

PORTLAND STATE UNIVERSITY

ECE 371/372 – Microprocessors

Prof. Douglas Hall

Introduction to BeagleBone Black and TI Code Composer Studio

Modified by: Doug Hall Fall 2018

Original Creation: Leela Kamalesh Yadlapalli

Contributions:

Saroj Bardewa, Nikhil Marda, Sanket Borhade, Sameer Ghewari,
Alex Olson, Prof. Phillip Wong, Tyler Hull

Table of Contents

List of Figures	4
Preface	5
Part – 1 – BeagleBone Black	6
Introduction	6
High Level Specifications	6
Setup and Handling your Personal BeagleBone Black	7
Connecting the Emulator to the Cable	8
Connecting the BeagleBone Black to the Cable	9
Handling the BeagleBone	11
Booting the BeagleBone Black	11
LED Indicators	12
Booting Info	12
Power ON Sequence for PERSONAL BeagleBone Systems	13
How to exit Linux	13
Power OFF Sequence	14
Troubleshooting	14
Part – 2 – Using TI Code Composer Studio to Create, run, and Debug Programs	15
Introduction	15
Project Creation	16
Writing Assembly files	20
Assembling, List File Generation, and Linking	22
Troubleshooting Hints	24
Configuring Emulator to be Ready to Load and Run Program	24
Launching the Configuration and Loading Your Program	26
Debugging Your Program	27
Debugging Options	27
Using single stepping to run and debug a program	28
Check on values in Registers, Memory and Disassembly	29

Breakpoints	30
Setting and Removing Hardware Breakpoints	31
References	33

List of Figures

Figure 1 : Rev. C BeagleBone Black	6
Figure 2 : BeagleBone Black overview	7
Figure 3 : Emulator flat cable orientation	8
Figure 4 : “Keyed” or “Not Populated” pins on the BBB and JTAG cable	9
Figure 5 : JTAG cable seated properly onto the BBB JTAG header	10
Figure 6 : JTAG emulator connected to BBB with flat cable and to emulator with USB cable	10
Figure 7 : Handling the BBB by touching its plastic headers	11
Figure 8 : BBB Lab setup (with cape)	11
Figure 8 : BBB with the cape removed –white plastic card pieces can be seen	12
Figure 9 : LEDs on BBB	13
Figure 10 : Main Start window	15
Figure 11 : Remember to choose N: Drive	16
Figure 12 : Creating New CCS Project	16
Figure 13 : Select CCS Project from the list	17
Figure 14 : Selecting Target for CCS Project	17
Figure 15 : Different fields in New CCS Project Creation	18
Figure 16 : Project Template	19
Figure 17 : Project Window	19
Figure 18 : Creating a new Assembly file (.s)	20
Figure 19 : GNU Linker Option	22
Figure 20 : Setting a list file	23
Figure 21 : Launch target configuration	25
Figure 22 : Linking a User defined target configuration to a project	25
Figure 23 : Debugging a program	26
Figure 24 : Load .out file	27
Figure 25 : Beginning of Debug process	27
Figure 26 : Debug options	28
Figure 27 : Register Entries	29
Figure 28 : Memory Content for a given memory address	30
Figure 29 : Disassembly shows program execution	30
Figure 30 : Set hardware breakpoints	31
Figure 31 : Setting Breakpoint at _start	32
Figure 32 : Debug window layout	32
Figure 33 : Auto program reload option	33

Preface

In this document, we shall introduce you to ARM assembly language programming on BeagleBone Black (BBB) Single Board Computer (SBC) with Texas Instruments' Code Composer Studio (CCS) Integrated Development Environment. This guide applies to CCS for the Windows, Linux, and MacOS environments. Instructions for installation of TI CCS can be found in another document. Links for further reference can be found in the appendix.

Hope you have great fun with the BeagleBone Black and ECE371/ECE 372!

Part – 1 – BeagleBone Black

Introduction

BeagleBone Black is a low-power, open-source hardware, single-board computer designed by Beagleboard.org (non-profit) foundation. It is produced by CircuitCo PCB Solutions out of Richardson, TX. This board is supported by a large open-source community and Texas Instruments. It was released on April 13, 2013 at \$45 (now ~\$60) and competes with Raspberry Pi and other ARM based SBCs. The board has undergone a number of production revisions. The boards we are using are Rev. C.



Figure 1 : Rev. C BeagleBone Black

BeagleBone Black or BBB is one of the better boards available on the market to learn about ARM microprocessors, microprocessor interfacing and embedded systems development with Linux. The BBB setup makes it possible to do “Bare Bones” assembly language programming and directly interact with the hardware without going through the Linux operating system. This capability is one of the main reasons the BBB was selected for this course. It has a lot of general purpose IO pins that can be used to interface with many peripherals. Being open-source, you can easily find all the documentation including the PCB files. Texas Instruments has also provided a lot of support with their datasheets, tutorials and software. Raspberry Pi does not provide the same kind of low-level access as BBB and doesn't publish complete data regarding its components. Therefore, it is not suitable for low level assembly language programming.

High Level Specifications

Without going into too much detail, BeagleBone Black board has following features:

- ARM Processor, Cortex – A8, TI AM3358 (v7 Architecture)
- 512 MB DDR3 RAM and 4 KB EEPROM
- 4 GB eMMC (Non Volatile Flash + Flash controller)
- Memory card, Ethernet Support
- HDMI Video output and USB port
- 3 Buttons – Reset, Boot, Power
- Many communication protocols, GPIO Ports, etc.

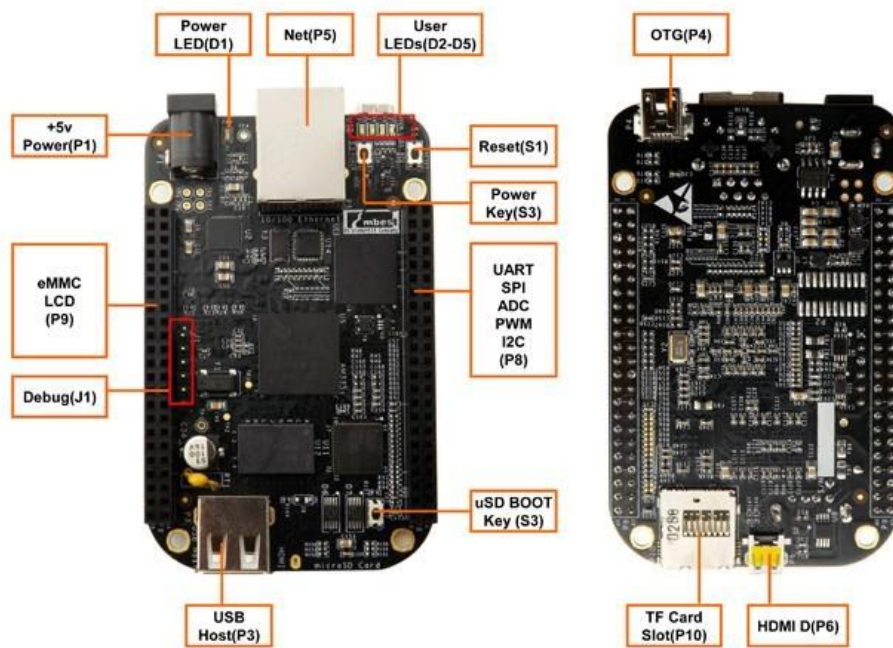


Figure 2 : BeagleBone Black overview

BeagleBone Black is a Single Board Computer (SBC). You can connect a keyboard and mouse to the USB port, a monitor to the HDMI port, plug the Ethernet cable in, and power it on. It will boot a pre-loaded version of Linux. But for this class, we are not going to use it as a SBC. We will be using it as a development board to explore how you communicate directly with the peripheral hardware devices.

Setup and Handling your Personal BeagleBone Black

If you purchased a BeagleBone Black board for use in this class, you'll need to get your Beaglebone Black board connected to your emulator, and your emulator connected to your computer before you can get started.

Connecting the Emulator to the Cable

The Blackhawk XDS100V2-D emulator uses a 1.27mm pitch 20 pin connector. This connector is much smaller than the standard 2.54mm pin spacing and you should take care while making the connections so that you don't damage the emulator, the BBB, or your cable. In addition, use a gentle touch when dealing with the cable as it uses a clamping method to push the wires onto small metal blades to make the connection. If the plastic connectors are strained or break, then the clamping force will not be able to maintain a good connection on the wires.

As you can see in figure 3 below, your ribbon cable will have 20 wires with one of them being a colored wire. Most cables have a red color, but as you can see, there may be other colors used as well.

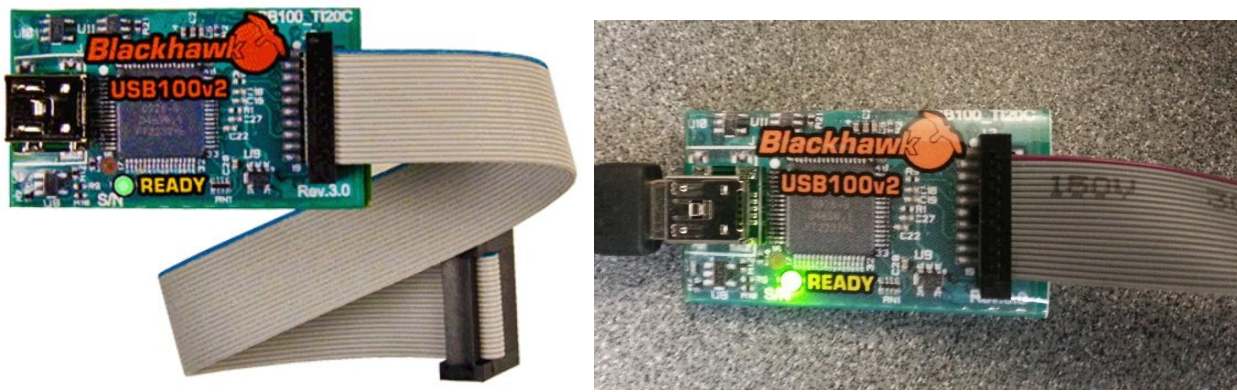


Figure 3 : Emulator flat cable orientation.

First: Orient your emulator with the USB port to the left. The text should be rightside up for you in this orientation.

Second: Orient your cable so that the colored wire is at the top as shown in the images above.

