Department of Electrical and Electronic Engineering
Imperial College London

EE3-19

Real Time Digital Signal Processing

Course homepage: http://learn.imperial.ac.uk

Lab I - Getting started with the TI C6x DSP: Becoming familiar with Code Composer Studio and the DSP Starter Kit

Paul D. Mitcheson
paul.mitcheson@imperial.ac.uk
Room 1112, EEE

Objectives

- Introduce Code Composer Studio (CCS) and familiarise yourselves with the DSP kit.
- Develop and run a simple program and understand that programs are managed in the context of a project.
- Familiarise with C and assembly syntax.
- Change project options and fix a simple syntax error.
- Use watch windows, breakpoints and File I/O.
- Learn how to measure the number of execution cycles for sections of code.
- Help you to avoid some of the common mistakes that are made when working with CCS.

Conventions used in this document and (subsequent labs)

- Arial text formatted black but not bold such as: MAXGAIN or nop 5 denotes text taken from C, ASM or H files.
- Arial text formatted black and bold such as **How does..?** denotes a question for you to answer.
- Courier text such as: `0 Errors, 0 Warnings, 0 Remarks` denotes a report from a CCS window.
- Times New Roman text, formatted bold and italic such as: ***volume.asm*** denotes a file, directory or the name of a project.
- Times New Roman text, formatted bold such as: **Project→Rebuild All** denotes navigation (e.g. menu or button or an option to set). This text is also used for anything you need to type into a dialog box.
- Texas Instruments, in their literature, refer to this DSP hardware as a **DSK** (**D**igital signal processing **S**tarter **K**it) or sometimes simply the “Target”. This document and subsequent ones will follow the same practice.

Introduction

In this lab you will get an introduction to Code Composer Studio (CCS) the software tool you will use in future labs for developing and running C6x DSP programs.

This lesson uses an example called ***volume1*** to develop and run a simple program. You will create a project from scratch, add files to it, and review the code. After you build and run the program, you will change build options using the build options dialog, and fix syntax errors using the editor. You will utilize basic debug techniques like breakpoints, watch windows and file I/O. Finally, you will measure the number of execution cycles for sections of your code.

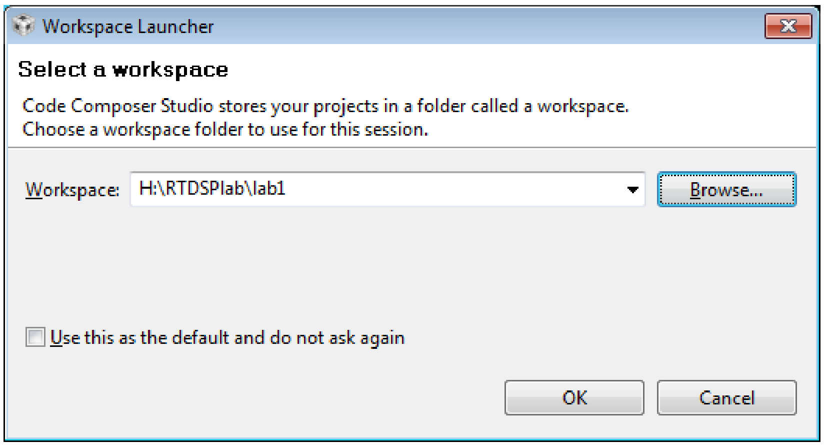
You must in pairs for lab exercises and any deliverables from labs are submitted as one report per pair. **Plagiarism will not be tolerated**. This lab is a bit different in that it is online and does not count towards your assessment. The questions spread throughout this lab document in bold type are unassessed but you should ask a lab demonstrator if you can't figure them out – and they are indicative of the type of questions in the Blackboard test. For lab 1 you could either take the test alone or with your lab partner. Lab 2 is marked as if it were to be assessed but marks are not counted (it allows you to calibrate your report style against the marking). For lab 2 hand in 1 report per lab pair. Labs3-5 are assessed through handing in electronic copies of your reports (which will include code snippets) as one report per pair.

Creating a Workspace

The workspace is the main working folder for CCS and where it stores project information to manage all the projects that you define to it. A workspace can have many projects defined within it but for the purposes of this lab and subsequent ones we have decided that you should create a workspace for each individual lab.

- Create a folder on your H drive called ***RTDSPlab*** this will be the root folder of all your workspace folders.
- Start CCS by double clicking the icon on the desktop (or using the start menu).

- You are asked to select (or create a new) workspace. By typing exactly what is shown in figure 1 create a new workspace called ***lab1*** under your ***RTDSPlab*** folder. Ensure the **Use this as the default and do not ask again** check box is **NOT** checked.



Workspace Launcher

Select a workspace

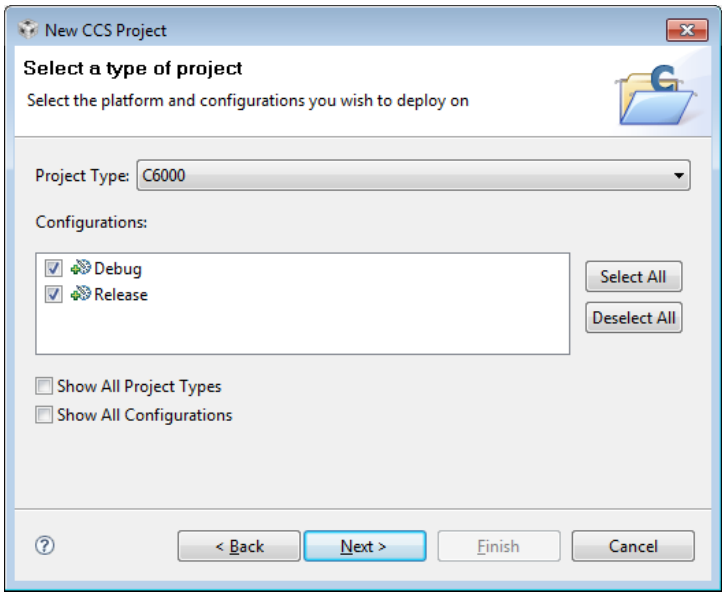
Code Composer Studio stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: H:\RTDSPlab\lab1

☐ Use this as the default and do not ask again

Figure 1: Creating a new workspace

- Hit **Ok** when done.
- Close the welcome screen that opens (by hitting the x on its tab).
- Create a project by using **File→New→CCS Project,** call it ***RTDSP*** by typing in the **Project Name** box (this also creates a ***RTDSP*** subfolder below ***RTDSPlab\lab1***)
- Hit **Next**
- Select **C6000** for the **Project Type** (see figure 2)



New CCS Project

Select a type of project

Select the platform and configurations you wish to deploy on

Project Type: C6000

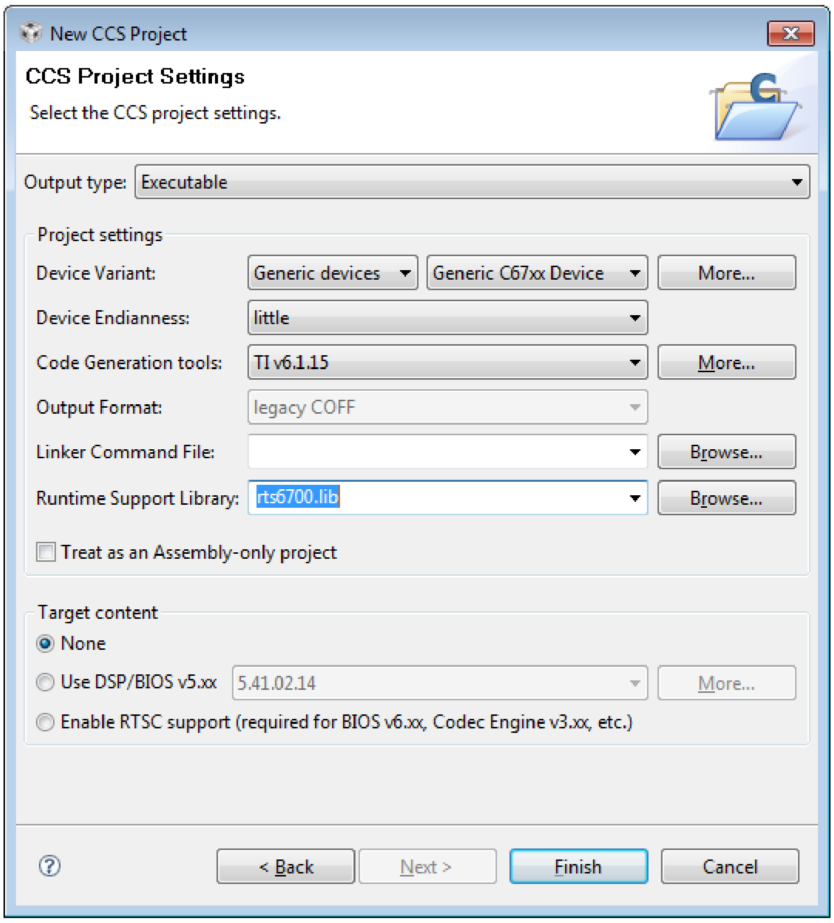
Configurations:

☑ Debug
☑ Release

☐ Show All Project Types
☐ Show All Configurations

Figure 2: Setting up a new project

- Hit **Next** twice (i.e. ignore the dialogue regarding **Additional Project Settings**.) Set up the dialogue as shown in figure 3:


New CCS Project

CCS Project Settings

Select the CCS project settings.

