LABORATORY MANUAL

CE2107
Microprocessor System Design and Development

Lab Experiment #1

Microprocessor System Development Toolchain
C-Programming for Embedded Systems

SESSION 2022/2023
SEMESTER 1

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

This lab session will be done in groups of 2. The group will work together to complete the lab exercises.

1. OBJECTIVES

1.1 Familiarise with the development tool chain for Microprocessor Design

1.2 Familiarise with the Major Features of an Integrated Development Environment Application

1.3 Understand the startup sequence of a microprocessor

1.4 Load and modify sample assembly and C program to test the hardware

2. LABORATORY

This experiment is conducted at the **Hardware Lab 2** at **N4-01b-05 (Tel: 67905036)**.

3. HARDWARE EQUIPMENT

- A Windows-based computer (PC) with a Universal Serial Bus (USB) port.
- Texas Instruments Robotic System Learning Kit (RSLK-MAX)
- A USB A-to-MicroB cable.
- Oscilloscope.

4. ACKNOWLEDGEMENT

This lab reference and leverage from the work done by Dr Jonathan Valvano on the RSLK-MAX. The original source (sans solution) can be downloaded online under filename slac799a.zip. Students can also download the original lab notes (slay052a.pdf) for reference but note that adaptation had been made so there are differences in information, instructions and tasks. The original RSLK Max workshop is also available online at https://university.ti.com/en/faculty/ti-robotics-system-learning-kit/ti-rslk-max-edition-curriculum

5. REFERENCES (Can be found in the Doc sub-folder)

[1] Wk1-2 Lecture Notes

[2] RSLK-MAX Construction Guide (sekp164.pdf)

[3] MSP432 Launchpad UG (slau597f.pdf)

[4] MSP432 TRM (slau356h.pdf)

[5] MSP432 Datasheets (msp432p401r.pdf)

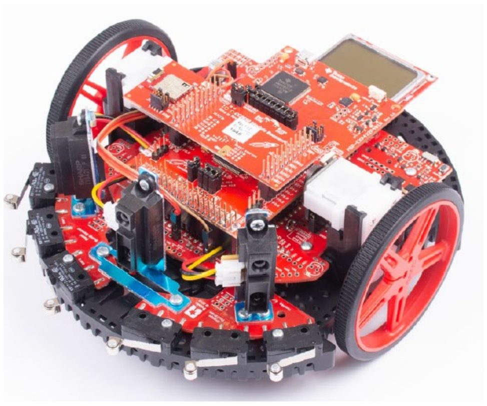
[6] ARM Optimizing Compiler UG (spnu151r.pdf)

[7] ARM Assembly Language Tools UG (spnu118u.pdf)

[8] Cortex M3/M4F Instruction Set (spmu159a.pdf)

6. INTRODUCTION

The Texas Instruments Robotic System Learning Kit (RSLK-MAX) consists of a MSP432 Launchpad. MSP432, an ARM Cortex M4F processor, act as a main controller to a robotic system consisting of DC motors and various sensors such as IR Sensor for measuring distance to an obstacle, reflectance sensor to detect the presence of a colour line below the robot, bump sensor to detect collision with obstacles and a tachometer to measure the rotation speed of the motor. On top of that, there are also switches and LEDs to provide means communication between user and the RSLK-MAX. Through this setup, various microcontroller hardware/software concepts can be learned, these include GPIO handling, exception handling, pulse width modulation (PWM), timer compare and capture operations, analog-to-digital converter (ADC), asynchronous serial communication and finite-state machine design.