Third: Line up the holes in the connector with the pins on the emulator. Then press the connector down onto the emulator until the emulator pins are making a solid connection with the cable connector.

Note: When removing this cable, make sure you gently wiggle the connector while pulling it away from the emulator. It is easy to bend the emulator pins if you do not pull off the connector straight away from the emulator while removing the connector.

When you return it, it would be best to leave the flat cable connected to the emulator, and just remove the end of the cable that is connected to your BeagleBone Black.

Connecting the BeagleBone Black to the Cable

The JTAG header soldered on your BeagleBone Black and the connector on the flat cable are both “keyed”. On the right hand side of the image below, you will see that the BBB side of the JTAG cable has a slightly larger connector with some plastic broken off inside one of the holes. This makes it so that the cable will only fit on in one orientation, and it will not fit onto the emulator. Since the emulator has one pin that is not populated, you must match up this empty pin with the filled hole on the connector to plug in the cable.

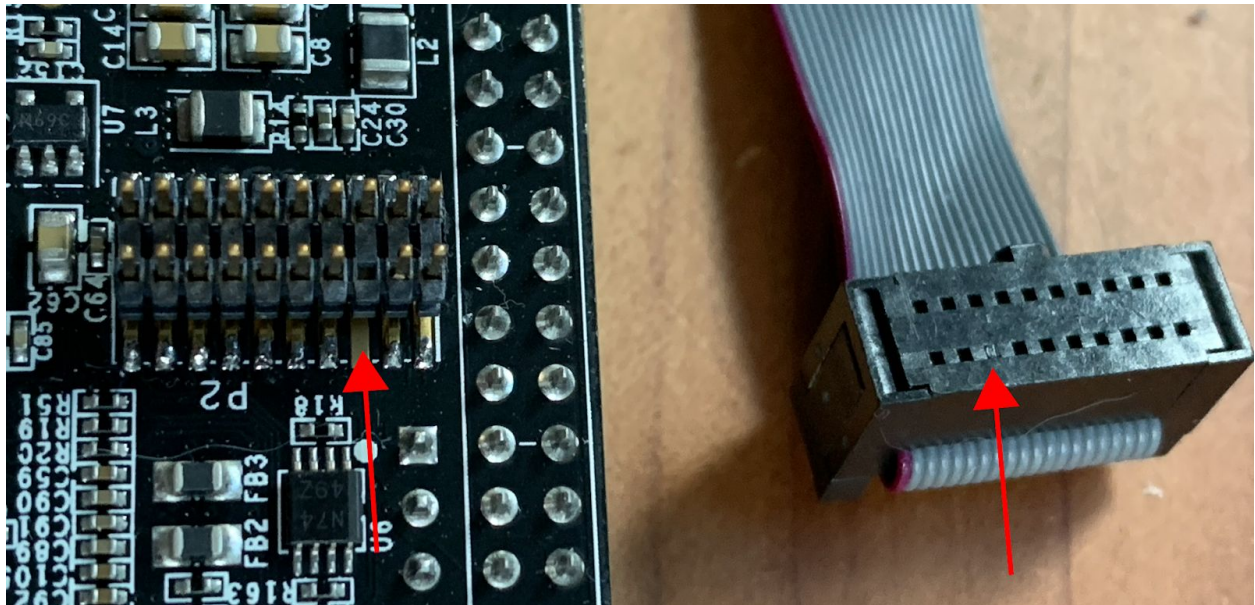


Figure 4 : “Keyed” or “Not Populated” pins on the BBB and JTAG cable.

First: Find the keyed (filled) hole and missing pin on your cable and BBB.

Second: Orient your cable so that the keyed pin and connector hole are lined up.

Third: Press the connector down onto the emulator until the emulator pins are making a solid connection with the cable connector.

Fourth: Connect the emulator to your computer using the provided USB cable.

Note: When removing this cable, make sure you gently wiggle the connector while pulling it away from the BBB. It is easy to bend the JTAG pins if you do not pull off the connector straight away from the BBB while removing the connector.

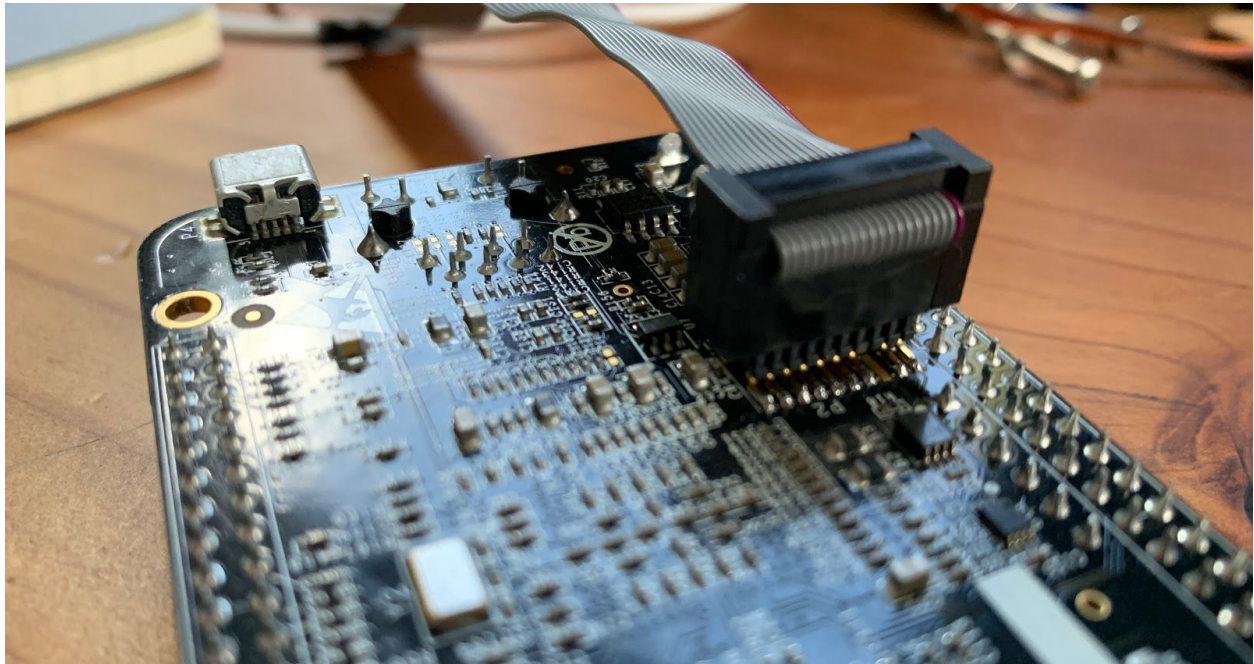


Figure 5 : JTAG cable seated properly onto the BBB JTAG header.



Figure 6 : JTAG Emulator connected to the BBB with the flat JTAG cable and to the computer with the provided USB cable.

Handling the BeagleBone

Warning: Try not to directly touch the board. As with any electronics device, static electricity from your body may burn out the chips on the board.



Figure 7 : Handling the BBB by touching its plastic headers.

You should do your best to not directly touch the board to avoid possible ESD issues. You can avoid this by touching the headers only, wearing an anti-static wrist strap, or finding a case, box, or standoffs for your BBB.

Booting the BeagleBone Black

The following sections will show you how to boot the BeagleBone Black into Serial mode.

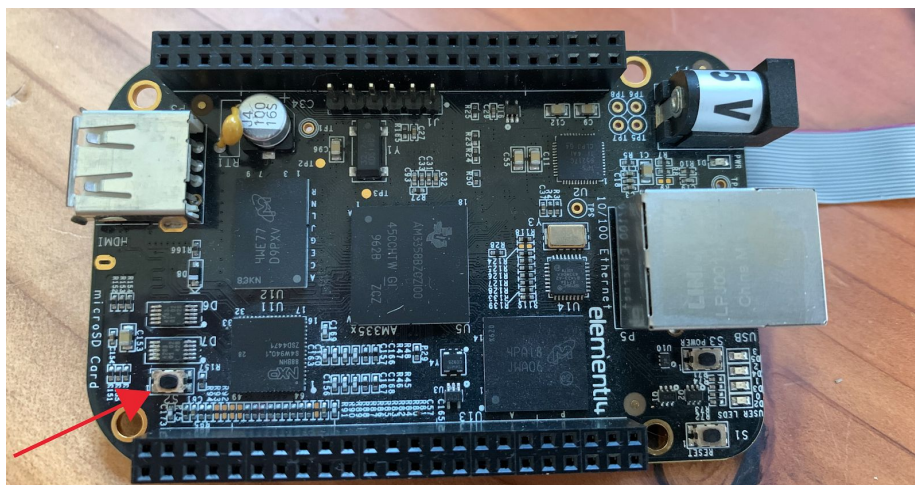


Figure 8 : Boot button location on the BBB. Button must be held down while powering on.

LED Indicators

Before explaining the booting procedure of BeagleBone Black, let us examine the status LEDs on the board. There are five LEDs on BBB as shown in Figure 5. The top left corner has a Power LED which stays on whenever the board is in operation. Likewise, the top right side has four user-configurable LEDs (USER3, USER2, USER1 and USER0, from the left to the right). These LEDs can be controlled through user programs as we shall see in later exercises. Now, let us work through the procedure to boot the board into the mode we need for this course.