Output type: Executable

Project settings

Device Variant: Generic devices | Generic C67xx Device

Device Endianness: little

Code Generation tools: TI v6.1.15

Output Format: legacy COFF

Linker Command File:

Runtime Support Library: rts6700.lib

☐ Treat as an Assembly-only project

Target content

● None

○ Use DSP/BIOS v5.xx 5.41.02.14

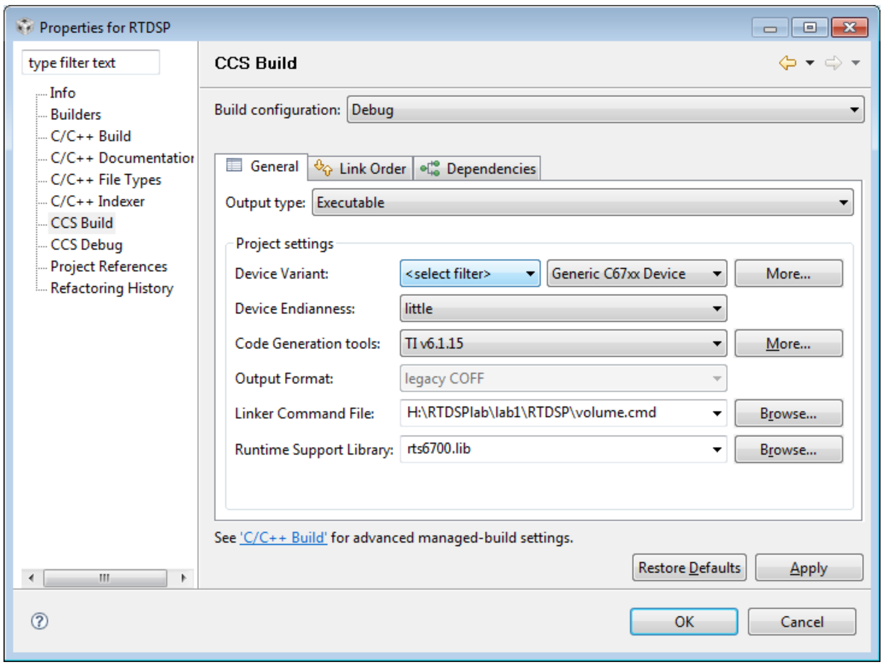
○ Enable RTSC support (required for BIOS v6.xx, Codec Engine v3.xx, etc.)

Figure 3: Final dialogue to set up project

- Hit **Finish.**

- Download the ***lab1.zip*** folder from Blackboard. Unzip the contents into ***H:\RTDSPlab\Lab1\RTDSP\*** folder. You do not have to explicitly bind the files into the project, as you may have done in other IDE's. As long as the files are in the project directory they will be bound in automatically.

- Use the menu **Project→Properties** to open the properties dialog then select **CCS Build** in the left hand tree structure. Ensure that the linker command file box is set up as of figure 4 by hitting the **Browse** button next to it and navigating and opening the file **H:\RTDSPlab\lab1\RTDSP\volume.cmd**



Build configuration: Debug

Output type: Executable

Device Variant: <select filter> Generic C67xx Device

Device Endianness: little

Code Generation tools: TI v6.1.15

Output Format: legacy COFF

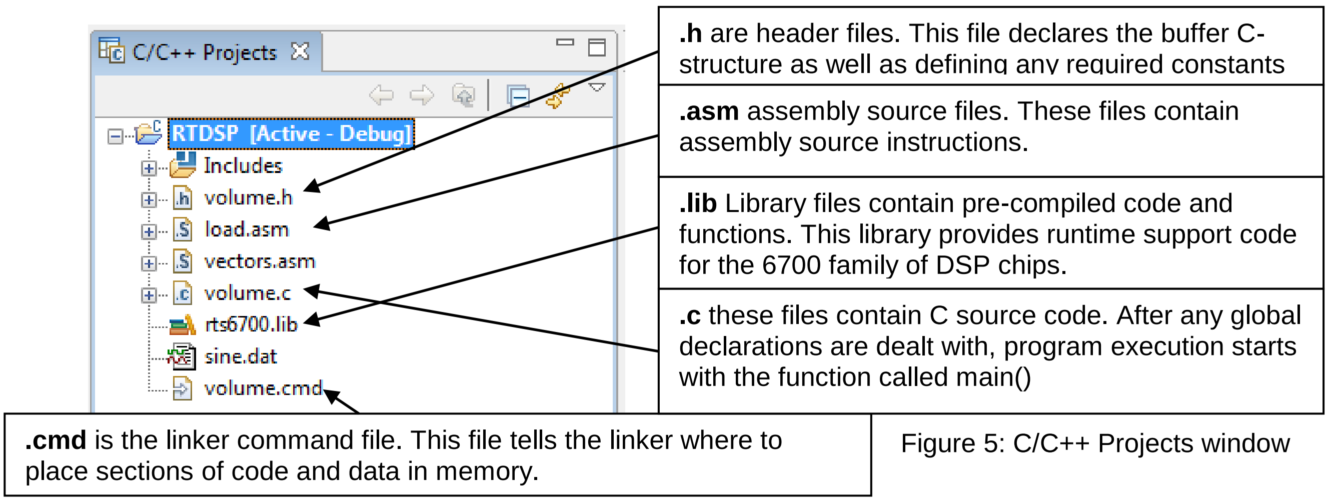
Linker Command File: H:\RTDSPlab\lab1\RTDSP\volume.cmd

Runtime Support Library: rts6700.lib

See 'C/C++ Build' for advanced managed-build settings.

Figure 4: Setting up the linker command file

- Hit **Apply** then hit **OK** to save the changes to the existing build. Then hit **OK** again to close the Properties dialogue.
- In order to navigate and see what files are held in your project you need to use the **C/C++ Projects** window. Back in the main window hit the following icon on the bottom left hand side of the screen
- In the pop up choose **C/C++ Projects**. This creates another icon on the bottom left of the screen that will allow you to reach **C/C++projects** window quickly.
- Right click **RTDSP** in the **C/C++ Projects** window, choose **Refresh**. Then click the + icon next to it. Your window should look like figure 5:



.h are header files. This file declares the buffer C-structure as well as defining any required constants

.asm assembly source files. These files contain assembly source instructions.

.lib Library files contain pre-compiled code and functions. This library provides runtime support code for the 6700 family of DSP chips.

.c these files contain C source code. After any global declarations are dealt with, program execution starts with the function called main()

.cmd is the linker command file. This file tells the linker where to place sections of code and data in memory.

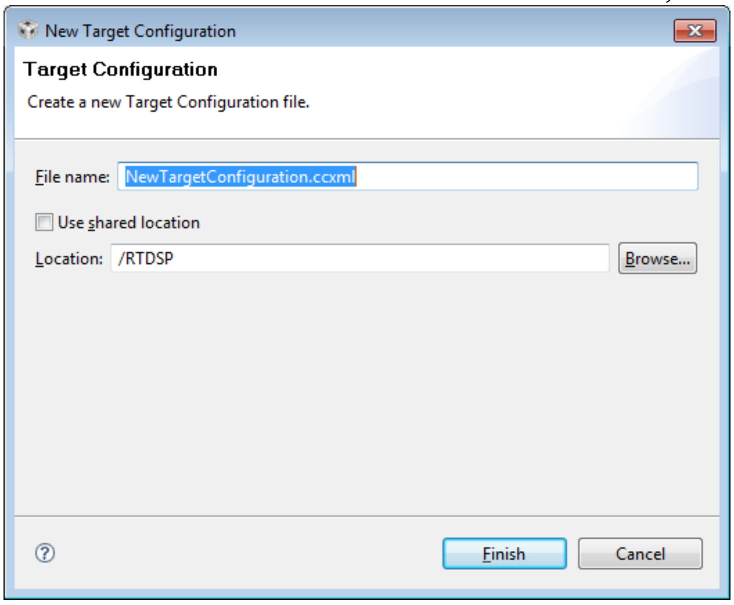
Figure 5: C/C++ Projects window

Familiarise yourself with the function of each file.

Connecting to a Target

You now need to connect to the DSK6713 hardware (which is referred to a target in CCS).

- Use the main menu to select **File→New→Target Configuration File**
- Ensure the use shared location box is **NOT** checked, the window should look like figure 6:



New Target Configuration

Target Configuration

Create a new Target Configuration file.

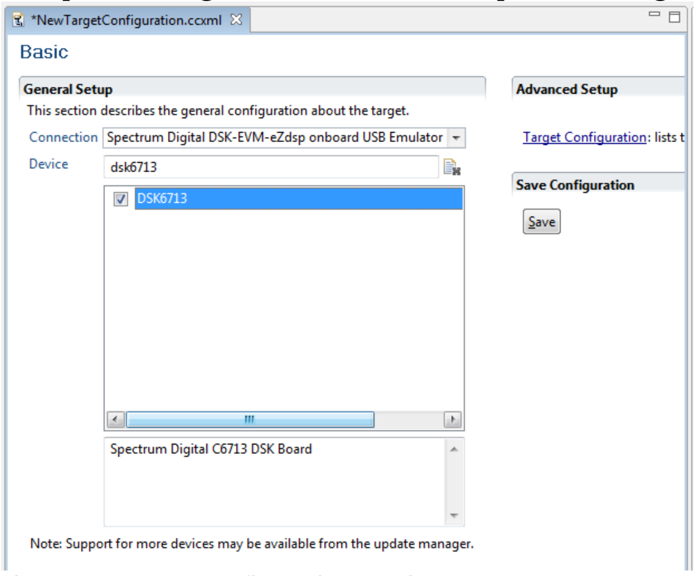
File name: NewTargetConfiguration.ccxml

Use shared location

Location: /RTDSP

Figure 6: Creating a Target configuration

- Hit **Finish.**
- Set up the configuration window that opens as of figure 7:



*NewTargetConfiguration.ccxml

Basic

General Setup

This section describes the general configuration about the target.

Connection: Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator

Device: dsk6713

DSK6713

Spectrum Digital C6713 DSK Board

Note: Support for more devices may be available from the update manager.

Advanced Setup

Target Configuration: lists t

Save Configuration

Figure 7: Target Configuration Settings

- Then hit the **Save** button on the right hand side of this window. The configuration window can then be closed if you wish.