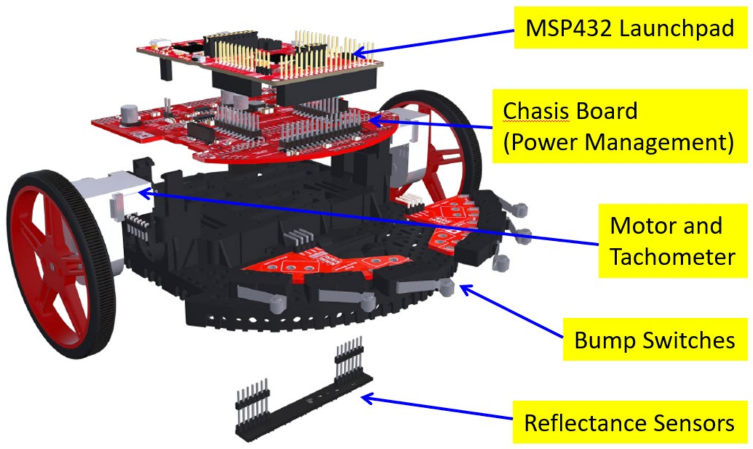


Fig. 1: Texas Instruments Robotic System Learning Kit (RSLK-MAX)

6.1 Power Management

The RSLK-MAX hardware is powered via batteries. The MSP430 Launchpad could also be powered via USB so it could still be programmed even if the battery power is not ON. Note that the 5V from USB is disconnected from the rest of the RSLK-MAX hardware and it should remain so in order not to damage the hardware. **Please check that the 5V header on the MSP430 Launchpad is not connected**. This is to allow USB and Battery power to be ON simultaneously. See Figure below.

Warning: Please ensure the +5V jumper on the MSP432 LaunchPad is disconnected or removed. Not removing this jumper will cause permanent damage to the LaunchPad and the TI-RSLK chassis board.

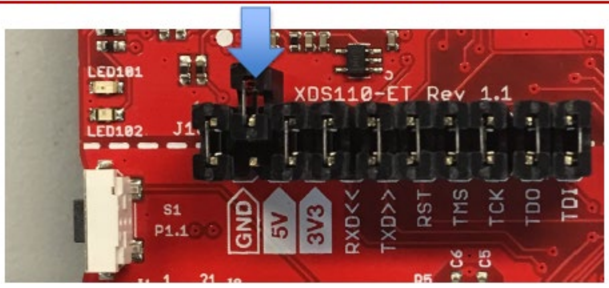


Figure 2a

The RSLK-MAX is powered by 6x AA batteries. Leave the **Slider** switch in **OFF** position. Use the **Power** Button to toggle the battery power ON/OFF. See figure below.

