to one otherwise to zero.

7.2.3 Image Resizing

Resizing an image to a smaller size is called downsampling. Resizing an image to a larger size is called upsampling. To implement a simple downsampling operation on a C code, one method is as follows. Take the required number of neighbor pixels and the set the average of them as the new pixel value of the downsampled image.

Exercises

1. Pick a 256x256 grayscale image. Form the “image.h” header file from this image.
2. Create a project to process the image in Exercise 1.
    a. Make all adjustments.
    b. Read the image from Exercise 1.
    c. Show the image using CCS V4.1 graph functions.
3. Extend the project in Exercise 2 to calculate the histogram of the image.
    a. Plot the histogram on CCS V4.1.
    b. Save the histogram array and read it using another program.
4. Extend the project in Exercise 2 to threshold the image.
    a. Show the thresholded image using CCS V4.1 graph functions.
    b. Save the thresholded image and read it read it using another program.
5. Extend the project in Exercise 2 to downsample the image by half in each coordinate.
    a. Show the downsampled image using CCS V4.1 graph functions.
    b. Save the downsampled image and read it using another program.

8. Using Interrupts

This chapter deals with how to define a task on DSP/BIOS and how to configure the timers and hardware interrupts. As a practical example, we consider sliding LEDs.

8.1 Sliding LEDs

As the application of this chapter, the four LEDs on the C6713 DSK will blink consecutively to forward and backward as shown in Figure 8.1. The sliding speed will be controlled by DIP switches on the DSK. You will control the speed by counting the interrupt invoked at each period of Timer 1.

■□□□
□■□□
□□■□
□□□■
□□■□
□■□□
■□□□

Figure 8.1 Sliding LEDs in time.

You will define the sliding algorithm as a task that will run independently from the main function. The main function will initialize the program. When the initialization is complete, the CPU will become idle and will start to run the LED sliding task.

8.2 Procedure

Create a new project file named “Sliding_LEDs”. Create a new DSP/BIOS configuration file in your project. To process periodic hardware interrupts from the Timer 0, you need to tie this interrupt to a periodic hardware interrupt service routine. Open the "Scheduling" section of the configuration window and right click to the “PRD-Periodic Function Manager” subsection as in Figure 8.2. Click on to “Insert PRD”.

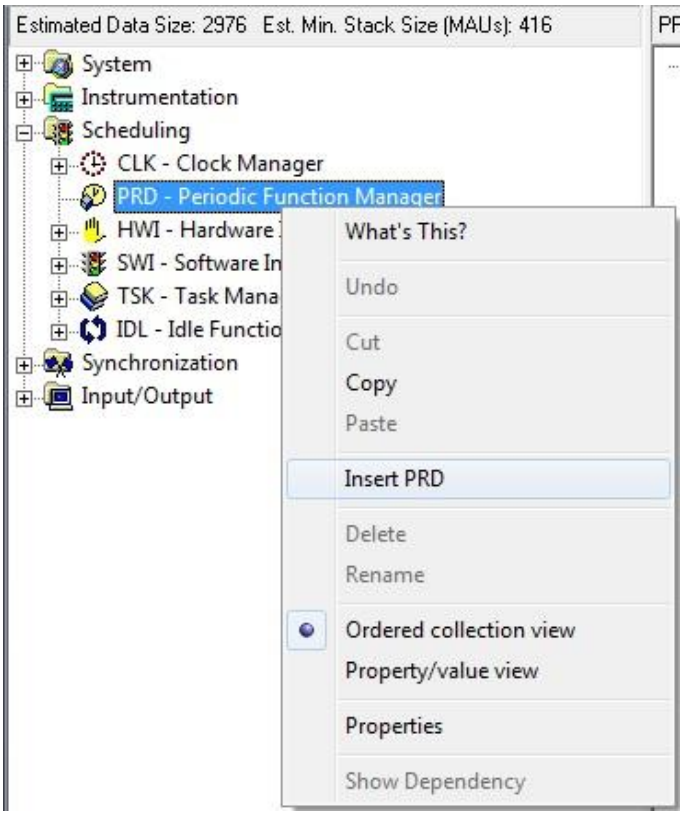


Figure 8.2 Scheduling section of the configuration window.

Open the properties window of PRD0 as in Figure 8.3. Set the "Periodic ticks" to 10 and the "timer ISR function" to "_timer_isr". Note the use of a leading underscore, as this interrupt service routine (ISR) is written as a C function. The leading underscore is required, because the C compiler places a leading underscore to all C symbols during compilation.