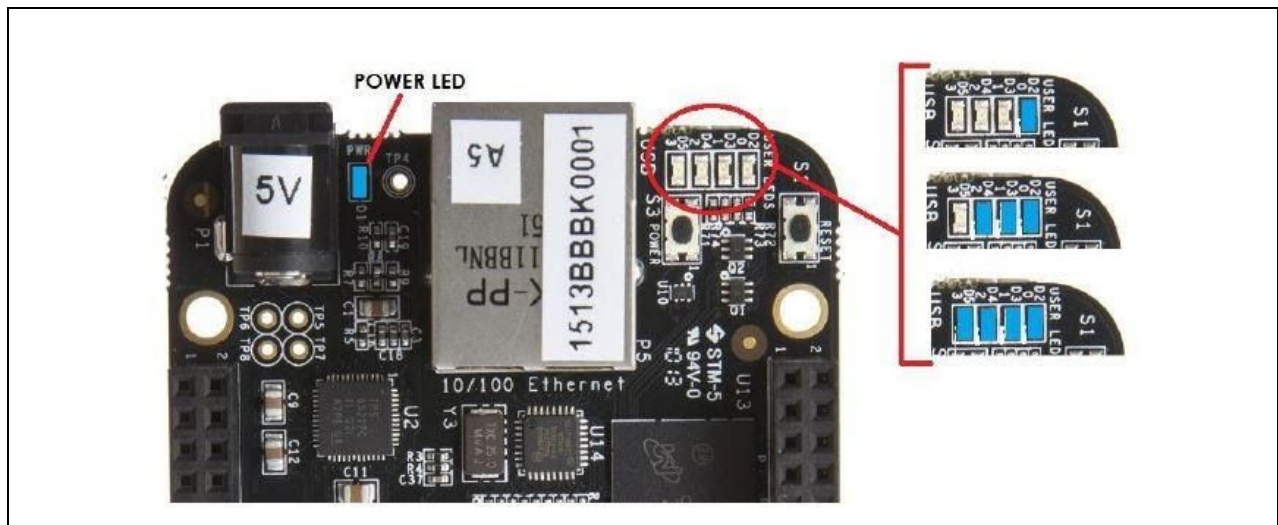


Figure 9 : LEDs on BBB

Booting Info

BeagleBone Black can be booted in a number of ways, the same way your personal computer has options to boot from a hard drive, network connection or thumb drive. There is a hard-coded ROM chip in **arm335x**, and the code stored inside it performs platform configuration and initialization as part of the public start-up procedure. BBB can be made to boot Linux from its onboard 4 GB Flash (eMMC), from an SD card in the SD slot, from a USB thumb drive connected to the onboard USB, or via serial port pins on the board.

The “boot button” on the board enables us to select one of these methods. The boot behavior can be changed by pressing and holding the boot button before the board is powered on with the power rocker switch. The onboard “boot button” alters the boot order as follows:

Without Boot switch press, before powerup:	eMMC (Linux)
With Boot switch press, before powerup:	Serial (Blank)

By default, if you power on the board without pressing the boot button on the board, it starts loading Linux from the 4 GB Flash chip. This will be clearly evident by observing the four USER LEDs.

When running Linux, the four USER LEDs have the following functionalities:

- USER0: Linux boot indicator –blinks in a heartbeat pattern
- USER1: MicroSD card access indicator
- USER2: Activity Indicator – turns on when the board is not idle
- USER3: eMMC access indicator

As you can notice, flashing of the USER0 LED indicates that BBB has finished booting into Linux.

For ECE 371 and ECE 372, we DO NOT want to boot into Linux. Follow the procedure below to boot into Serial mode:

Power ON Sequence for PERSONAL BeagleBone Systems

- Verify that the USB cable is plugged into the BBB this should be the mini-usb side, but **NOT** into a power source (Your computer or a 5V wall adapter). This would be the USB-A side of the cable.
- Press and hold the Boot button. When you press the button down, you should hear or feel a click.
- While still holding down the boot button, power on the board by plugging in the USB-A side of the cable and supplying power to the BBB.
- After waiting for a few seconds, release the boot button.
- If the sequence has been followed correctly, only the Power LED on the board should light up and the 4 USER LEDs on the right should stay off.
- At this point, the board is ready for our purpose. You can start using Code Composer Studio.

Do not panic if you miss the above sequence and the board starts booting Linux. The booting of Linux will be evident when the user LEDs start glowing in addition to the power LED. Follow the section 'how to exit Linux' ahead.

How to exit Linux

If you accidentally booted into Linux, (perhaps because you forgot to hold down the Boot button when turning on the power), then the Power LED will glow and several USER LEDs will be blinking.

- Wait for at least a minute. This gives enough time for Linux to boot completely. As stated in the boot info section, we can find that the Linux has booted completely when only the first LED stays blinking in the Heartbeat fashion and the remaining three stays off. **Turning off the power prematurely could corrupt the Linux file system!**
- Now, once Linux is booted properly, press the “power button” on the board (the switch closer to USER LEDs on the board as shown in Figure 5), **NOT by disconnecting power to the board**. Also remember that both power and reset buttons are covered by one plastic piece. Press down and then release the Power button (to the left of the Plastic piece). You should hear or feel a "click" as the button makes contact.

- After several seconds, the USER LEDS and the Power LED will all be off. The BeagleBone Black is now in an off state, though power is still present! Switch off the power supply, or unplug the BBB from the power supply.

Caution: Do not press the Reset button while the board is booting Linux or has booted Linux! This just resets the processor on the board and it again starts to boot Linux. Also remember that you need to turn off the power to the board when you have exited Linux before trying to boot the correct way.

Power OFF Sequence

- After you are done with programming, just flip the power rocker switch from the power supply to the OFF position.
- Make sure that the power LED on the board is switched off. This ensures that the board has now shut down.

Note to students with their own BeagleBone Black boards: BeagleBone Black doesn't come with the 20 Pin JTAG connector by default. It just comes with the pin outs at the back of the board. For the purpose of this Lab, a JTAG header has been soldered carefully onto the board to which the emulator is attached via the Flat ribbon cable. Further info on how to solder with pictures can be found at:

http://www.tincantools.com/wiki/Flyswatter2_BeagleBone_Black_How_To

However, if you need to do this on your own board, I suggest you get help from the PSU ECE Prototyping Lab.

Troubleshooting

As always, follow regular procedures in booting and maintain caution while handling the board. When you have booted into Linux, power it down when it has completely booted and remember to power it down before trying to start up again with the boot button. Pressing the reset button while Linux is loading does nothing but restart loading Linux. Reset doesn't have any effect once you boot into Lab mode. If you run into trouble despite this guide, please contact the TA or professor. Do not attempt to move boards around or change wiring!

Part – 2 – Using TI Code Composer Studio to Create, run, and Debug Programs

Introduction

Code Composer Studio (CCS) is an Integrated Development Environment for developing embedded applications with TI Embedded processors. TI produces a variety of processors, and BBB has an AM3358 Cortex A8, a processor in TI's Sitara family of ARM Application processors. We will be using the V8 of CCS for ECE 371 and ECE 372. CCS offers both the proprietary TI developed tool chain (Compiler, Assembler, Linker etc.) as well as the GNU open source Tool chain. We have the GNU gcc tool chain installed in the Lab. The free version of TI CCS can be only used with the GNU tool chain. We are using TI CCS in our lab because of the support TI has provided for the BeagleBone Black and the XDS100V2 emulator.

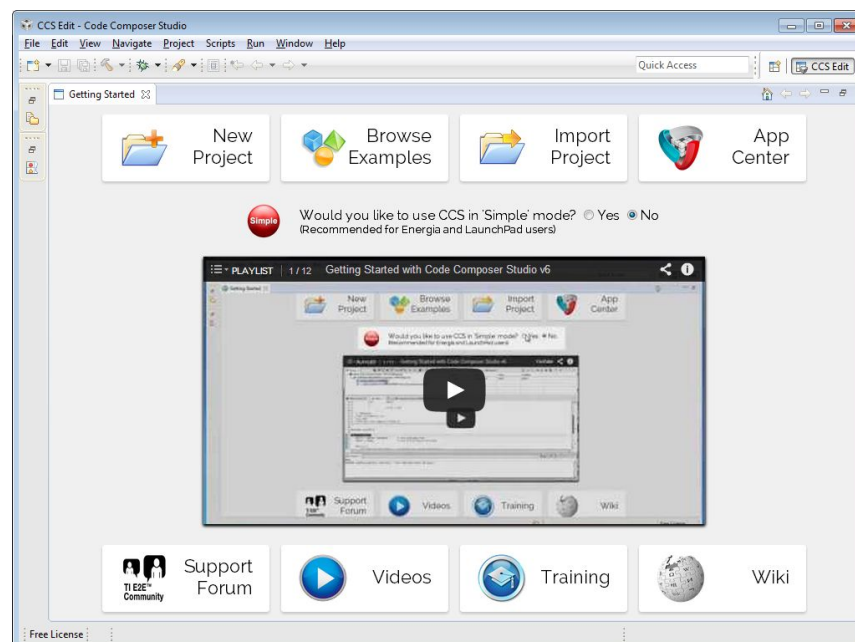


Figure 10 : Main Start window

To start CCS, click on the CCS icon or select it from the program list. In the Windows Program list, a folder may have been created. Click on Code Composer Studio 8.X.X folder, and in that folder click on Code Composer Studio 8.X.X file with the cube next to it (where the X's stand for subversion numbers). When you open the CCS for the first time, it will ask you a directory for (saving and retrieving) your workspace. **ALWAYS USE MAPPED DRIVES. NEVER USE UNC PATHS.** The path should be: **C:\(Workspace Directory)**, as shown below. Tick the **“Use this as default and do not ask again”** box, so you do not have to enter the workspace every time you start CCS.

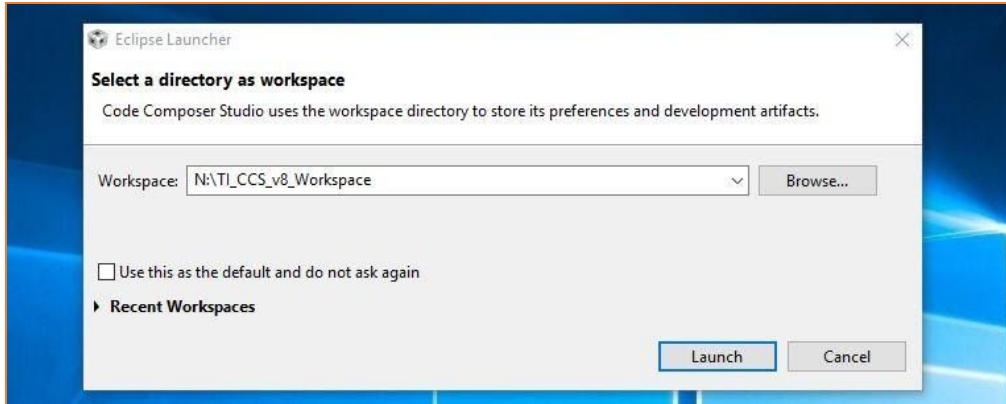


Figure 11 : Remember to choose C: Drive

Project Creation

- To start creating a new project, Go to **File -> New -> Project**

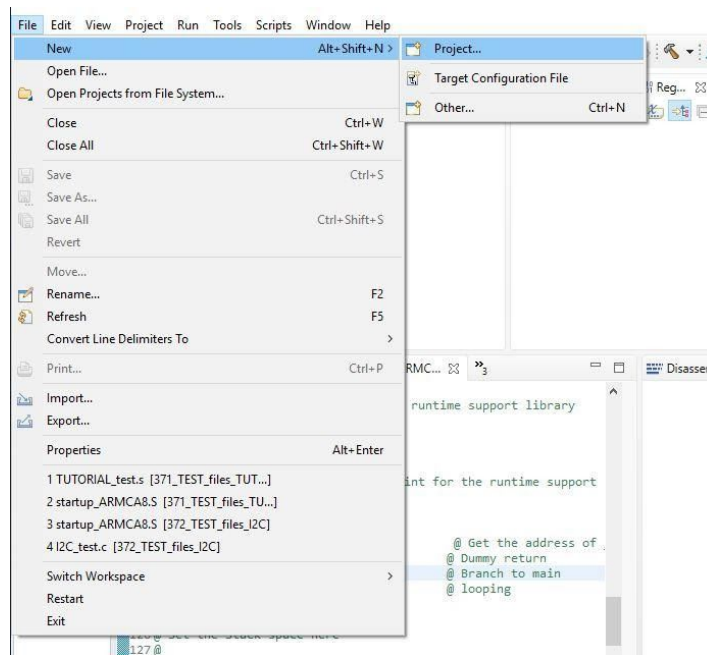


Figure 12 : Creating New CCS Project

- Next choose: **Code Composer STUDIO -> CCS Project** then click **Next**.