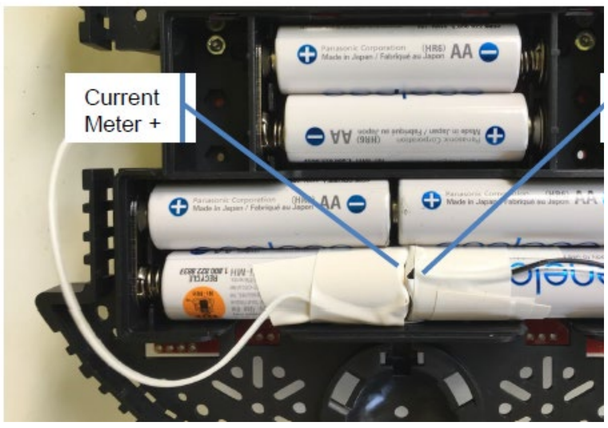


Figure 2b

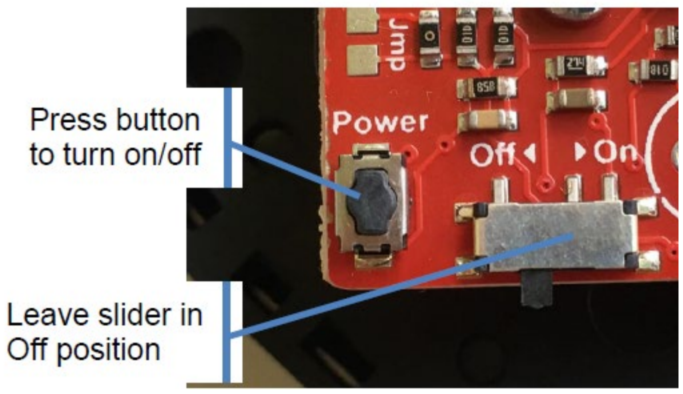


Figure 2c

6.2 Connecting the RSLK-MAX to the PC

Connect the USB cable to the MSP432 Launchpad board and the PC. On the PC, open the *Device Manager*. Scroll to *Ports (COM & LPT)* to check that the two Virtual COM ports for the RSLK-MAX and the two XDS110 drivers (for Emulation and Debug) are enumerated properly (See Figure 2e).

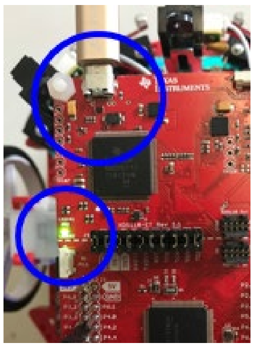
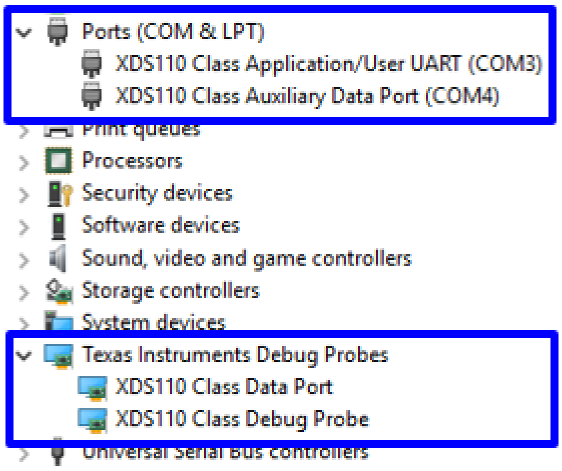


Figure 2d



Ports (COM & LPT)
XDS110 Class Application/User UART (COM3)
XDS110 Class Auxiliary Data Port (COM4)
Print queues
Processors
Security devices
Software devices
Sound, video and game controllers
Storage controllers
System devices
Texas Instruments Debug Probes
XDS110 Class Data Port
XDS110 Class Debug Probe
Universal Serial Bus controllers

Figure 2e

6.3 Code Composer Studio

The first step to any embedded development is to set up the software development environment we plan to use once the hardware has been chosen. It is often popular and wise to choose an Integrated Development Environment (IDE). An IDE can have a list of features that aid in the ease or speed of software development. In the hardware context, this could include providing critical debugging information needed to understand the memory usage and performance of the software on the processor.

Code Composer Studio (CCS) is an industry-ready IDE option that is provided by Texas Instruments for use with TI microcontrollers and embedded processors. CCS has many features that make it very capable for professional engineers to develop firmware for real products. It comprises a suite of tools (optimizing C/C++ compiler, source code editor, project build environment, debugger, profiler) used to develop and debug embedded applications. Because it can do so much, it can also be a lot to learn for beginners, but don't get discouraged as this module will direct you on how to set up CCS so you can go through exercises smoothly as you build your robotic system.