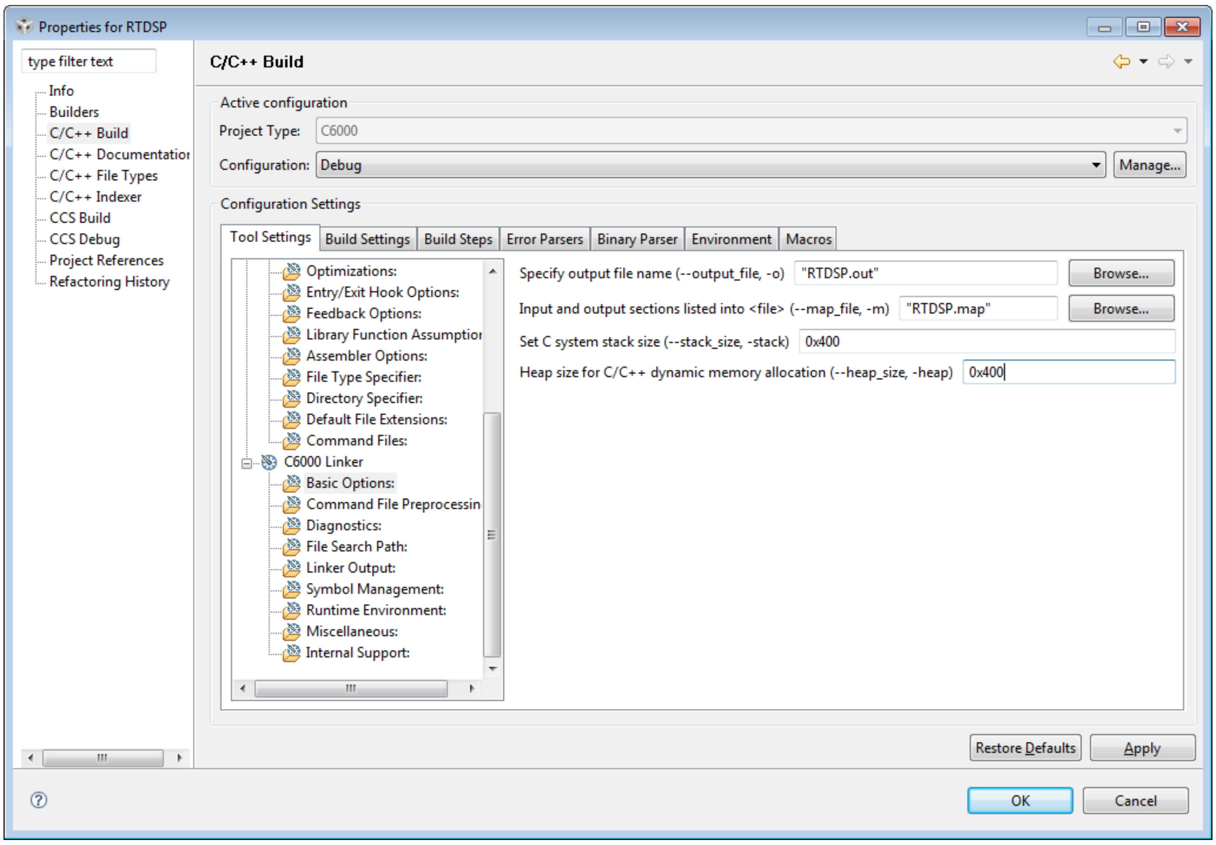
- Open up the **C/C++ Projects** window. By double clicking ***volume.c*** in this window it will open the code into a separate window so you can review it.

- Place your mouse cursor in the code window. Now select **Project→Properties** from the main menu. The **Properties** dialogue will open (note that placing your mouse cursor somewhere in your code is necessary as the **Properties** option in the menu is greyed out otherwise).

- In the **Properties** dialogue select **C/C++ Build** in the tree structure then select **Basic Options** from the **C6000 Linker** section.

- Set the **Stack** and **Heap** to **0x400** as of figure 8

- Hit **Apply** then **OK**



Active configuration
Project Type: C6000
Configuration: Debug

Configuration Settings

Specify output file name (--output_file, -o) "RTDSP.out"
Input and output sections listed into <file> (--map_file, -m) "RTDSP.map"
Set C system stack size (--stack_size, -stack) 0x400
Heap size for C/C++ dynamic memory allocation (--heap_size, -heap) 0x400

Figure 8: Build settings

Reviewing the Source Code

As you have seen double clicking on any code file shown in the **C/C++ project** view opens a text editor with the contents of the file. If not already open double-click on the ***volume.c*** file in the **C/C++ project** view to open the source code in the right half of the Code Composer Studio window. Maximise this window.

Note the following functions within the code:

- After the main() function prints a message, it enters an infinite loop. Within this loop, it calls the dataIO() and processing() functions. (Ignore the statement wrapped with #ifdef and #endif as it is not executed at the moment. more on this later).
- The processing() function multiplies each value in the input buffer by the gain and puts the resulting values into the output buffer (buffers are implemented using arrays). processing() then calls the assembly load() routine, which consumes instruction cycles based on the processingLoad value passed to the routine.
- The dataIO() function in this example does not perform any actions other than to return. Rather than using C code to perform I/O, we will use a breakpoint to read data from a file on the host into the inp_buffer location.

Make sure you understand how ***load.asm*** and ***volume.c*** work by opening the files into CCS and reading the comments. (You will have to use the **C/C++ Project** window, **Maximize**/**Minimize**/**Restore** buttons to navigate between the various files).

Questions

1. **Why do some of the statements in the assembly code *load.asm* simply consist of a NOP statement? What will happen if these statements are removed?**

2. **What is the purpose of the -- and ++ operators in the following code:**

```
while(size--){
      *output++ = *input++ * gain;
}
```

3. **In the code above what are the two purposes of the * operator?**

4. **What is the difference between how the ++ operator works with an Integer variable such as size and a pointer to an integer variable such as input?**

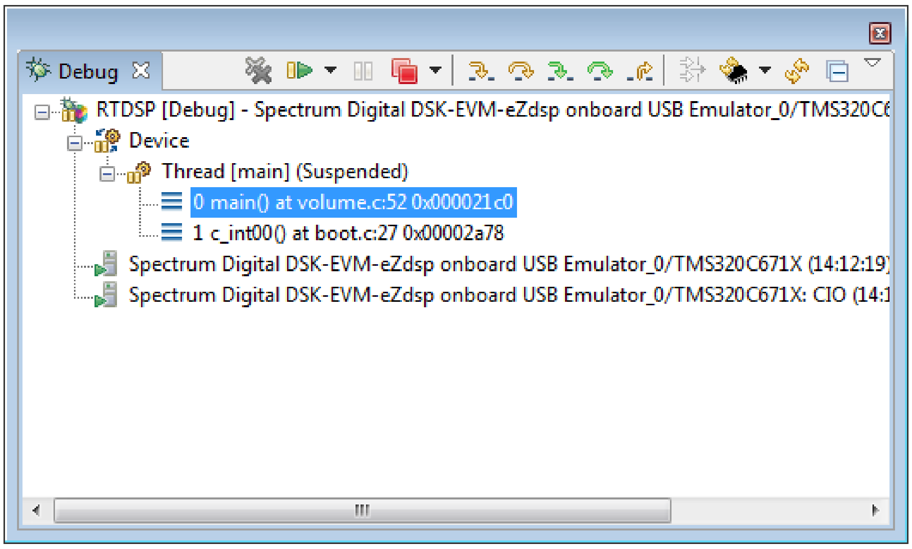
5. **What is the size of the I/O buffers in hex and denary?**

6. **By what gain value will the input signal be amplified when the program is initially run?**

7. **If the cycle time for the 6713 processor is 4.4ns how much time is taken for each call to load() if the argument of the function call (processingLoad) is set to 30? (Do not include any time due to code for setting up the loop and returning from the function).**

Build and Run the Program

- Choose **Project→Rebuild Active Project**. This invokes CCS to recompile, reassemble, and re-link all the files in the project. The **Console** window and the **Problems** window that appear display messages about this process.
- A successful compilation (i.e. the **Console** window reports: `Finished building target: RTDSP.out`) will result a file called ***.out*** being created. Note also for success there will be no errors reported in the **Problems** window.
- Plug the PSU connector into J5 of the DSK6713 hardware and ensure that the +5 volt LED is lit (re-cycle the power if the DSK is already connected). Then connect the USB cable to the PC.
- Hit the **Debug** icon in the toolbar. This will connect to the hardware, load the ***.out*** file onto the DSK and open a debug window (figure 9). If you make any changes to your code you must re-compile and then load the ***.out*** file onto the DSK in order to see the affects of the changes you have made. It is important to remember that the code runs on the DSK hardware and not on the PC!



```
RTDSP [Debug] - Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator_0/TMS320C6
  Device
    Thread [main] (Suspended)
      0 main() at volume.c:52 0x000021c0
      1 c_int00() at boot.c:27 0x00002a78
  Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator_0/TMS320C671X (14:12:19)
  Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator_0/TMS320C671X: CIO (14:1
```

Figure 9: The debug window

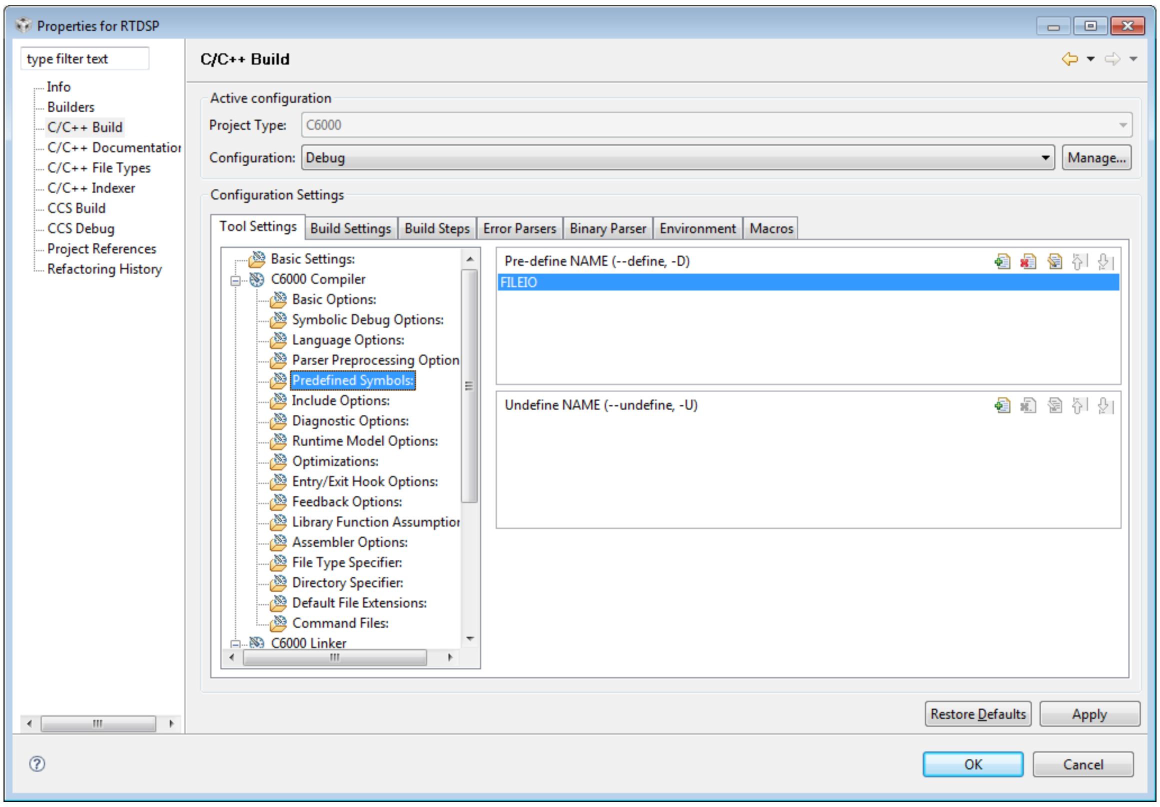
- You can detach the **Debug** window if you wish by right clicking its tab and choosing **Detached**. Hit the run icon (or use the menu **Target→Run**.) The text `volume example started` should appear in the **Console** window.
- Hit the halt icon (or use the menu **Target→Halt**) to quit running the program.