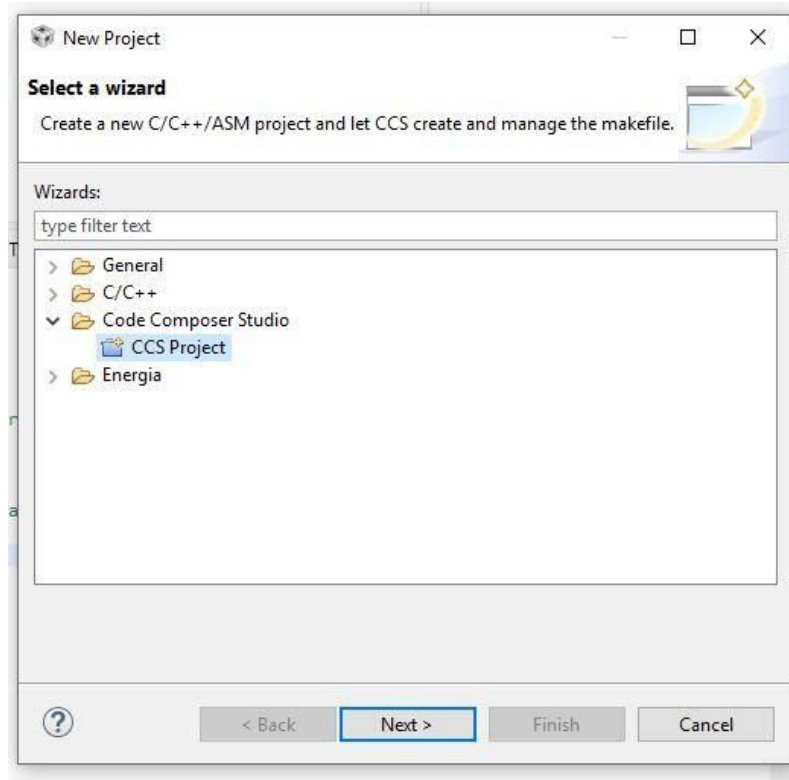


Figure 13 : Select CCS Project from the list

- When the Dialog box appears, select the following settings:
 - Target: AM 33x – Cortex A8 and second option being BeagleBone_Black
 - Connection: Texas Instruments XDS100v2 USB Emulator

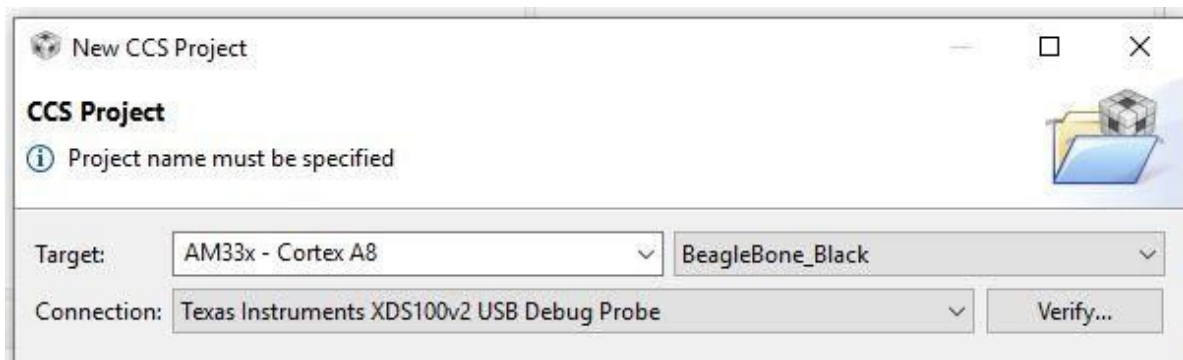


Figure 14 : Selecting Target for CCS Project

- Now, give your project a name (“Sample” in my case). Make sure that the compiler selected is GNU. Refer to the figure.

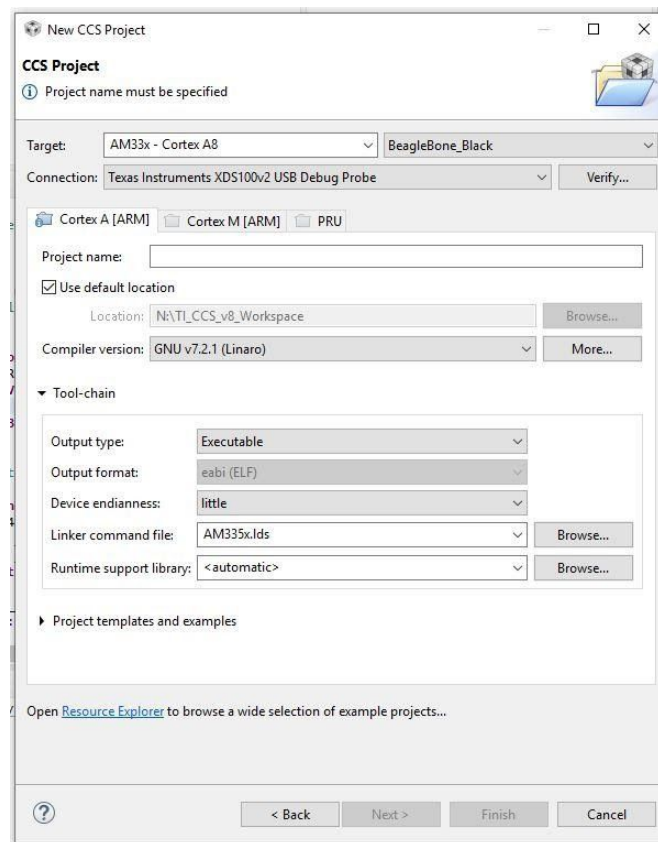


Figure 15 : Different fields in New CCS Project Creation

- Now, Click on Advanced settings which open more options as shown above. Verify for following options:
 - Output type: Executable
 - Device Endianness: Little
 - Linker Command File: AM335x.Lds
 - Runtime Support library: <automatic>
- Now, click on Project templates and examples and select Empty Project under Empty Projects heading and click finish.
Make sure that you are not selecting the “**Empty Project (with main.c)**” option, as it would load the CCS environment suitable for a ‘C Language’ project. *If you accidentally select “**Empty Project (with main.c)**”, simply delete the “**main.c**” file in the project.*

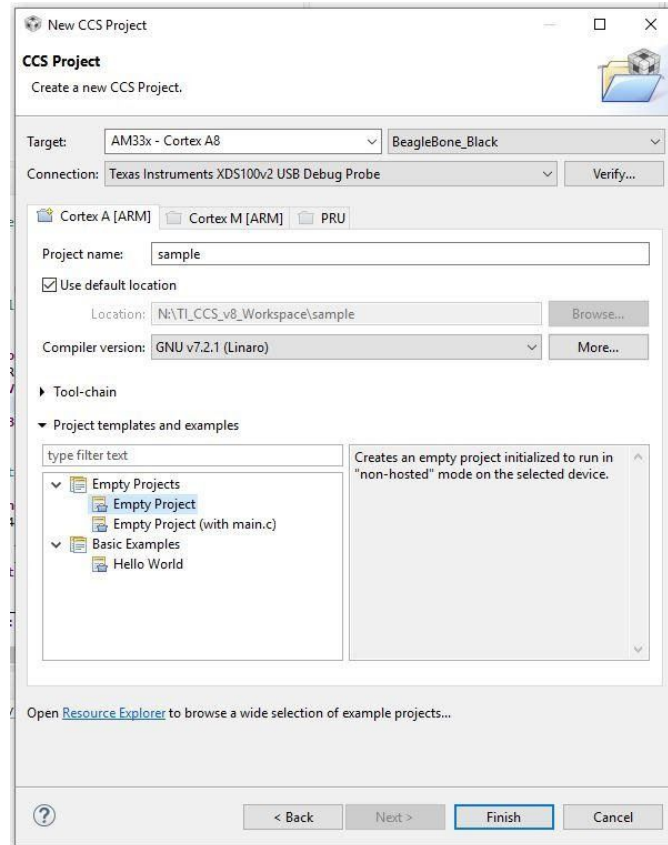


Figure 16 : Project Template

- After this, you can see the project open and listed in the Project Explorer window on the left. (shown below)

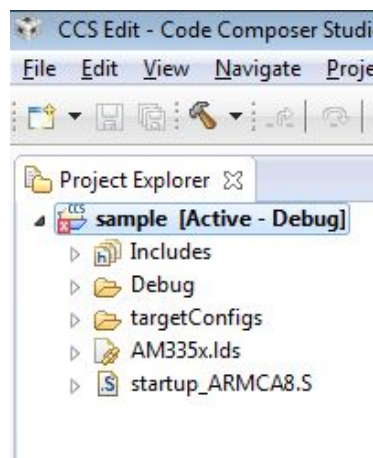


Figure 17: Project Window

- Right click on the project (sample [Active - Debug] here), go to New -> File and then enter the file name of the assembly file you are going to write. The file can have any appropriate name but always give the file a **.s (dot s)** suffix to indicate it is an assembly source code file as shown in the figure below (**Main_sample.s**). After you click Finish, a blank file and edit window, with the file name you created, opens up.