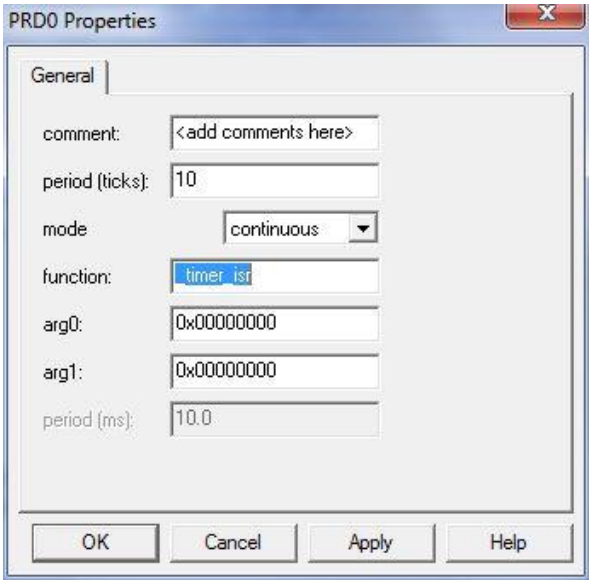


Figure 8.3 PRD configurations.

At this point, we have configured the PRD on the chip. Finally, we will configure the DSP/BIOS configuration file to manage the LED sliding operation. Open the "Scheduling Section" of the "Configuration Window". Right-click on the "TSK-Task Manager", and insert a task called “tsk0” as in Figure 8.4.

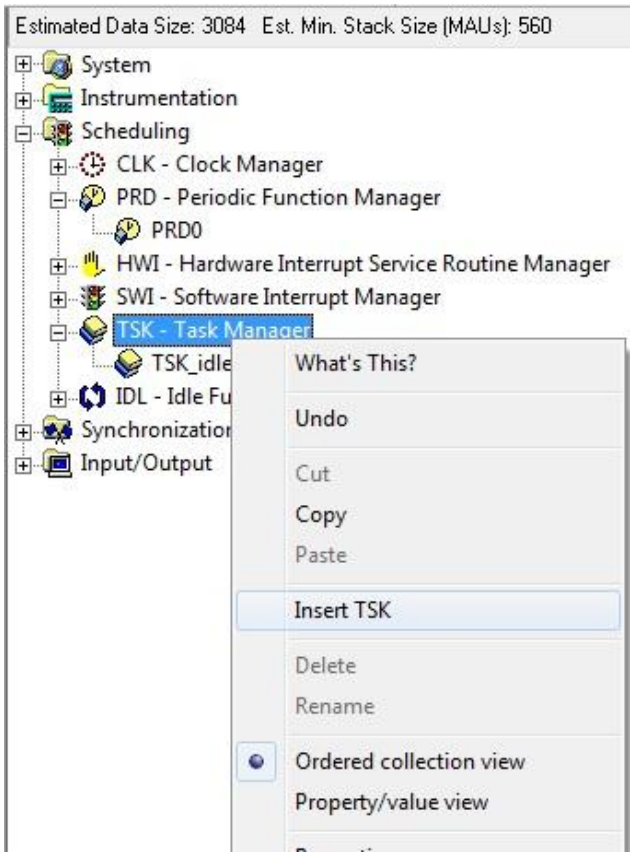
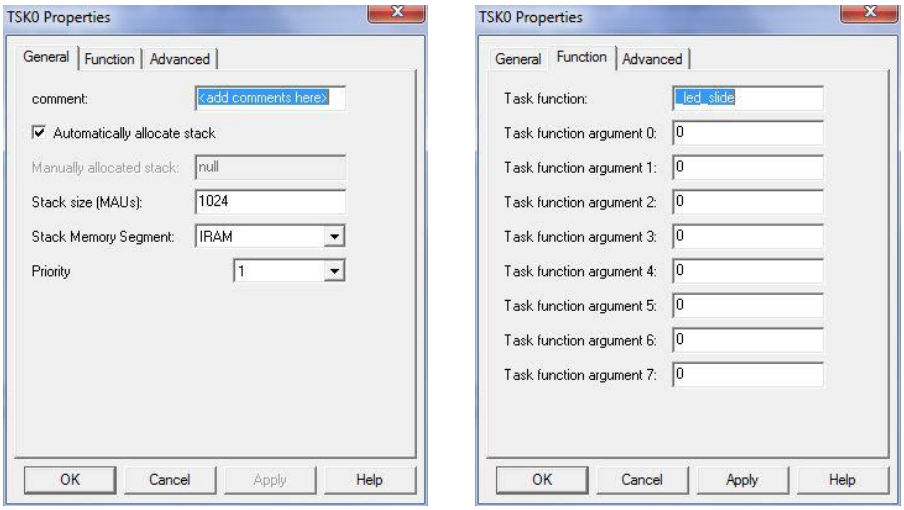


Figure 8.4 Inserting a task.