Changing Program Options and Fixing Syntax Errors

Pre-processor statements (denoted by beginning the statement with a #) are not C statements[1] and are executed exclusively by the compiler, hence such statements do not produce any machine code per se (although their side affects do!)
C statements enclosed by #ifdef and #endif pre-processor statements are optionally compiled depending on the absence/presence of a symbol.

Find the code in ***volume.c*** enclosed by the pre-processor commands (#ifdef and #endif)
This line of C code (which has a syntax error) did not compile and hence run because the symbol FILEIO was undefined.

- Put the cursor in the code window for **volume.c** then choose the menu **Project→Properties**. Click **C/C++ build** in the tree diagram. In the **C6000 compiler** section choose **Predefined Symbols**. Using the add button Type **FILEIO** in the **Pre-Define NAME** field (figure 10) Click **OK**. The C code after the #ifdef FILEIO statement in the program will now be compiled when you recompile the program. Click **Apply** then **OK**


Properties for RTDSP

C/C++ Build

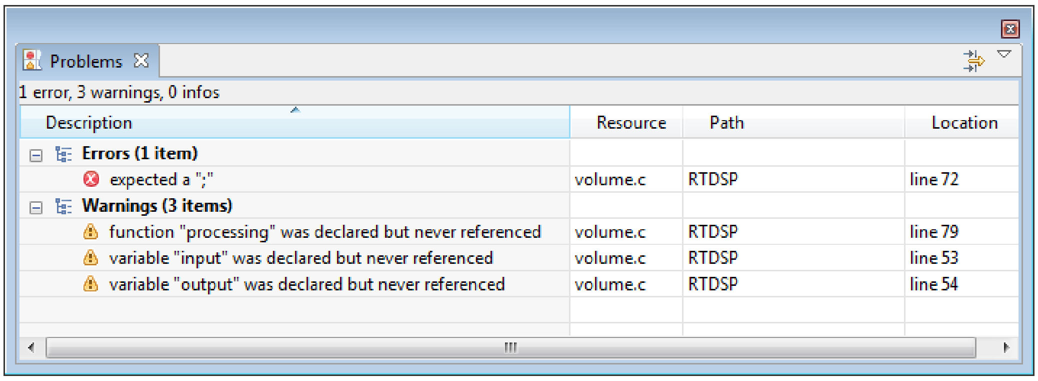
Active configuration
Project Type: C6000
Configuration: Debug

Configuration Settings
Tool Settings: C6000 Compiler → Predefined Symbols:
Pre-define NAME (--define, -D): FILEIO
Undefine NAME (--undefine, -U):

Figure 10: Properties window

[1] A common mistake is to incorrectly end a pre-processor statement with a colon (;) (the C syntax for end of statement) for example: #define PI 3.14; is incorrect! It should be #define PI 3.14

- Choose **Project→Rebuild Active Project.** The problems frame now shows some errors/warnings. A build message appears at the top of this frame `1 Error, 3 Warnings, 0 Infos` and below this each problem is detailed (see figure 11).



Problems

1 error, 3 warnings, 0 infos

Description	Resource	Path	Location
Errors (1 item)			
expected a ";"	volume.c	RTDSP	line 72
Warnings (3 items)			
function "processing" was declared but never referenced	volume.c	RTDSP	line 79
variable "input" was declared but never referenced	volume.c	RTDSP	line 53
variable "output" was declared but never referenced	volume.c	RTDSP	line 54

Figure 11: Syntax error reporting

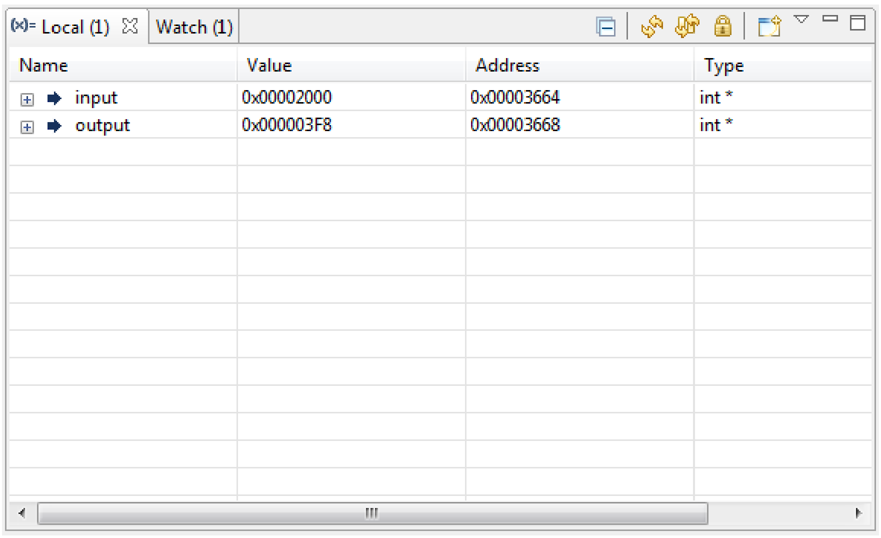
- Double-click on the line that describes the location of the first syntax error: `expected a ";"`. Notice that the ***volume.c*** source file opens, and your cursor should be on the following line: processing(input, output);
- Fix the syntax error in the line above the cursor location (The semicolon is missing.) It should look like this: puts("begin processing");
- Notice that an asterisk (*) appears next to the filename in the Edit window's title bar, indicating that the source file has been modified. The asterisk disappears when the file is saved.
- Choose **File→Save** or press **Ctrl+S** to save your changes to ***volume.c***.
- Choose **Project→Rebuild Active Project** to rebuild updated files. If you fixed the error successfully you will be prompted to reload the code to the DSK.
- Click yes to reload the ***.out*** file.
- Choose **Target→Go to main** to begin execution from the main function. (The Target menu contains the options given in the debug window and some extra options). The execution halts at the opening brace to main (the cursor position is identified by a small blue arrow in the margin of the code window).
- Make sure the **console** tab is in view then choose **Target→Run**. (or use the icon in the debug window). The text `"volume example started"`, followed by `"begin processing"` should appear in the Console window.

- Choose **Target→Halt** to quit running the program. (If you wish to re-run the program click **Target→Restart** followed by **Target→Go to main** before running to reset the cursor).

Using Breakpoints and the Watch Window

When you are developing and testing programs, you often need to check the value of a variable during program execution. In this section, you will use breakpoints and the Watch Window to view such values. You will also use the step commands after reaching the breakpoint.

- Choose **Target→Reload Program**.
- Double-click on the tab of the ***volume.c*** code window (this maximises the window).
- Right click the line that says: dataIO(); in the pop up that appears choose **New Breakpoint→Breakpoint** This will set a software breakpoint. An icon indicates that a breakpoint has been set.
- Choose **View→Watch** A watch window appears as a tab next to the Local watch window. At run time, these tabs show the values of watched variables. Select the Locals tab. This tab shows local variables that are local to the executed function.
- If the cursor is not at main, choose **Target→Go to main**. Why are the variables input and output shown in the locals watch window (figure 12)?



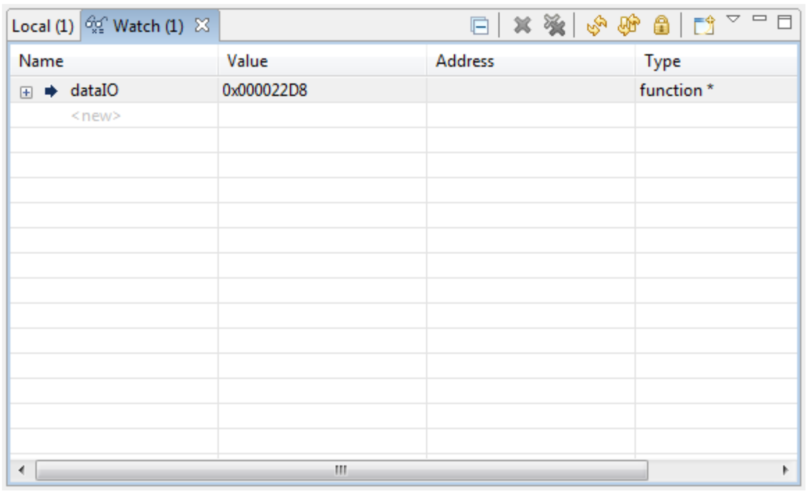
Name	Value	Address	Type
input	0x00002000	0x00003664	int *
output	0x000003F8	0x00003668	int *

Figure 12: Local watch window

- Choose **Target→Run**.
- Select the **Watch(1)** tab.
- Click in the **Name** column (where it says <new>) and type **dataIO** as the name of the variable to watch.

- Hit return to save the change. An address value[2] should immediately appear (figure 13).

[2] The actual value (a memory address location) may vary from the one shown in this document.