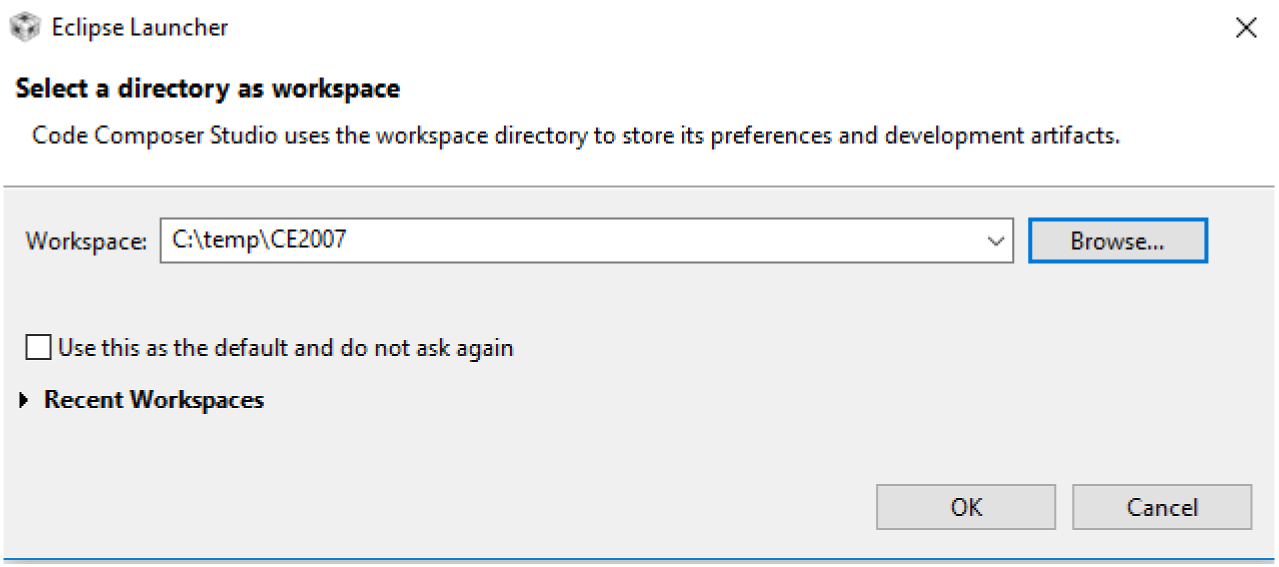
Your code is stored inside of a CCS project. A project can contain many items including your code files, configurations, and other relevant files. The Project Explorer in CCS shows us the various components used for each project. A linker builds a single software system by connecting (linking) software components. In CCS, the build command performs both a compilation and a linking. In an embedded system, the loader will program object code into flash on the microcontroller. We place object code in flash ROM because flash can retain its information if power is removed and restored. In CCS, the Debug command performs a load operation and starts the debugger.

A debugger is a set of hardware and software tools we use to verify system is operating correctly. The two important aspects of a good debugger are control and observability.

6.4 Setting up the CCS environment

Step1: Download the lab project zip file (**CE2007Labs-students.zip**) from NTULearn and copy it into your network drive and unzip. Do not store this into the Local PC because you'll be working and building on this copy of the code for the next 5 labs.

Step 2: Double click the CCS icon on the desktop, click on the browse button and direct CCS to the network folder which you unzip the project files to previously. In the example below, the project files are unzipped to folder C:\temp\CE2007. Click OK to proceed to initialize the selected folder as the workspace.


Eclipse Launcher

Select a directory as workspace

Code Composer Studio uses the workspace directory to store its preferences and development artifacts.

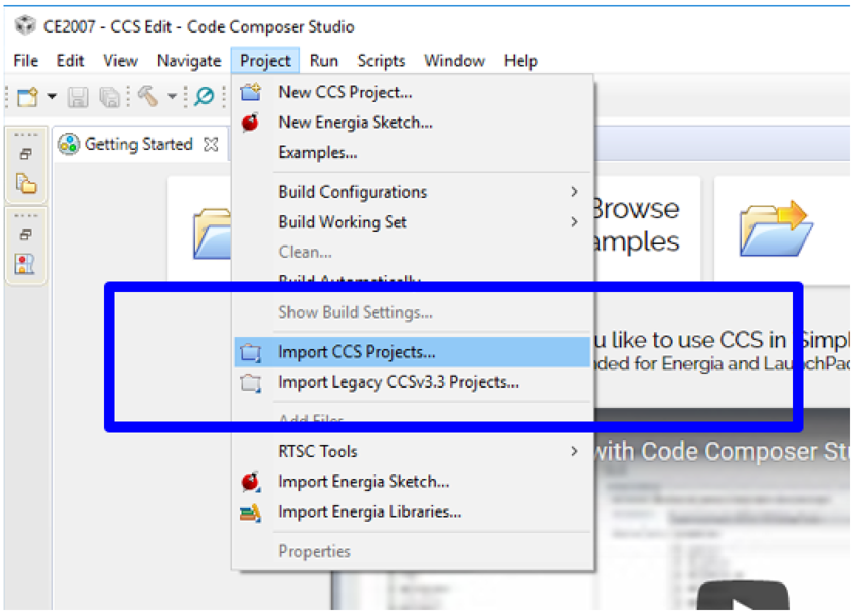
Workspace: C:\temp\CE2007 Browse...