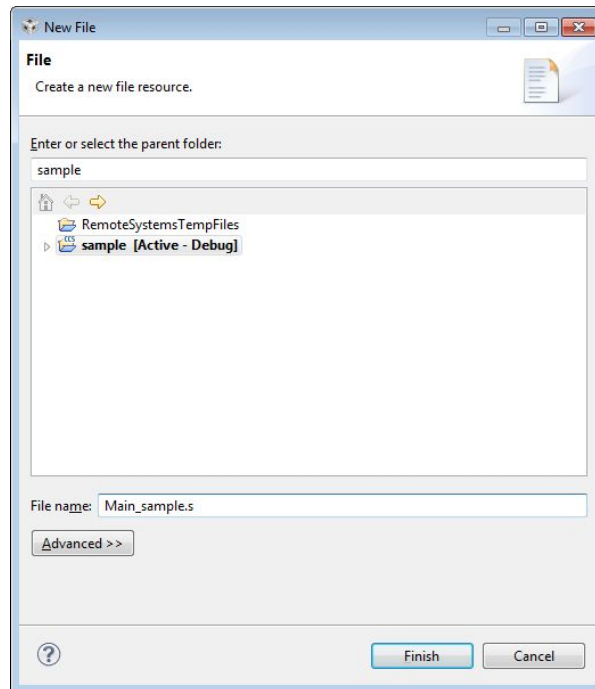


Figure 18 : Creating a new Assembly file (.s)

Writing Assembly files

- Now, you can start writing the assembly code. Enter the Array Multiply program below in the edit window.
Also, remember the following when entering programs:
 - An at-sign (@) begins a comment for the rest of a line.
 - DO NOT use semicolon (;) for comments. It may cause program errors.
 - Provide **header** comments and describe what the program does briefly
 - Always make sure that the labels start in the first column.
 - Provide sufficient tabs for the instruction mnemonics.
 - Always comment what your instructions are doing.
 - Always check carefully for any typing errors before going on to the Assembly operation. This reduces cycling back through edit to fix them later. As explained in the text, the array multiplication program multiplies each half word from the **MULTIPLICANDS** array by the same numbered half word in **MULTIPLIERS** array and puts the result in the same numbered element in **PRODUCTS** array. This program lets you see how loops and labels

work. Pay close attention to how **.text (dot text)**, **.global(dot global)**, **.data(dot data)** are defined in the text.

NOTE: DO NOT COPY AND PASTE!!!

This can cause ascii errors in Code Composer's editor that you can't see and can only be found by deleting all the spaces in the program. It will likely be faster to type in your programs than to debug errors you can't see.

```
@ Array Multiply Program
@ This program multiplies each half word from the Multiplicands
@ array by the same numbered half word in MULTIPLIERS array and
@ puts the result in the same numbered element in PRODUCTS array.
@ Uses R1-R4, R6-R8
@ Douglas V. Hall September 2016

.text
.global _start
_start:
.equ    NUM, 4
        LDR R1,=MULTIPLICANDS    @ Load pointer to MULTIPLICANDS array
        LDR R2, =MULTIPLIERS     @ Load pointer to MULTIPLIERS array
        LDR R3, =PRODUCTS        @ Load pointer to PRODUCTS array
        MOV R4, #NUM             @ Initialize loop counter
NEXT:   LDRH R6,[R1]              @ Load a MULTIPLICAND Half word in R6
        LDRH R7, [R2]            @ Load a MULTIPLIER half Word in R7
        MUL R8, R6, R7           @ Multiply
        STR R8, [R3]             @ Store result in PRODUCTS array
        ADD R1,R1,#2             @ Increment MULTIPLICAND pointer to next
        ADD R2,R2,#2             @ Increment MULTIPLIER pointer to next
        ADD R3,R3,#4             @ Increment PRODUCTS pointer to next
        SUBS R4,#1               @ Decrement loop counter by 1
        BNE NEXT                 @ Go to NEXT if all elements not multiplied
        NOP                     @ Instruction for breakpoint. Does nothing.

.data
MULTIPLICANDS: .HWORD 0x1111, 0x2222, 0x3333, 0x4444
MULTIPLIERS:   .HWORD 0x1111, 0x2222, 0x3333, 0x4444
PRODUCTS:      .WORD  0x0, 0x0, 0x0, 0x0
.END
```

Assembling, List File Generation, and Linking

Once you have carefully entered the Array Multiply program and saved it, the next step is to **Build** the program. The first step in a build is to assemble the source file to create object code (**.o (dot O)**) file. Then, if the assembly is successful, the next step in the Build is to **link** the **.O** file to produce an executable (**.out(dot out)**) file. Here are the steps for this process.

- First you need to tell the **Linker** not to include startup files. This can be done by right clicking on the project name in the **Project Explorer**, and then clicking **Properties** (or press both **ALT** and **ENTER** keys at the same time on your keyboard). In response, a dialog box appears.
 1. Select **Build** and expand **GNU Linker** and expand **Basic** (**Build** ☐ **GNU Linker** ☐ **Basic**)
 2. Tick the first check box which says “**Do not use the standard system startup files when linking**”
 3. Click Ok.
- Next, you may optionally insert your project name in the Output file box as, for example, sample.out and insert your project name in the map file box as, for example, sample.map and click OK. If left as depicted in Figure 16, CCS will substitute the project name for you.

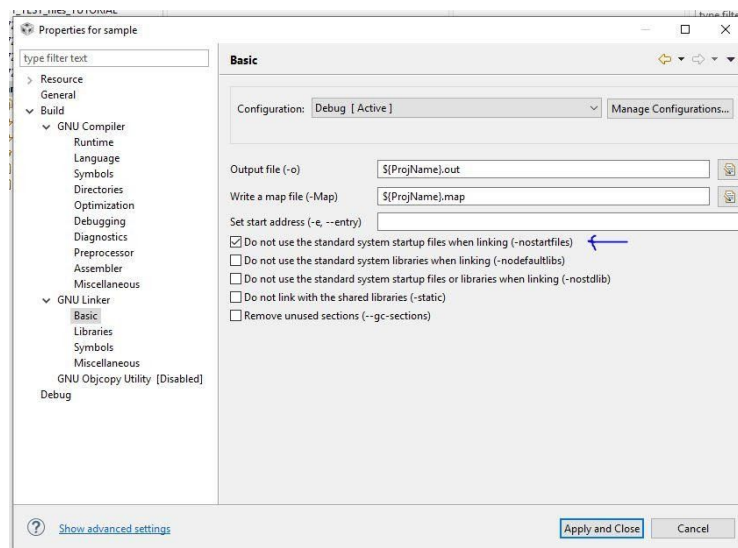


Figure 19 : GNU Linker Option

- Now for generating List files, reopen the Project Properties or by right clicking on the project (or **ALT+ENTER**) and then clicking properties, and go to **Build -> GNU Compiler -> Assembler**. Now at Other Assembler flags, click the add button (green plus on file) and enter
-a=(filename).lst (dash a equals filename.lst) replacing (filename) with your filename. For example, the
-a=Main_sample.lst in the screenshot below.

This will generate a list file in the directory of your project. i.e. if your Project directory is e.g. N:\TI_CCS_v6_Workspace\sample, then the list file can be found at:
N:\TI_CCS_v6_Workspace\sample\Debug\sample.lst

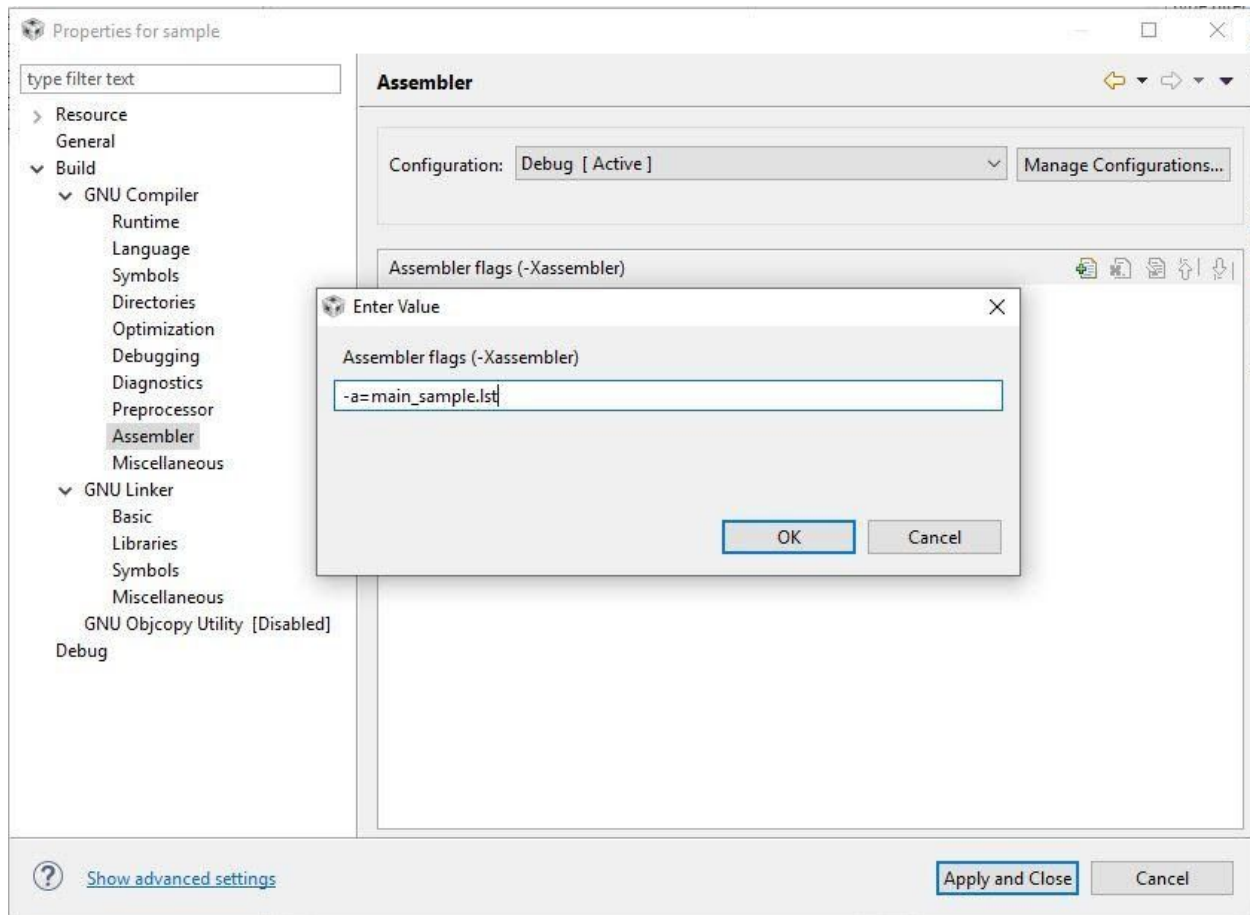


Figure 20 : Setting a list file

- In order to see the compiler log, click on **view -> console**. Now you are ready to build the project.
- Right click on the project name in Project Explorer and click on '**Build Project**' (Alternatively, click on **Project** tab and select Build Project: **Project->Build Project**). You should be able to see some information about the compilation in Console. If everything is right, the compiler successfully produces a file called, for example, sample.out.
 - If you have made any errors, it reports them both in the **Console** and the **Problems** window.
 - Check for the errors and try to clear them one by one in the edit window. Always start with the first error, since subsequent errors may be a result of the first. Cycle through the edit-ReBuild loop until the Build is successful. The troubleshooting hints below may help.