Name	Value	Address	Type
dataIO	0x000022D8		function *
<new>			

Figure 13: watch window with a watch on dataIO

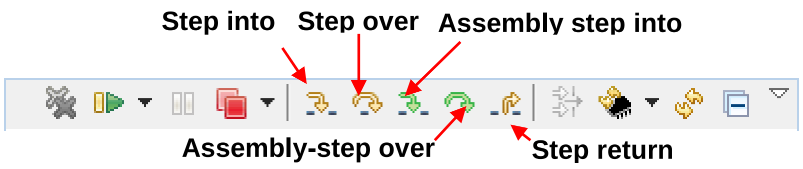
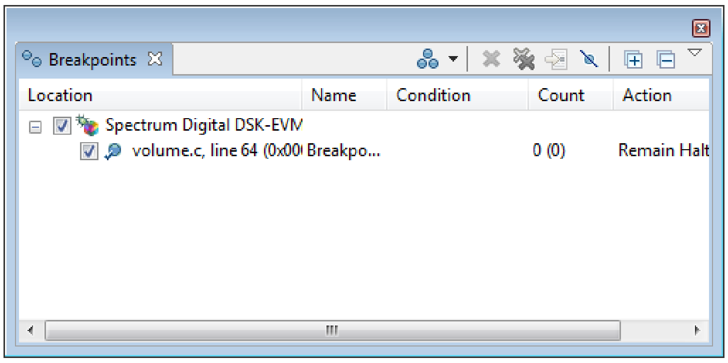


Figure 14: toolbar for debugging (these toolbar functions are also available in the **Target** menu)

- Click the (Step Over) toolbar button to step over the call to dataIO().
- Experiment with the other step and run commands in the toolbar shown in figure 14. To understand how the tools that work with assembly function you will need to open a disassembly window (**View→Disassembly**), size the C code window and the **Disassembly** window so they are fairly large and side by side, detach the **Debug** window if necessary (right click its tab, select **Detached**).
- After you have finished experimenting, click the menu **View→Breakpoints** and in the window that opens (figure 15) click (**Remove All Breakpoints**) before proceeding to the next part of the lesson. Close the breakpoint window.



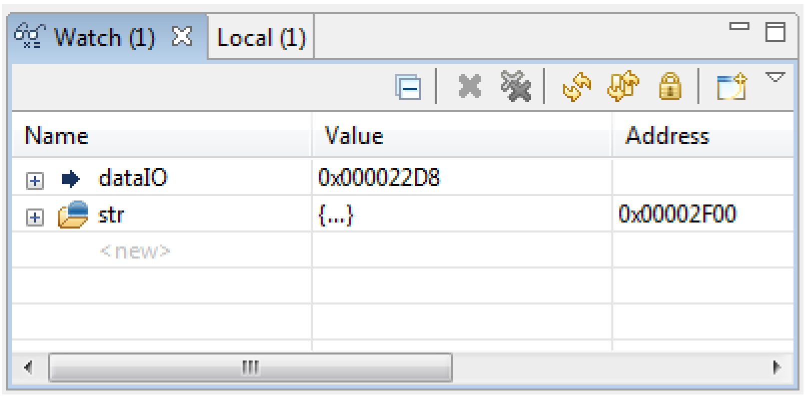
Location	Name	Condition	Count	Action
Spectrum Digital DSK-EVM				
volume.c, line 64 (0x00	Breakpo...		0 (0)	Remain Halt

Figure 15: Breakpoints window

Using the Watch Window with Structures

In addition to watching the value of a simple variable, you can watch the values of the elements of a structure.

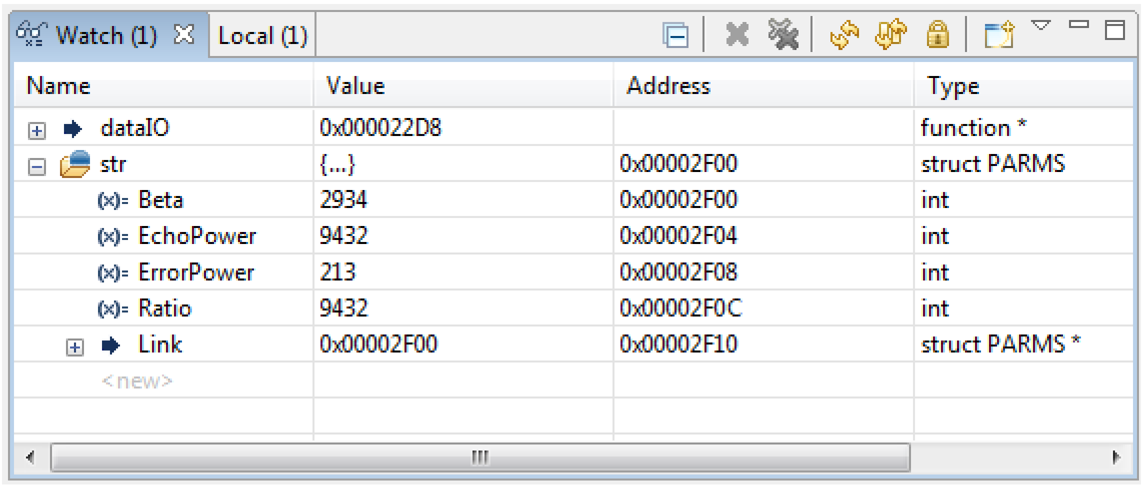
- Select the **Watch1** tab.
- Click on the expression icon in the **Name** column where it says <New> and type **str** as the name of the expression to watch.
- Hit return to save the change. In **Value** column braces ({...}) should immediately appear as in figure 16.



Name	Value	Address
dataIO	0x000022D8	
str	{...}	0x00002F00
<new>		

Figure 16: watch window with a watch on dataIO and a structure (str)

- Look at the global declaration section in ***volume.c*** .A structure (called str) of type PARMS is declared globally and initialized (the structure type is defined in ***volume.h***)
- Click once on the + sign next to **str,** to expand this line to list all the elements of the structure and their values. (See figure 17, the address values may vary.)



Name	Value	Address	Type
dataIO	0x000022D8		function *
str	{...}	0x00002F00	struct PARMS
Beta	2934	0x00002F00	int
EchoPower	9432	0x00002F04	int
ErrorPower	213	0x00002F08	int
Ratio	9432	0x00002F0C	int
Link	0x00002F00	0x00002F10	struct PARMS *
<new>			

Figure 17: Expanding a watch by clicking the + sign

You can click on the Value of any element in the structure to edit the value for that element.

- In the **Value** column of the Watch window, change the value of a variable. Hit enter. The value in the Watch Window changes colour to red, indicating that you have changed it manually.
- Select the **str** variable in the Watch Window and hit the **Delete** key. Repeat this step for all expressions in the Watch 1 Window.

Using Breakpoints for File I/O

In this section, you add a breakpoint, which is modified to read data from a file on your PC. This is useful for algorithm development. You can use the technique as follows:

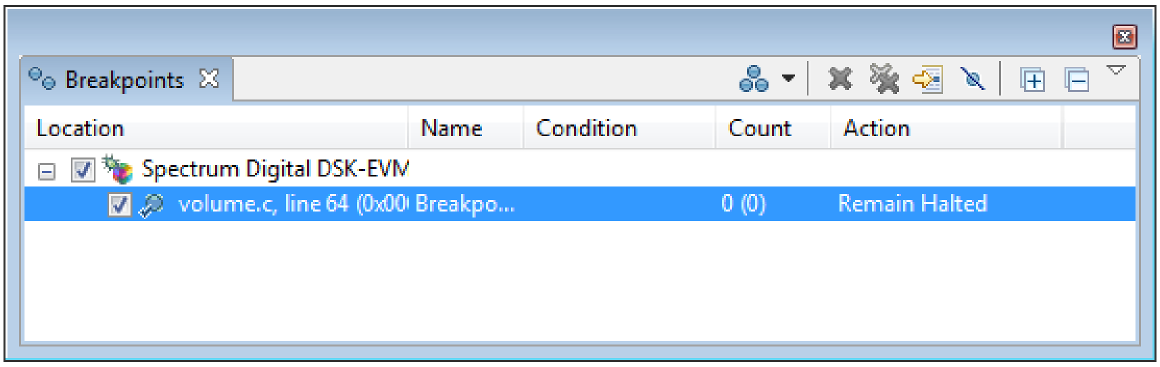
- To transfer input data from a file on the host PC to a buffer on the target for use by the algorithm.
- To transfer output data from a buffer on the target to a file on the host PC.
- To update a window, such as a graph, with data.

This lesson shows you how to use a breakpoint to transfer the contents of a file on the PC to the target for use as test data. It also uses a breakpoint to update all the open windows when it is reached.

- Choose **Target→Reload Program**.
- Maximize the ***volume.c*** code window.
- Put your cursor in the line of the main function that says: dataIO();

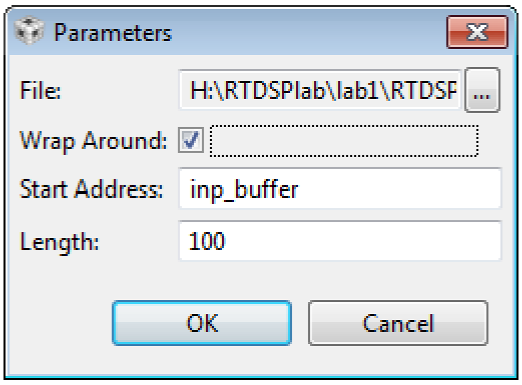
The dataIO function acts as a placeholder, you will add to it later. For now, it is a convenient place to add the breakpoint that injects data from a file.

- Right click the dataIO() line and click **New Breakpoint→Breakpoint** from the pop-up menu.
- Select **View→Breakpoints** from the menu. A window will appear (figure 18). In the **Action** column select **Read Data from File.** A pop-up will appear (similar to figure 19).



Location	Name	Condition	Count	Action
Spectrum Digital DSK-EVM				
volume.c, line 64 (0x00	Breakpo...		0 (0)	Remain Halted

Figure 18: Breakpoints window



File:	H:\RTDSPlab\lab1\RTDSF
Wrap Around:	✓
Start Address:	inp_buffer
Length:	100

Figure 19: Settings for connecting ***sine.dat*** to input buffer

- In the **File** window, click the ... browse button.
- Browse to the **\lab1\RTDSP** folder you created, select **sine.dat** and click **Open**.
- In the Parameters dialog, change the **Start Address** to inp_buffer and the **Length** to 100. Also, put a check mark in the **Wrap Around** box (see figure 19).