☐ Use this as the default and do not ask again

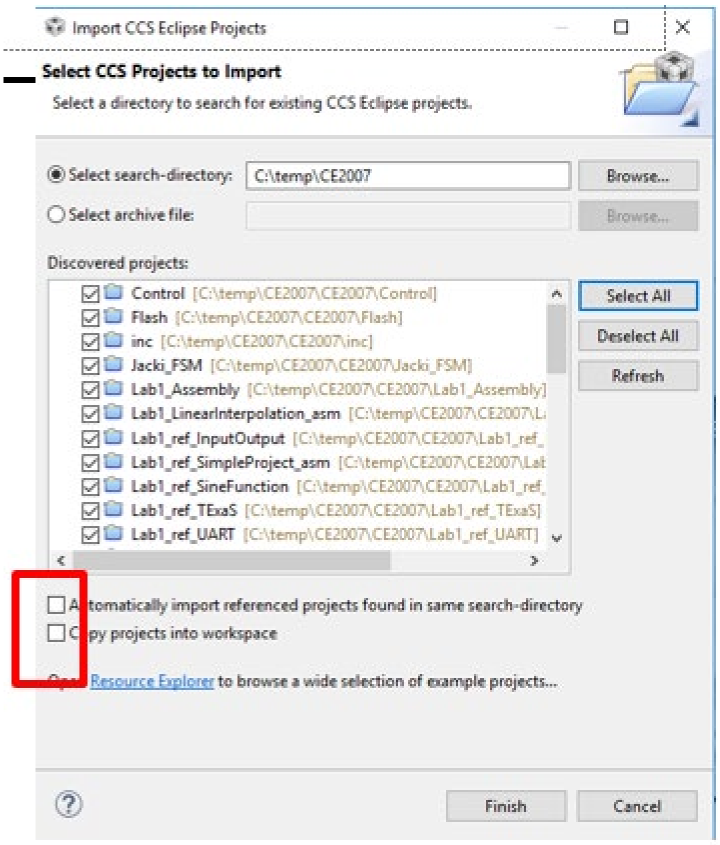
▸ **Recent Workspaces**

OK Cancel

Step 3: Import all projects in the project directory into CCS.


CE2007 - CCS Edit - Code Composer Studio

Pointer to the projects directory and click on Browse. A list of all CCS projects in the project directory will be shown. Click on “Select All”, make sure the two boxes indicated below are unchecked and click on ‘Finish’ button to proceed with the projects import.



Import CCS Eclipse Projects

Select CCS Projects to Import

Select a directory to search for existing CCS Eclipse projects.

(●) Select search-directory: C:\temp\CE2007 [Browse...]

( ) Select archive file: [Browse...]