Right-click and select the properties of the new task. In the properties window, set the “Priority” field to “1” under the “General” tab as in Figure 8.5 (a). Under the “Function” tab set the “Task function” to “_led_slide” as in Figure 8.5 (b). Don’t forget to place the leading underscore!



(a) Setting priority **(b) Setting task function**

Figure 8.5 Setting task properties.

We have completed our DSP/BIOS configuration. Now, it's time to place the necessary lines to your source code to control and get the interrupts from the Timer 0. Create the source file named “Sliding_LEDs.c”. Place the header files that is needed to initialize the C6713 DSK as shown below.

```
#define CHIP_6713 1
#include "dsk6713.h"
#include "dsk6713_led.h"
#include "dsk6713_dip.h"
```

We will use the timer to control the delay between each LED blinking. The timer ISR will be invoked every 10 ticks of the timer clock. The following calculations lead to

TMS320C6713 CPU clock period = 1/225 MHz = 4.44 ns

Time period for timer 1 interrupt = 4.44*10 = 44.4 ns

This means the timer will give an interrupt every 44.4 ns. We can control the amount of the delay by counting the number of interrupts. As the counter, declare a global variable named “ttick”. In the file “Sliding_LEDs.c”, define the ISR function as shown below. This will increase the value of the “ttick” on each of the interrupts of the timer period.

```
void timer_isr(void)
{
        ttick++;
        return;
}
```

To initialize the C6713 DSK, LEDs, and DIP switches, your main() function should be as follows.

```
void main()
{
        /*DSK initialization*/
        DSK6713_init();
        DSK6713_LED_init();
        DSK6713_DIP_init();
        ttick=0;

        /* fall into DSP/BIOS idle loop */
        return;
}
```

Now, place the LED sliding task function into the “Sliding_LEDS.c” file. The function will read the DIP switch to select the number of timer periods to wait on each LED blink. There are seven states on the LED sliding algorithm. Therefore, the sliding loop will be controlled by a loop counting from 0 to 6. According to the state of the loop, the corresponding LED will become “on”. The code of the slider task is as follows.

```
/*led slider task*/
void led_slide(void)
{
        int i;
        int speed;
        while(1)
        {
                /*Select the speed by DIP switches*/
```

```
/* By choosing  the number of the interrupts to wait*/
if(DSK6713_DIP_get(0) == 0)
                {
                        speed=500;
                }
                if(DSK6713_DIP_get(1) == 0)
                {
                        speed=2;
                }
                else
                {
                        speed=5;
                }

                for(i=0;i<7;i++)
                {
                        if(i<4)
                        {
                                ttick=0;
                                while(ttick<speed)
                                {
                                        DSK6713_LED_on(i);
                                }
                                DSK6713_LED_off(i);
                        }
                        else
                        {
                                ttick=0;
                                while(ttick<speed)
                                {
                                        DSK6713_LED_on(6-i);
                                }
                                DSK6713_LED_off(6-i);
                        }
                }
```

```
        }
}
```

Exercises

1. Build and run your project and observe the sliding LEDs.
2. Change the speed of the sliding LEDs by using DIP Switches.
3. Calculate the number of the Timer periods needed to blink the LEDs every 3 seconds.

9. Further Reading

For more advanced topics on C6713 DSK, the reader can check the reading list given below. Also up to date information on the DSK can be obtained from Texas Instrument's web site.

Further reading list

1. R. Chassaing and D. Reay, *Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK*, Wiley-IEEE Press, Second Edition, 2008
2. S. Qureshi, *Embedded Image Processing on the TMS320C6000 DSP*, Springer, 2005
3. N. Kehtarnavaz, *Real-Time Digital Signal Processing based on the TMS320C6000*, Elsevier Publishing, 2004
4. S. A. Tretter, *Communication System Design using DSP Algorithms: With Laboratory Experiments for the TMS320C6713 DSK*, Springer, 2008