- The Address field specifies where the data from the file is to be placed. The inp_buffer is declared in ***volume.c*** as an integer array of BUFSIZE (a constant that is defined in ***volume.h***).
- The Length field specifies how many samples from the data file are read each time the Breakpoint is reached. You use 100 because that is the value set for the BUFSIZE constant in ***volume.h*** (0x64).
- The Wrap Around option causes reading to wrap to the beginning of the file when it reaches the end of the file. This allows the data file to be treated as a continuous stream of data even though it contains only 1000 values and 100 values are read each time the Breakpoint is reached.

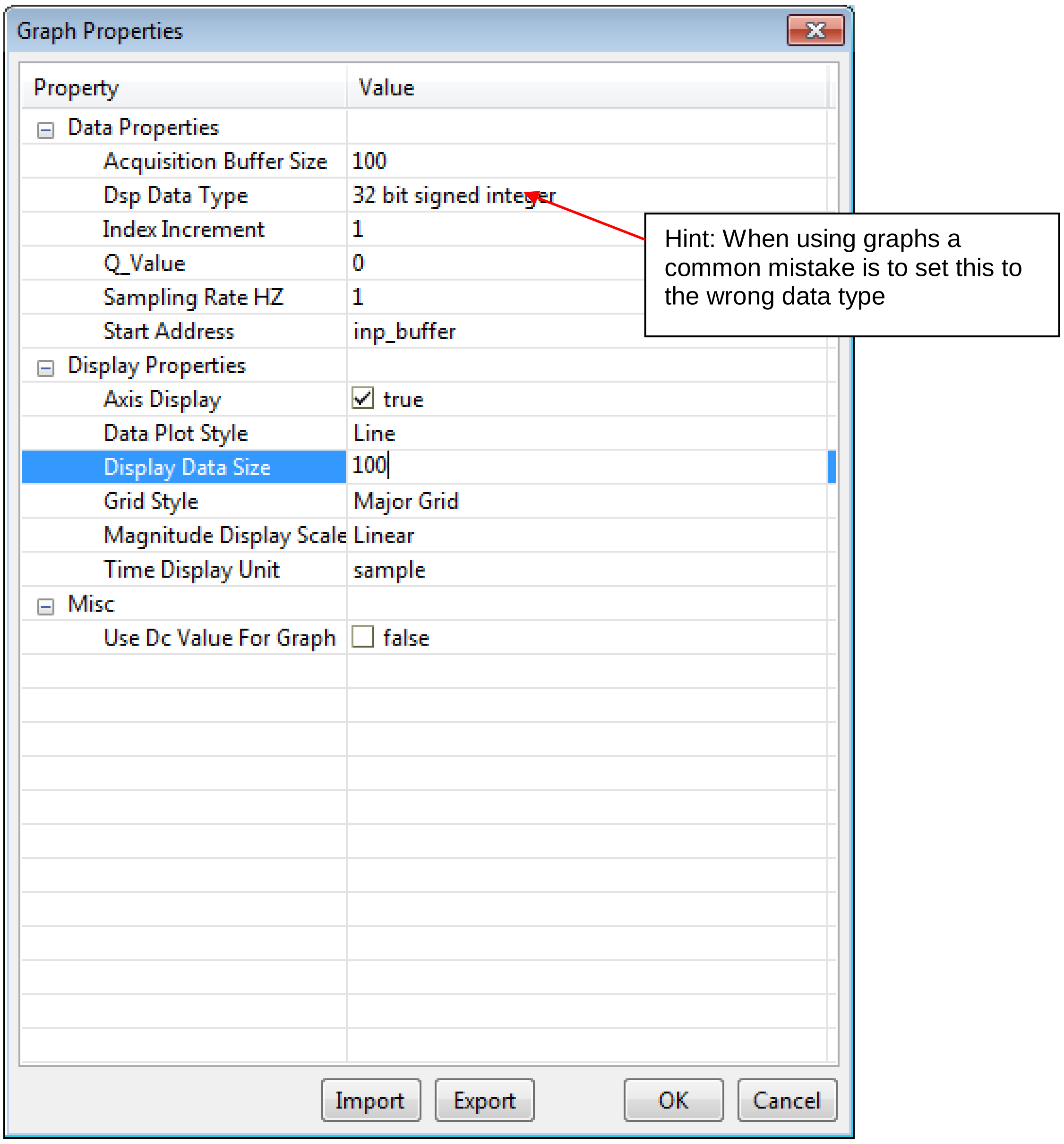
- Hit **OK** to close the parameters dialogue.

Displaying Graphs

If you ran the program now, you would not see much information about what the program was doing. You could set watch variables on addresses within the inp_buffer and out_buffer arrays, but you would need to watch a lot of variables and the display would be numeric rather than visual.

There are a variety of ways to graph data processed by your program. In this example, you view a signal plotted against time. You open and configure the graphs in this section and run the program in the next section.

- Choose **Tools→Graph→Single Time**.
- In the Graph Property Dialog, change the **Start Address**, **Acquisition Buffer Size, Display Data Size, DSP Data Type** and **Display Data Size** properties to the values shown in figure 20.

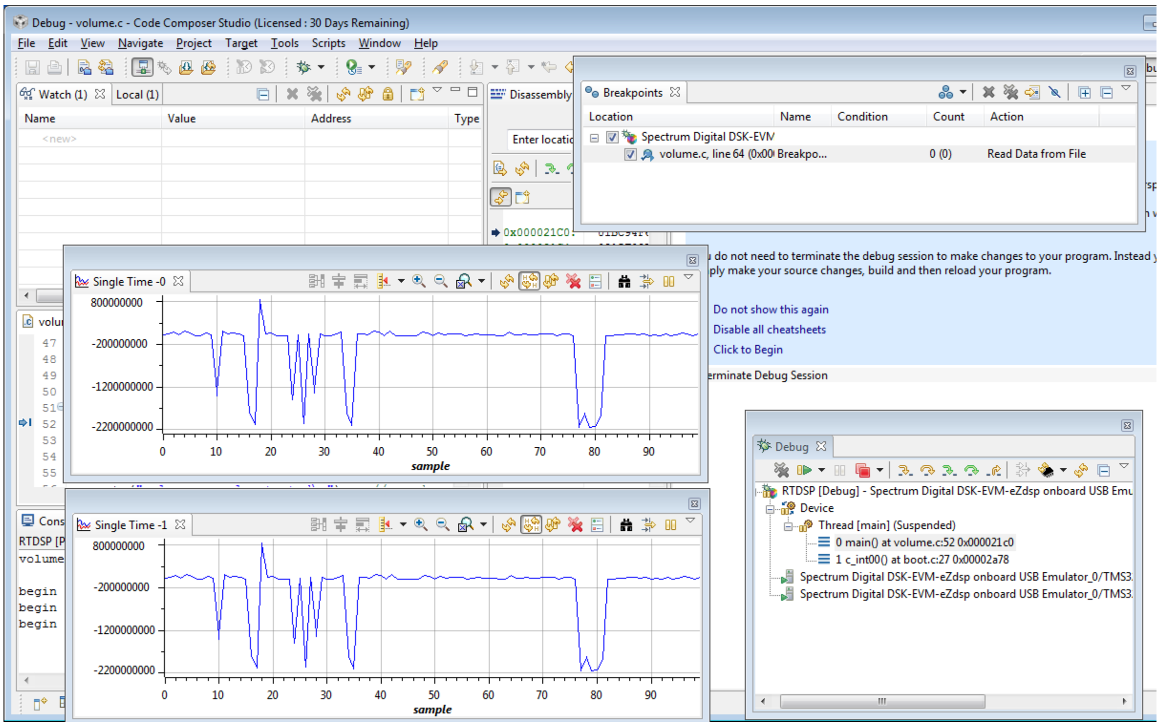

Graph Properties

Property	Value
⊟ Data Properties	
Acquisition Buffer Size	100
Dsp Data Type	32 bit signed integer
Index Increment	1
Q_Value	0
Sampling Rate HZ	1
Start Address	inp_buffer
⊟ Display Properties	
Axis Display	✓ true
Data Plot Style	Line
Display Data Size	100
Grid Style	Major Grid
Magnitude Display Scale	Linear
Time Display Unit	sample
⊟ Misc	
Use Dc Value For Graph	☐ false

Hint: When using graphs a common mistake is to set this to the wrong data type

Figure 20: Graph properties dialog

- Click **OK**. A graph window for the Input Buffer appears (labelled **Single time -0)**.
- Right-click on the tab of the Input graph window and choose **Detached** from the pop-up menu.
- Choose **Tools→Graph→Single Time** again.
- This time, change the **Start Address** to **out_buffer.** All the other settings are the same as you set for the input buffer.
- Click **OK** to display the Output graph window. Right-click on the tab of the Output graph window and choose **Detached** from the pop-up menu. Place the input window (titled **Single Time-0**) above the Output window (titled **Single Time -1**). Your windows should look something like figure 21:


Debug - volume.c - Code Composer Studio (Licensed : 30 Days Remaining)
File Edit View Navigate Project Target Tools Scripts Window Help
Watch (1) Local (1)
Name Value Address Type
Disassembly
Breakpoints
Single Time -0
Single Time -1
sample
Debug

Location	Name	Condition	Count	Action
Spectrum Digital DSK-EVM				
volume.c, line 64 (0x00	Breakpo...		0 (0)	Read Data from File

u do not need to terminate the debug session to make changes to your program. Instead y
ply make your source changes, build and then reload your program.

Do not show this again
Disable all cheatsheets
Click to Begin

erminate Debug Session

```
RTDSP [Debug] - Spectrum Digital DSK-EVM-eZdsp onboard USB Emu
  Device
    Thread [main] (Suspended)
      0 main() at volume.c:52 0x000021c0
      1 c_int00() at boot.c:27 0x00002a78
  Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator_0/TMS3
  Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator_0/TMS3
```

Figure 21: Displaying Graphs

Updating the Graphs while the program is executing

So far, you have placed a Breakpoint, which temporarily halts the target, transfers data from the host PC to the target, and resumes execution of the target application. However, the Breakpoint does not cause the graphs to be updated. In this section, you create 2 more breakpoints that cause the graphs to be updated.