Discovered projects:

- [✓] Control [C:\temp\CE2007\CE2007\Control]
- [✓] Flash [C:\temp\CE2007\CE2007\Flash]
- [✓] inc [C:\temp\CE2007\CE2007\inc]
- [✓] Jacki_FSM [C:\temp\CE2007\CE2007\Jacki_FSM]
- [✓] Lab1_Assembly [C:\temp\CE2007\CE2007\Lab1_Assembly]
- [✓] Lab1_LinearInterpolation_asm [C:\temp\CE2007\CE2007\L
- [✓] Lab1_ref_InputOutput [C:\temp\CE2007\CE2007\Lab1_ref_
- [✓] Lab1_ref_SimpleProject_asm [C:\temp\CE2007\CE2007\Lat
- [✓] Lab1_ref_SineFunction [C:\temp\CE2007\CE2007\Lab1_ref_
- [✓] Lab1_ref_TExaS [C:\temp\CE2007\CE2007\Lab1_ref_TExaS]
- [✓] Lab1_ref_UART [C:\temp\CE2007\CE2007\Lab1_ref_UART]

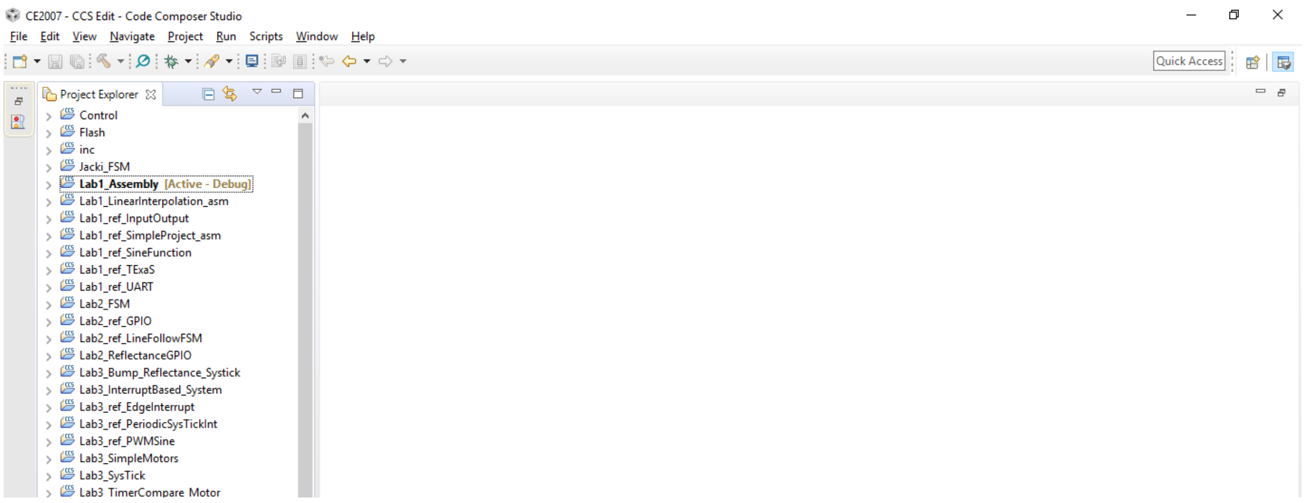
[Select All] [Deselect All] [Refresh]

- [ ] Automatically import referenced projects found in same search-directory
- [ ] Copy projects into workspace

Open Resource Explorer to browse a wide selection of example projects...

[Finish] [Cancel]

Upon successfully import, your CCS window should look similar to that shown below. You are now ready to proceed with the experiments! Remember, do not save your work on the local PC, save them into your network drive.



CE2007 - CCS Edit - Code Composer Studio

- Control
- Flash
- inc
- Jacki_FSM
- **Lab1_Assembly** [Active - Debug]
- Lab1_LinearInterpolation_asm
- Lab1_ref_InputOutput
- Lab1_ref_SimpleProject_asm
- Lab1_ref_SineFunction
- Lab1_ref_TExaS
- Lab1_ref_UART
- Lab2_FSM
- Lab2_ref_GPIO
- Lab2_ref_LineFollowFSM
- Lab2_ReflectanceGPIO
- Lab3_Bump_Reflectance_Systick
- Lab3_InterruptBased_System
- Lab3_ref_EdgeInterrupt
- Lab3_ref_PeriodicSysTickInt
- Lab3_ref_PWMSine
- Lab3_SimpleMotors
- Lab3_SysTick
- Lab3_TimerCompare_Motor

7. EXPERIMENT

7.1 File structure

Following are some naming convention and folder description of the various files you'll be using for lab1-5.