- You should now have a successful build with a sample.out executable file and a sample.lst list file in your project directory. You can also find your list file at the directory as specified.

Troubleshooting Hints

- If you encounters errors such as:

```
../Main_sample.s:25: Error: bad instruction `instruction for breakpoint. Does nothing.'
```

Make sure that there are no ';' for comments instead of '@'

- ../Main_sample.s:25: Error: bad instruction `no'
Such errors might occur due to illegal instructions.

Configuring Emulator to be Ready to Load and Run Program

Now that you have your program built successfully, you need to set the emulator configuration to help it initialize the processor on the board, so you can load your program, run it, and debug it.

- To open the emulator configuration, go to **View -> Target Configuration** (as shown in Figure 18)
A window should open at the right.
 - Open the folder **Projects -> Sample (i.e your folderName) -> targetConfigs** .

Here you should already find a **BeagleBone_Black.ccxml** file being set to default. This should be the case if you have created the project successfully by following the steps as specified.

If somehow, the **.ccxml** is not there or got deleted, you can right click on the **User Defined** folder and select **New Target Configuration**. On the dialog box enter a name of your choice with a **.ccxml** extension. Now you can see the file being created under the User Defined folder. To link the **.ccxml** file to your project, right click on the **.ccxml** file and then select **Link File to project -> sample (i.e, your project name)**.

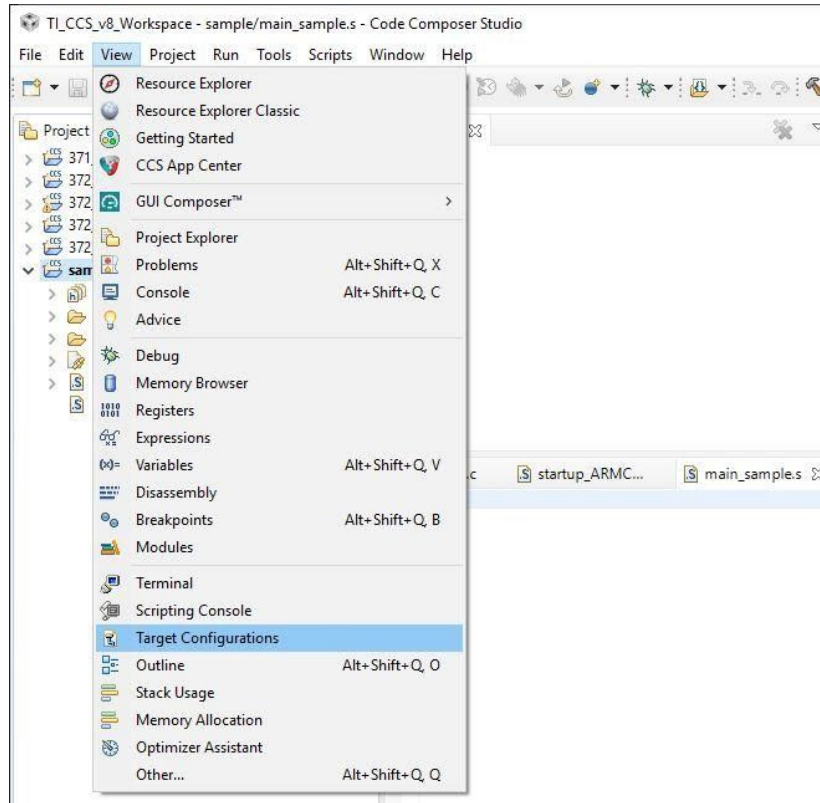


Figure 21 : View target configuration

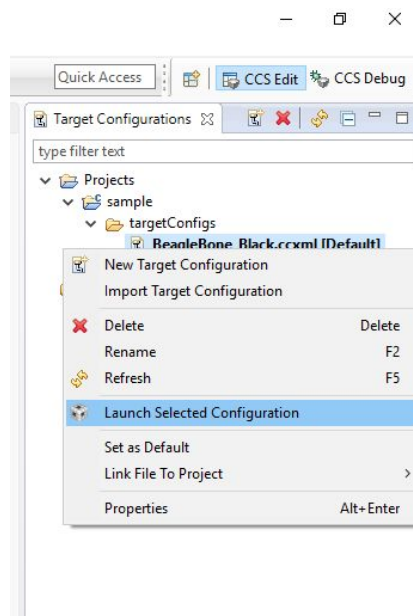


Figure 22 : Launch selected configuration

Launching the Configuration and Loading Your Program

- Once the Setup of the emulator is done, we need it to load the program we have built using the tool chain. To do that, go to **View -> Target Configurations** (shown in Figure 18). There select the **.ccmxl** file we have configured, right click and select “**Launch Selected Configuration**” (see Figure 19)
- The entire window structure changes to debug view. This can be seen at the Top Right Corner as the windows perspective changes from CCS Edit to CCS Debug.

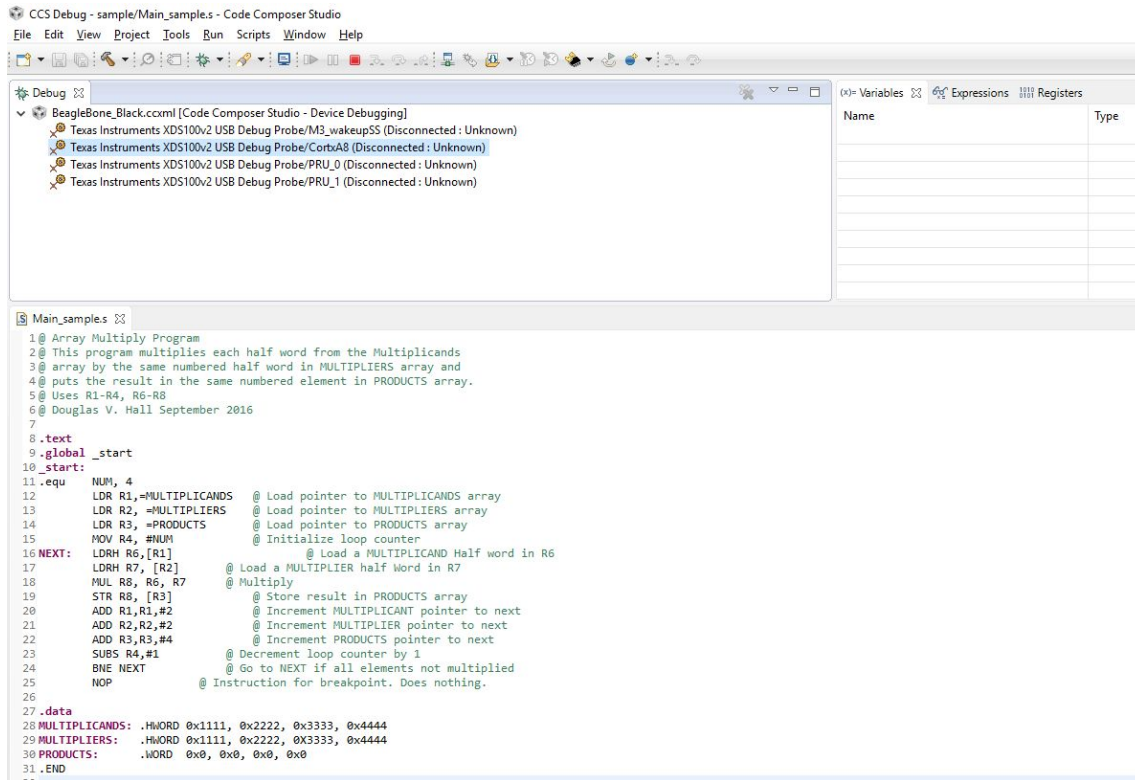


Figure 23 : Debugging a program

- Now in the Debug windows, select **TI XDS100v2 USB Emulator/CortexA8**. You can see that it is in a disconnected state. Right click on this entry and select “**Connect Target**”. This causes the emulator to connect to the BBB and initialize it with the configuration provided in the .gel file mentioned previously. A new window ‘Console’ pops up at the bottom showing the configuration progress and finally that AM335x initialization has been done. After this, you can see in the Debug window that the state of CortexA8 entry changes to Suspended.
- Now your program can be loaded for execution by going to the Menu bar and selecting **Run -> Load -> Load Program** (shown in Figure 21) At the Dialog box that pops up, click Browse and browse to the location of the project. It will be inside your Workspace directory. Inside the Project, the executable .out file can be found in the Debug directory. Once it is selected and you click OK, the emulator loads it into the processor.

The source file of the Startup_ARMCA8.S pops up onto the screen. The processor is halted at the entry point in the startup_ARMCA8.S file.