1. In the ***volume.c*** window, put your cursor in the line that calls dataIO.
2. Right click this line and select **New Breakpoint→Breakpoint**
3. In the breakpoint window select **Update View** as the action for this new breakpoint, in the pop-up that appears (figure 22) select **Single Time -0** Hit **Ok**

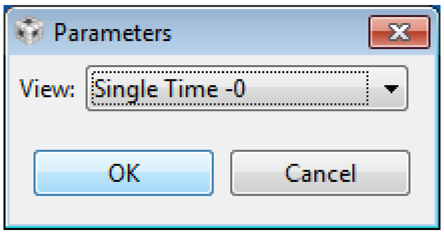
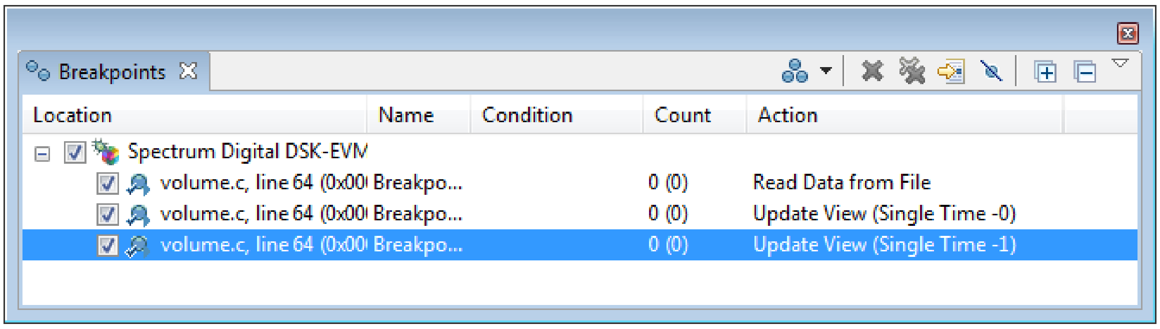


Figure 22: Breakpoint set to update a graph

4. Repeat steps 1 to 3 but this time select **Single Time -1** as the view to update. The Breakpoint window should look like the figure 23:



Location	Name	Condition	Count	Action
Spectrum Digital DSK-EVM				
volume.c, line 64 (0x00	Breakpo...		0 (0)	Read Data from File
volume.c, line 64 (0x00	Breakpo...		0 (0)	Update View (Single Time -0)
volume.c, line 64 (0x00	Breakpo...		0 (0)	Update View (Single Time -1)

Figure 23: breakpoint window with 3 operations to do on 1 breakpoint

You put breakpoints on the same line as the initial breakpoint forcing the target to halt only once to perform all operations—transferring the data and updating the graphs.

5. Arrange the windows so that you can see both graphs.
6. Click Target->Run[3]. You should see graphs similar to those shown in figure 24.
7. Each time the breakpoint is reached, the IDE gets 100 values from the ***sine.dat*** file and writes them to the inp_buffer address.
8. Choose **Target→Halt** to quit running the program.

[3] **Note:** Code Composer Studio briefly halts the target whenever it reaches a breakpoint. Therefore, the target application may not meet real-time deadlines. At this stage of development, you are testing the algorithm. For analyzing real-time behaviour you need to use RTDX and DSP/BIOS. This is not covered in this course.

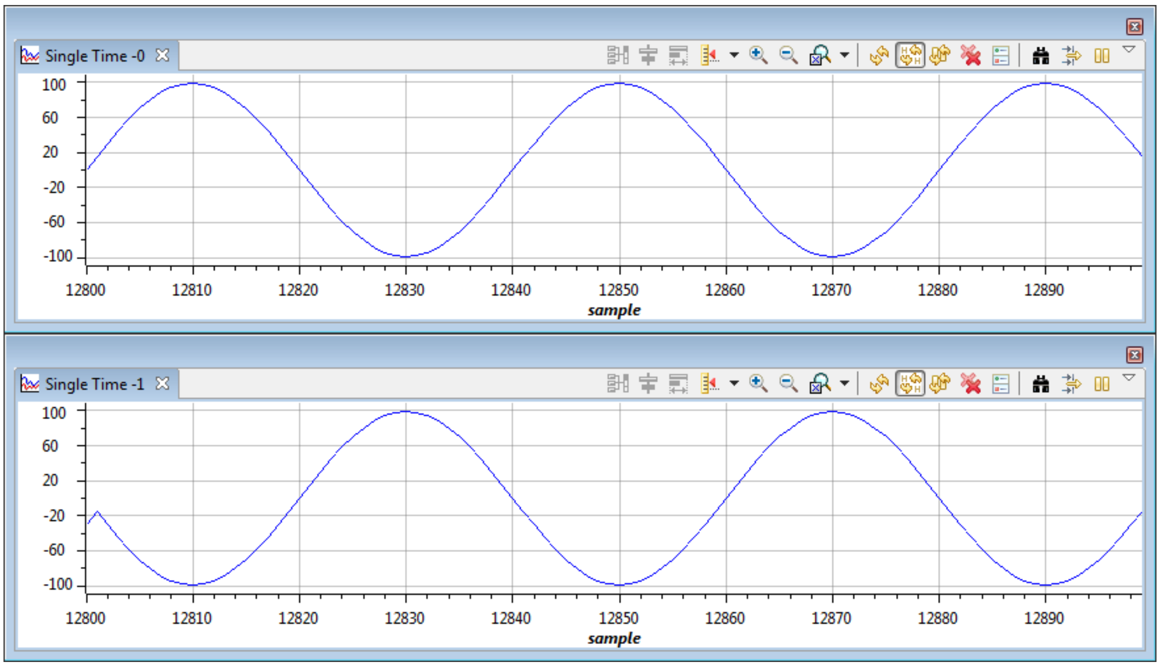


Figure 24: Input and Output graphs connected and updated via breakpoints.

Adjusting the Gain

Recall that the processing function multiplies each value in the input buffer by the gain and puts the resulting values into the output buffer. It does this by performing the following statement within a while loop: *output++ = *input++ * gain;

This statement multiplies a value in inp_buffer by the gain and places it in the corresponding location in the out_buffer. The gain is initially set to MINGAIN, which is defined as 1 in ***volume.h***. To modify the output, you need to change the gain. One method is to use a watch variable.

- Go to the watch window and select the **Watch1** tab.
- Click on <New> in the Name column and type gain as the name of the variable to watch.
- Hit return to save the change. The **Value** (1) should immediately appear.
- If you have halted the program, **Target-→Run** to restart the program. Observe the input and output graph you created earlier.
- Choose **Target→Halt**.
- In the Watch Window, select the **Value** of gain and change it to 10. Hit return.
- Choose **Target→Run**. Notice that the amplitude of the signal in the output graph changes to reflect the increased gain (figure 25). Note you can also change the watch value whilst the program is running to see the graphs updated immediately.

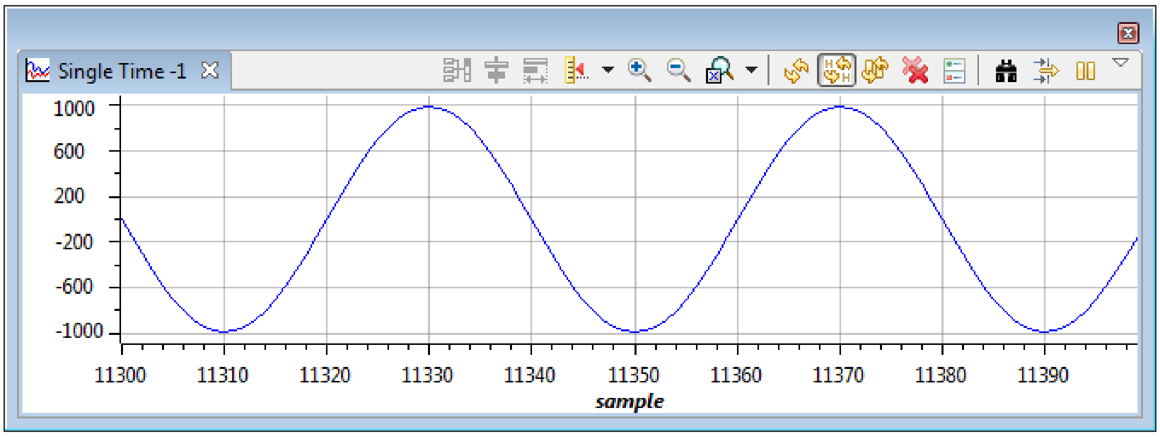


Figure 25: Effect of increasing the gain

- Choose **Target** →**Halt**.
- Remove all the breakpoints and close all graph windows.

Measuring Instruction cycles

Execution time becomes an important issue with DSP algorithms as the software usually has to meet real-time deadlines. The tools for measuring various aspects of code efficiency are called **profiling** tools and one particular profiling tool included in CCS can measure the number of execution cycles for sections of code. You will need to use this tool for subsequent laboratory sessions to measure the instruction cycles of the MAC (**M**ultiply **AC**cumulate) algorithms for the digital filters we will ask you to develop.

