LAB 01
Basic Debugging Techniques

Goals:

• Fix the broken Assembly Code. The program should blink the Green LED every half second, but the Red LED is steadily lit.
• Step through code, line by line; observe the program flow and data stored in the General Purpose and Status Registers.
• Time an iteration of the delay subroutine using the CPU timer and breakpoints.

Bonus:
Rewrite the code so the Red and Green LEDs toggle back and forth every second. +10

Pre Lab Questions:

• What 5 General Purpose Registers are being used and how do they affect the program?
• What Status Bits in the Status Register are you looking at?
• How long in clock cycles does one iteration of the DELAY subroutine last?
• What changes to the code did you make to correct the Assembly Program?
Basic Debugging Techniques

Lab Guide

The Assembly Program is supposed to blink the Green LED every half second but the Red LED is steadily lit instead. Use the following Basic Debugging Techniques to find the cause of the program:

**Line Stepping**

Click the Hammer icon to build the test code. This is a good way to check for any errors before debugging. Once the code is successfully built, click the Green Bug icon to start the Debugging Process. You should now see new set of icons:

You’ll notice a Play/Pause and Stop Button. Next to the Stop button are two Yellow Arrow buttons; Step Into and Step Over, shortcut keys F5 and F6 respectively.

Pushing the Play/Pause button will cause the program to run in real time; **DO NOT** do this yet. Pushing the Stop button will cause the Debugging process to stop; also **DO NOT** do this yet.
For now we will only concern ourselves with the Step Into button. You will notice the program is stopped at the first executable line of code, this case, line 38. This is denoted by a Blue Arrow; this line is the next line to be executed (as of now, this line has NOT been executed). If we push the Step Into button, the currently highlighted line will be executed and the next line will be denoted by an arrow.
Register Inspection

You should notice some tabs labeled **Variables**, **Expressions**, and **Registers**. Click the **Registers** tab. In this Lab we will concern ourselves with the **Core Registers** and **Port_1_2** items.
Click the Grey Chevron on the left of Core Registers and you will see the Program Counter, Stack Pointer, General Purpose Registers R4 through R15 as well as the SR or Status Register. Note that although it is not labeled as such, R3 is used as a Constant Generator Source Register; long story short, you cannot use it as a General Purpose Register.
Clicking the Grey Chevron on the left of **Status Register** will open show us the **Status Bits V, N, Z and C**. Do not worry about the other bits for now. You can check the **Port_1_2 Registers** in a similar manner.
Any changes in the value stored in a register will be highlighted with Yellow.
Breakpoints

You may want to skip to certain part of your code. This is where **Breakpoints** come in handy. Above, the program is stopped at line 56. Let’s skip toe line 62 by setting a **Breakpoint**. We do this by simply double clicking the empty space to the left of line 62; a **Blue Orb with a Check Mark** will appear.

We are now paused at line 62; all previous lines were executed in real time. **Breakpoints** can be set before or during the Debugging Process but it is recommended that you set **Breakpoints** while you are debugging. **Breakpoints** will be saved for each session. If there was an error in the code or if you changed the code such that the **Breakpoint** now points to an empty line or a different instruction, the **Breakpoint** will disappear.
CPU Timer

We can use Breakpoints with the CPU Timer to time how long a segment of code takes to execute in clock cycles.

On the menu, Click Run > Clock > Enable. A Small Yellow Clock will appear somewhere on your window, usually on the bottom right hand corner.
Set two **Breakpoints**; one on each of the first and last lines of the code you want to time. Push the Play/Pause button.

As you can see, the code took 3 clock cycles to execute. To **Reset the CPU Clock** and start timing from zero, double-click the **Small Yellow Clock**.
Restarting the Program

You can restart the program without having to stop the Debugging Process and rebuilding the Project. Either use the **Hard** or **Soft** Reset Buttons. The only time you have to stop the Debugging Process is if you need to change the code.

**Notes:**

- If you are certain your code doesn’t have any errors, you can simply click the **Green Bug** icon to build and start the Debugging Process.
- In Assembly Projects, **Step Into** and **Step Over** do the same thing; **Step Over** will become useful in C Projects.
- When **Inspecting Registers**, you can switch between different representations of the value stored in the register. Simply right click the value and choose a **Number Format**. Additionally, you can even change the value stored in a register on the fly!
- You cannot view data stored in a register if the program is running in real time; you must pause the program first.
- **Breakpoints** can have conditions set to determine when to pause at the breakpoint. You can also update the view of the Registers after each change at a **Breakpoint**. Right click on a **Breakpoint** and choose **Breakpoint Properties...** to play with the different **Breakpoint** settings that are available.
- **Code Composer** does not run the program in real time when using the **CPU Timer**. As such, when timing a segment of code, execution will take much longer. If you are trying to time a loop, its best to time one iteration of that loop and make an educated guess as to how long the entire loop will take to execute given the number of iterations in that loop. If you try and time the entire loop, **YOU WILL BE WAITING FOREVER**.
Basic Debugging Techniques
Lab Hints

There are many ways to solve this problem and all are correct as long as the desired behavior is achieved. However, there are three deliberate errors, which if corrected, will result in the micro behaving as desired.

It is recommended that you study how the program is behaving at first to see if you can discover the simple errors.

Lookup and study the operations in the program to gain insight on how the data is being used in the program.

**General Purpose Registers (R4 – R15)** are 16 bits while the **Port Registers** are only 8 bits, with each bit corresponding to a pin on the micro (BIT 0 corresponds to Pin P1.0, so on and so forth). Look at your micro and the datasheet to see what pins the **Red** and **Green** LEDs are connected.
Lab Notebook Questions:

- What is the purpose of being able to step through your program line by line?
- If we have a 'jz' command that is not functioning as expected, where would be a good place to look for troubleshooting purposes? (Hint: What register would we be looking at; why?)
- What is the maximum value we can store in a General Purpose Register? What happens if we try to store a value larger than this?
Basic Debugging Techniques
Frequently Asked Questions

*How long is a Clock Cycle?*

It depends on how the micro is set. The MSP430’s CPU can be set anywhere from 32kHz to 16MHz. By default, the MSP430 is set to 1MHz which is useful since that means 1 clock cycle is 1µs in real time.

*I see Green Arrows that say Assembly Step Into/Over; why aren’t we using those for this lab if the project is in Assembly?*

Those arrows may become useful when we start using C projects. The Yellow Arrows will function the same, however we can use the Green Arrows to see how **Code Composer** is translating our C code into Assembly and step through this code.

*Where are Registers R0, R1 and R2?*

R0 is the **PC (Program Counter)**, R1 is the **SP (Stack Pointer)** and R2 is the **SR (Status Register)**. R3 is used as a **Constant Generator Source**, meaning that the compiler will use it to store commonly used values in order to save a few clock cycles here and there.