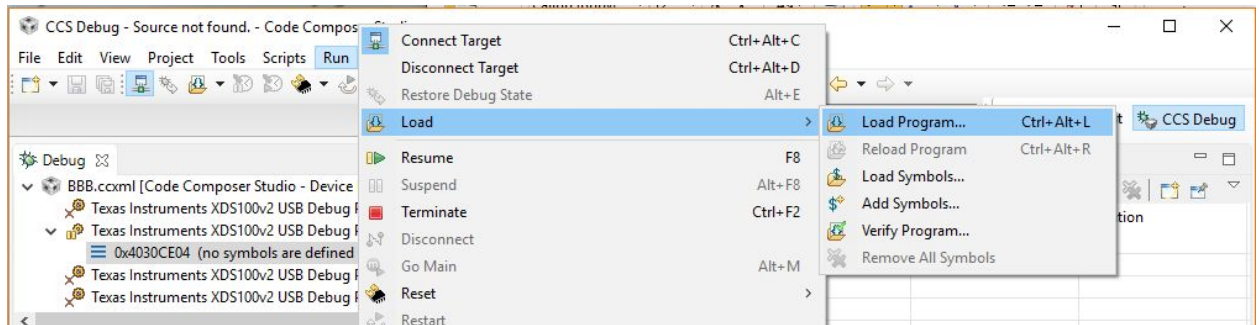


Figure 24 : Load the .out file

- Do not be worried by the following info shown in the log. This warning is normal for assembly programs.

```
CortexA8: AutoRun: Target not run as the symbol "main" is not defined
```

Debugging Your Program

Debugging Options

You have now written and compiled your program, configured the emulator to enable you to load your program, and loaded your program into the B3 board. As you can see on the screen and below, after the program loads, execution stops at the Entry section of startup_ARMCA8.S file as shown in the figure below. There are two basic ways that you can execute the startup code and your program from this point. You can single-step through the program or you can set a breakpoint at some later point in the program and tell the debugger to execute the instructions up to that breakpoint. We will start with showing you how to single step one instruction at a time and observe the contents of registers and memory locations at each step, so you can follow the action.

```
77 Entry:
78 @
79 @ The stack for all the modes (Abort, FIQ, etc.) is set by
80 @ the runtime support library.
81 @
82
83 @
84 @ Set up the Vector Base Address Register
85 @
86 LDR r0, =_isr_vector
87 MCR p15, 0, r0, c12, c0, 0 @ Write VBAR Register
88
```







Figure 25: Beginning of Debug process

Using single stepping to run and debug a program

- For this method, you need to first get familiarized with the debugging buttons on the toolbar of CCS as shown below.



Figure 26 : Debug options

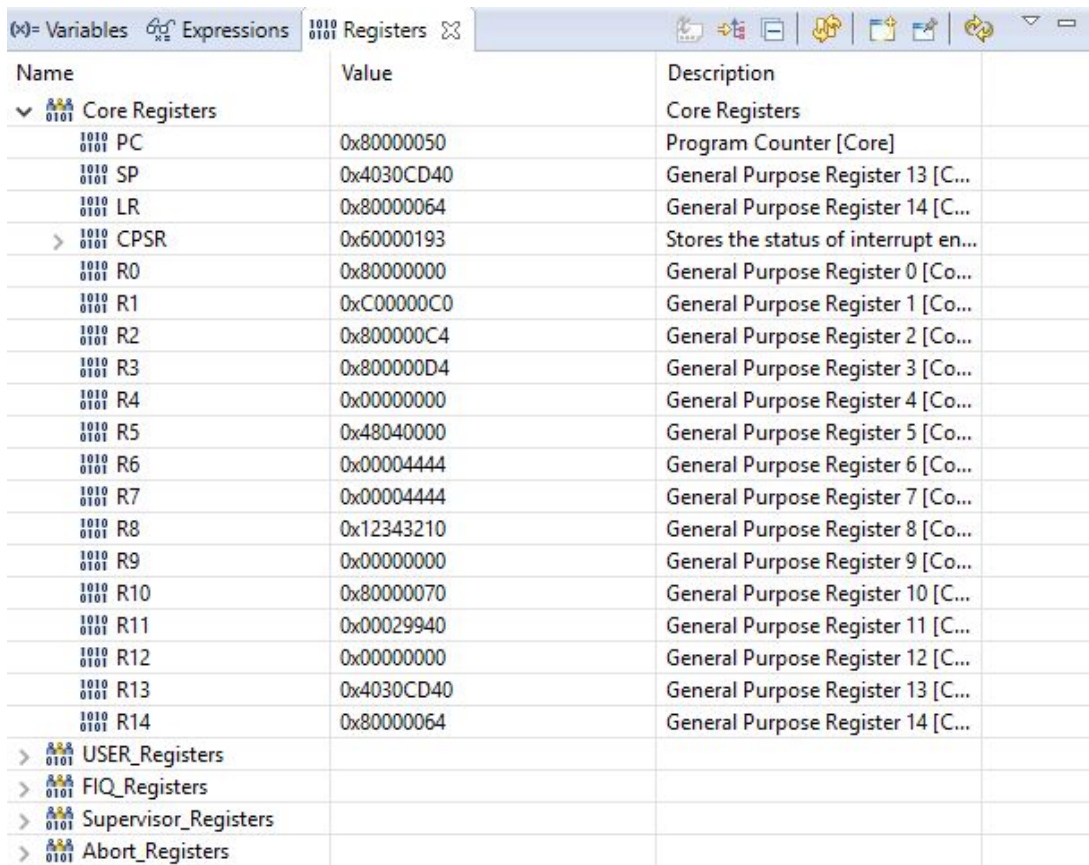
-  The first button is the **restart** button, which is useful to restart the execution of the program you loaded into the processor, if you want to debug it again from the start.
-  This is a '**set breakpoint**' button which is used to set breakpoints at a specific point in the program or in memory. You will get to use this a little later.
-  These green arrow buttons are '**Assembly Step Into**' and '**Assembly Step over**' respectively. The Assembly step into button steps over one instruction at a time and if it encounters any procedures in the code, it steps into the first instruction of the procedure. The Assembly step over button also steps over one instruction at a time, but it doesn't switch to procedures and executes all the instructions within a procedure and steps over to the next instruction of the program.
-  This button is the 'refresh debug views' button, which resets all the windows to a default state. It is not necessary for your debugging.
-  These buttons with orange arrows are 'Step Return', 'Step over' and 'Step Into' buttons. These buttons work similar to the Green buttons mentioned above except that these program flow buttons are usually used to debug C/C++ programs. In our case of assembly language programming, these buttons usually have the same behavior as the green arrow assembly step into and step over buttons. The Step return button is usually greyed out.
-  These buttons are Session control buttons. They are respectively Resume, Suspend and Terminate. Pressing the Resume button tells the debugger to execute instructions until a breakpoint is reached or until the suspend button is clicked. We will use this button after we show you how to insert a breakpoint. The Suspend button is used to stop a CPU which is executing a program. The terminate button terminates the debugging session.

Now that you know all the buttons, you can reach the start of your program by slowly pressing the green Step Into button until you reach the first instruction of your program at **_start**.

Check on values in Registers, Memory and Disassembly

As you single step through your program you would like to observe the effect of each instruction on the Registers.

- To see the registers displayed, go to **View-> Registers**. When the **Register** list pops up, select **Core Registers**. You should then see the familiar **R0-R14** registers (shown in Figure 24) with the value in each displayed. Values are updated as a program runs. (The values of flags and other bits in the CPSR register can be seen by expanding the CPSR register inside the register window.)



The screenshot shows the 'Registers' window in ARM DevStudio. The 'Core Registers' section is expanded, displaying a list of registers with their names, values, and descriptions. The registers are R0 through R14, plus PC, SP, LR, and CPSR. The CPSR register is highlighted with a green arrow. Below the Core Registers, there are sections for USER_Registers, FIQ_Registers, Supervisor_Registers, and Abort_Registers, all of which are currently collapsed.

Name	Value	Description
Core Registers		Core Registers
PC	0x80000050	Program Counter [Core]
SP	0x4030CD40	General Purpose Register 13 [C...
LR	0x80000064	General Purpose Register 14 [C...
CPSR	0x60000193	Stores the status of interrupt en...
R0	0x80000000	General Purpose Register 0 [Co...
R1	0xC00000C0	General Purpose Register 1 [Co...
R2	0x800000C4	General Purpose Register 2 [Co...
R3	0x800000D4	General Purpose Register 3 [Co...
R4	0x00000000	General Purpose Register 4 [Co...
R5	0x48040000	General Purpose Register 5 [Co...
R6	0x00004444	General Purpose Register 6 [Co...
R7	0x00004444	General Purpose Register 7 [Co...
R8	0x12343210	General Purpose Register 8 [Co...
R9	0x00000000	General Purpose Register 9 [Co...
R10	0x80000070	General Purpose Register 10 [C...
R11	0x00029940	General Purpose Register 11 [C...
R12	0x00000000	General Purpose Register 12 [C...
R13	0x4030CD40	General Purpose Register 13 [C...
R14	0x80000064	General Purpose Register 14 [C...
USER_Registers		
FIQ_Registers		
Supervisor_Registers		
Abort_Registers		

Figure 27 : Register Entries

- You would also, perhaps, like to see a display of the memory used by your program, so you can see when a result gets written to memory. Go to **View->Memory**. Alternatively, you can right click on a register, and select “**View Memory at Value**”. When the Memory window pops up, enter the address 0x80000000 (as shown Figure 25)

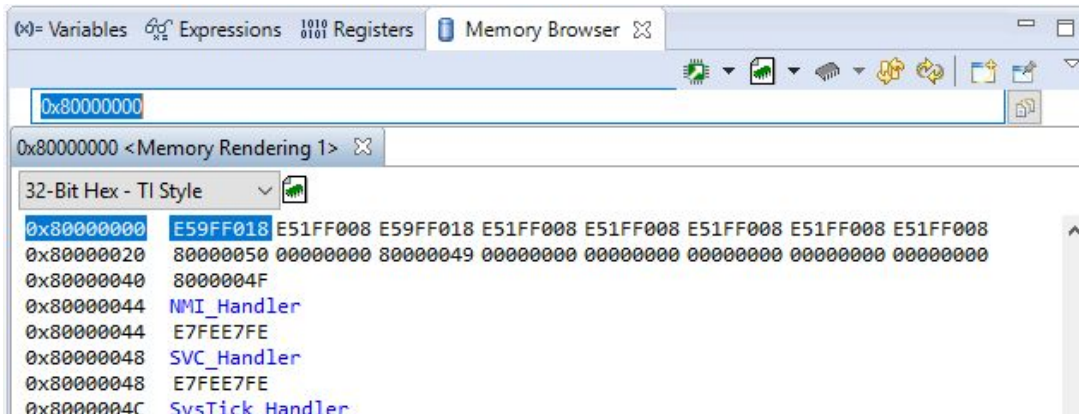


Figure 28 : Memory Content for a given memory address

- To see a display of the disassembly of your program you can go to **View->Disassembly**. When the Disassembly window appears, enter the address 0x80000000 in the box and hit Enter (shown in Figure 26). You should see `_START` and the first instruction of your program at 0x80000070.