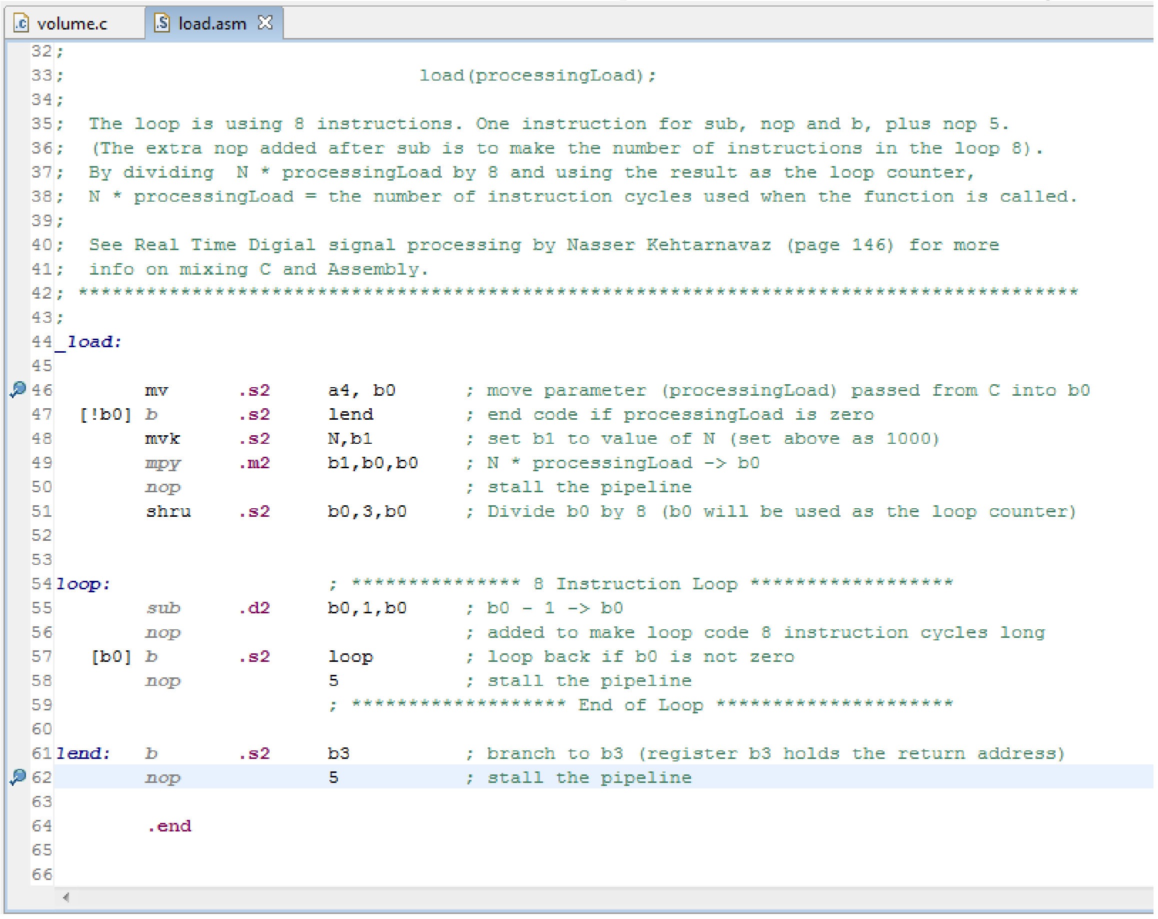
Using the profiling clock to measure instruction cycles

The simplest tool CCS provides for measuring instruction cycles is the profiling clock. This clock measures instruction cycles between two breakpoints.[4]

- Select **Target**→**Clock**→**Enable**. A clock icon 🕐 :0 appears at the bottom of CCS
- Go to the **Watch(1)** tab. Create a watch on the variable processingLoad

[4] This method does have some accuracy issues, errors due to managing the breakpoint. These errors are usually less than a few instruction cycles so for the purpose of this lab ignore such errors.

- Using the **C/C++ Projects** window open the ***load.asm*** file into a code window. Maximise the window then insert two breakpoints as shown in figure 26:

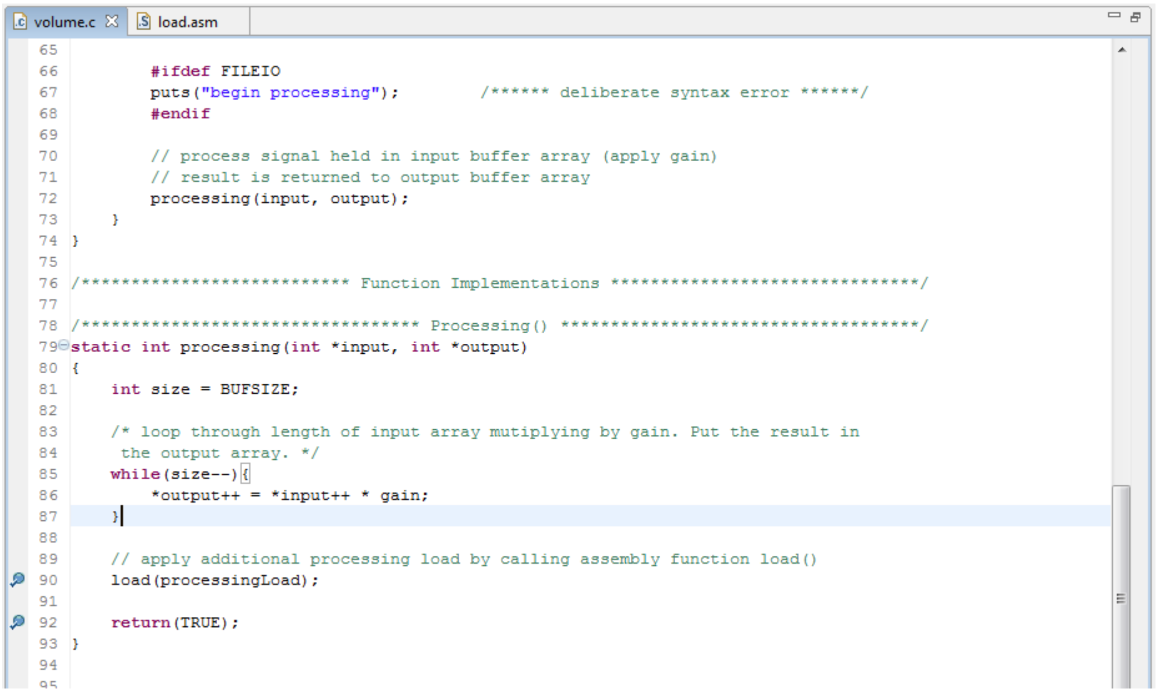


```
;
;                               load(processingLoad);
;
;  The loop is using 8 instructions. One instruction for sub, nop and b, plus nop 5.
;  (The extra nop added after sub is to make the number of instructions in the loop 8).
;  By dividing  N * processingLoad by 8 and using the result as the loop counter,
;  N * processingLoad = the number of instruction cycles used when the function is called.
;
;  See Real Time Digial signal processing by Nasser Kehtarnavaz (page 146) for more
;  info on mixing C and Assembly.
; *************************************************************************************
;
_load:

        mv      .s2     a4, b0      ; move parameter (processingLoad) passed from C into b0
  [!b0] b       .s2     lend        ; end code if processingLoad is zero
        mvk     .s2     N,b1        ; set b1 to value of N (set above as 1000)
        mpy     .m2     b1,b0,b0    ; N * processingLoad -> b0
        nop                         ; stall the pipeline
        shru    .s2     b0,3,b0     ; Divide b0 by 8 (b0 will be used as the loop counter)


loop:                   ; *************** 8 Instruction Loop ******************
        sub     .d2     b0,1,b0     ; b0 - 1 -> b0
        nop                         ; added to make loop code 8 instruction cycles long
   [b0] b       .s2     loop        ; loop back if b0 is not zero
        nop             5           ; stall the pipeline
                        ; ******************* End of Loop *********************

lend:   b       .s2     b3          ; branch to b3 (register b3 holds the return address)
        nop             5           ; stall the pipeline

        .end


```

Figure 26: Breakpoints set in ASM code

- Click **Target->Run**, CSS will run the code to the first breakpoint at the line that states mv a4, b0 and then halt (you will see a blue arrow denoting the current position of the cursor is at this breakpoint).
- Double click on the number to the right of the clock :0 to reset it to zero.
- Run the code to the second breakpoint. Take a note of the number of instruction cycles reported next to the clock icon it should be 1014 cycles. (This count does not include the NOP 5 instruction)
- Increase the value of processingLoad in the watch window to 2 (remember to hit return after entering or the value will not be set) then run to the first breakpoint again (this may take a few hits of **Run**). Reset the clock again by clicking on the count number next to it. Run to the second breakpoint and the cycles measured should increase to 2014.
- Try processingLoad set to values of 5, 10 and 12.
- By using the comments in the code as help, ensure you understand why changing the variable processingLoad varies the instruction cycles proportional to: 1000 X processingLoad.

- Instruction cycles can also be measured between C code statements. Remove the breakpoints set in ***load.asm*** (you can double click on their icons in the margin) then maximize **volume.c** Set two breakpoints in the C code as of figure 27:



```
        #ifdef FILEIO
        puts("begin processing");        /****** deliberate syntax error ******/
        #endif

        // process signal held in input buffer array (apply gain)
        // result is returned to output buffer array
        processing(input, output);
    }
}

/*************************** Function Implementations ******************************/

/********************************** Processing() ************************************/
static int processing(int *input, int *output)
{
    int size = BUFSIZE;

    /* loop through length of input array mutiplying by gain. Put the result in
     the output array. */
    while(size--){
        *output++ = *input++ * gain;
    }

    // apply additional processing load by calling assembly function load()
    load(processingLoad);

    return(TRUE);
}
```

Figure 27: Breakpoints set in C code

- Reset processingLoad to 1 and then measure the instruction cycles between the breakpoints. Notice that we are still measuring the cycles consumed by ***load.asm*** but the result (1025) is greater due to managing the call to assembly from the C code.
- Close CCS

Questions

8. **After making changes to C code and/or Assembly what should be done before the code can execute (and where does the code finally execute)?**

9. **A Programmer writing C code notices that the algorithm they are implementing requires a constant, which will not change at runtime. Rather than type the value many times in the C code he/she decides to keep the code tidy by using a pre-processor directive and thus writes the following line of code:**

   ```
   #define GRAVITY = 10;
   ```

   What are the two syntax errors in this statement? Does the (corrected) statement produce any machine code? Why?

10. **The assembly program *load.asm* uses a set directive to define a constant N which is then set to 1000. (The set directive in assembly parallels the #define directive used in C). By setting N to 125 (i.e. 1000 by 8), suggest a modification to *load.asm* so that the program uses fewer instruction cycles in the code section used to set up the loop. How many instructions does this save?**

Deliverable

You should be able to answers the questions in **bold** type (10 questions in total). A mixture of some of these questions and others about this lab and section 1 of the lecture notes will be tested in online quiz on Blackboard. This test is not for marks, but will provide good feedback for your general understanding of some basics you need for the rest of the course.

Revision History

29th Jan 2007
11th Dec 2007
27th Jan 2009
11th Jan 2010 – added mark allocation
23rd Sept 2010 – Updated document for CCsV4 and Windows 7 (DH)
18th Dec 2012 – Added info about individual/group work
20th Jan 2014 – Added info about online quiz