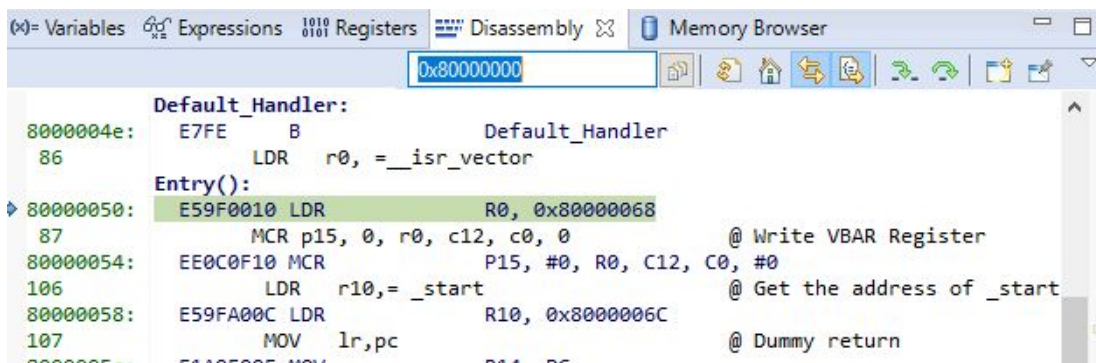




Figure 29 : Disassembly shows program execution

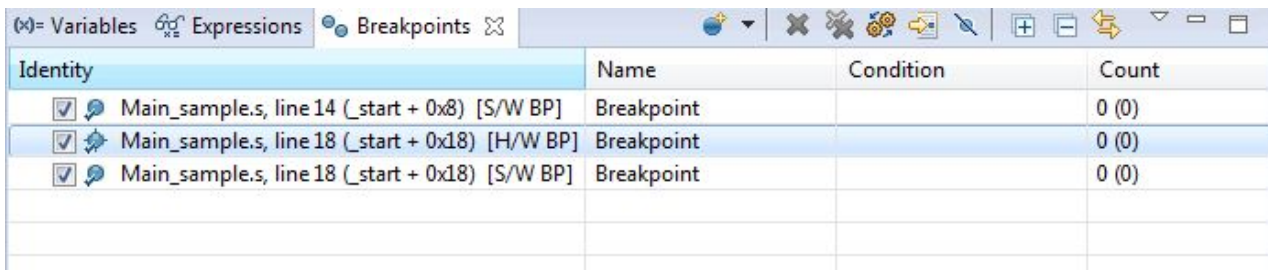
- Now use the green Step Into button  to step through a few instructions and observe the values produced in the registers. If you step past the first STR instruction, you should see the first result written to memory. Don't step all the way to the end of the program. We will show you how to insert a breakpoint at the end of the program and run to that breakpoint.

Breakpoints

Breakpoints in code cause the program execution to pause when the breakpoint location is reached. You can check for changes in registers or memory while you are paused at a breakpoint and then continue to a later breakpoint. There are generally two kinds of breakpoints : Hardware breakpoints (which use the emulator) and software breakpoints (which are a function of the CCS IDE). **For this course we use hardware breakpoints instead of software breakpoints**, because software breakpoints are not fully supported in our setup and cause errors.


Setting and Removing Hardware Breakpoints

- As an example of how to set a hardware breakpoint, put the cursor on the NOP instruction at the end of your program. Right click on that line and select **Breakpoint (Code Composer Studio) -> Hardware** breakpoint. After that you can see a Hardware breakpoint icon on the left side of the line number for that line.
- Now, press the Resume button and you can see that the execution continues and stops at the NOP instruction. You can then examine the contents of registers and memory to see if they are as expected.
- If you insert a breakpoint earlier in the program, you can press  **Resume** to run to that breakpoint and then single-step or Resume to reach the next breakpoint. These features allow you to check if your program is working correctly at each step and if not, debug the program, fix any error and run through the Build-Debug cycle again.
- To see all the breakpoints in a list you go to **View->Breakpoints**. To remove or to disable a breakpoint, you select the breakpoint, right click to bring up a menu and click on the desired action, disable, or delete.




Identity	Name	Condition	Count
<input checked="" type="checkbox"/> Main_sample.s, line 14 (_start + 0x8) [S/W BP]	Breakpoint		0 (0)
<input checked="" type="checkbox"/> Main_sample.s, line 18 (_start + 0x18) [H/W BP]	Breakpoint		0 (0)
<input checked="" type="checkbox"/> Main_sample.s, line 18 (_start + 0x18) [S/W BP]	Breakpoint		0 (0)

Figure 30 : Set hardware breakpoints

- Note that if you want to run the program from the start again at any point, just click on the toolbar Restart  button.

Setting the entry point to the starting label in your program file

- If you are tired of single stepping to the first instruction of your program every time, you can set a breakpoint at the first instruction and click **Resume**  Alternatively, you can directly set the execution to start at the first instruction of your program by simply inserting **_start** at the right place in the linker setup. To do this, right click on the project, select Properties, and in the Project properties, Under GNU Linker, Basic, enter the label **_start** at Set starting address option as shown in Figure 28. Then, after you load your program, you will see that the execution is set to start at the first line of your program instead of in the startup program.

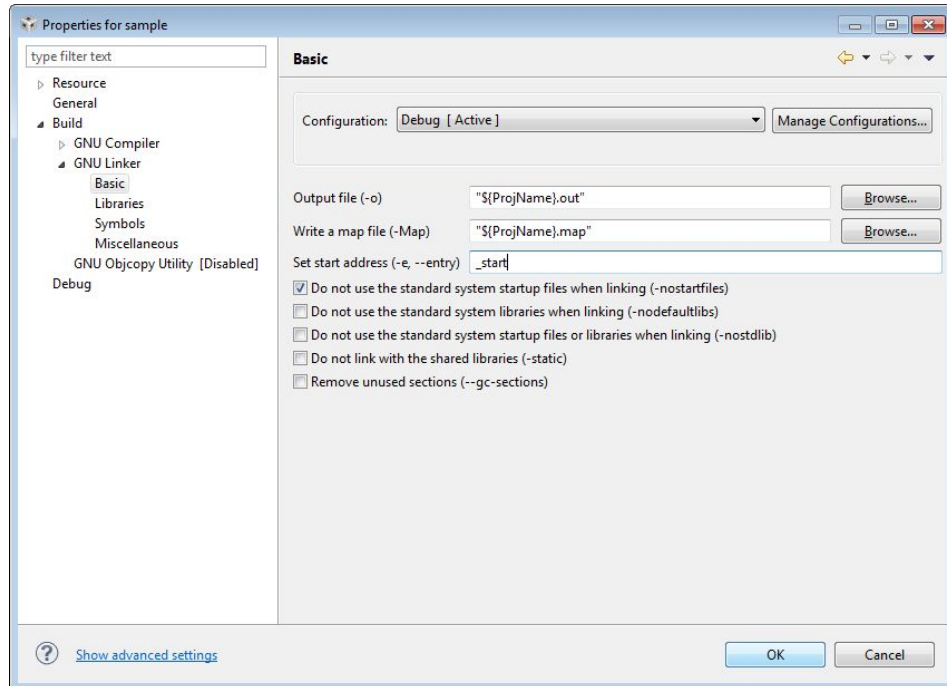


Figure 31 : Setting Breakpoint at _start

- During the course of debugging your program, you might need to make changes to your program code. For that, you can switch to the CCS Edit (shown in Figure 29) window by clicking on the **CCS Edit** tab in the upper right corner of the CCS window.

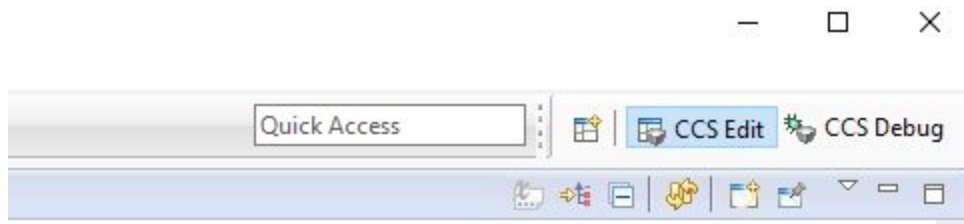


Figure 32 : Debug window layout

- To rebuild your program after you have corrected and saved it, right click on the project, and select Rebuild project. If the project builds correctly, a dialog box as shown below, then pops up and asks if you want to reload the program automatically. Click on yes.

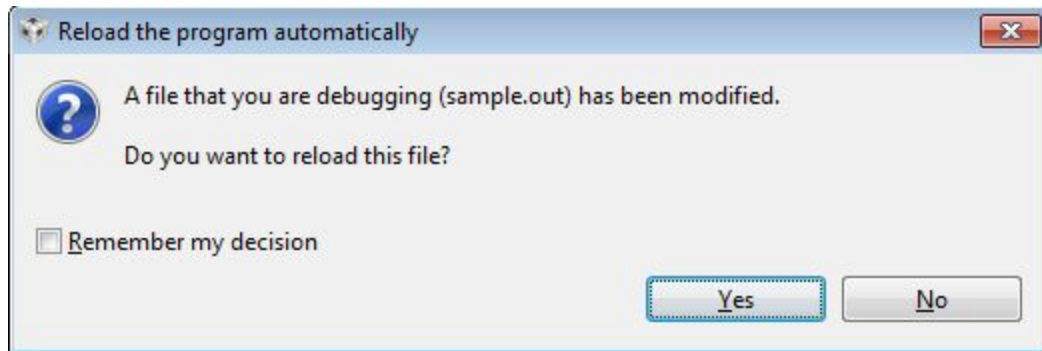


Figure 33 : Auto program reload option

- Now, you can change to the Debug with the CCS Debug tab and start running and debugging again.

References

- BeagleBone Black System Reference Manual:
<https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual>
- BBB wiki: <http://elinux.org/Beagleboard:BeagleBoneBlack>
- Emulator: <http://www.blackhawk-dsp.com/products/usb100v2D.aspx>