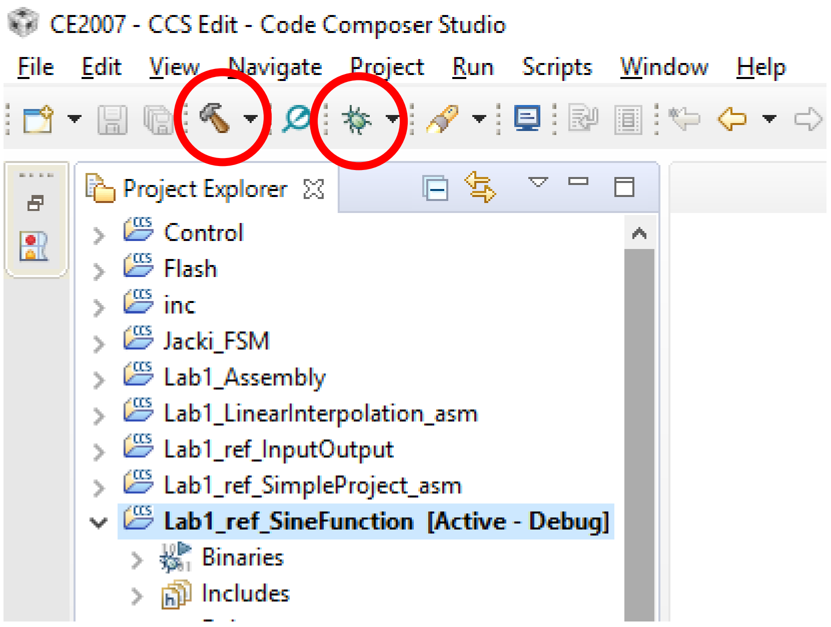
- Project with "LabXref_XXX" are reference projects that gives you examples and hints on how to implement a particular function, configure a particular peripheral etc.
- Project with "LabX_XXX" are CCS projects which contain missing codes that you need to complete.
- The project source files are located in two folders
    - In individual project directory, e.g. Lab1_C_Programming project will have an equivalent "Lab1_ C_Programming" sub-folder in the main directory.
    - In "inc" sub-folder. These are typically peripheral driver files or function files that are shared across multiple CCS projects for different sections or different labs.
    - You will typically find comments such as "// write this for Lab 4" in places which requires you to fill in the blank.
- **In the remaining sections of this lab manual, the paragraphs or statements related to steps that require your action are in blue font to help you identify where you need to use your hands.**

7.2 Introduction to CCS IDE

In this section, you'll edit-compile-link-download a sample project and go through various debug features available in the CCS IDE.

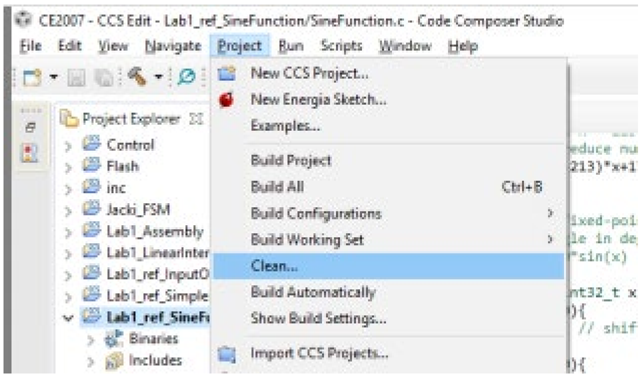
Lab1ref_SineFunction is a very simple software project that does not take/produce input/output. It calculates a y=sin(x) using a cubic approximation and fixed-point math. It fills an array with results. You can use the project to learn how to build (compile), debug (download and start the debugger), run, halt, and observe the array. You should also reset the processor, set a breakpoint, run until the breakpoint, and then single step (step in, step out, and step over).

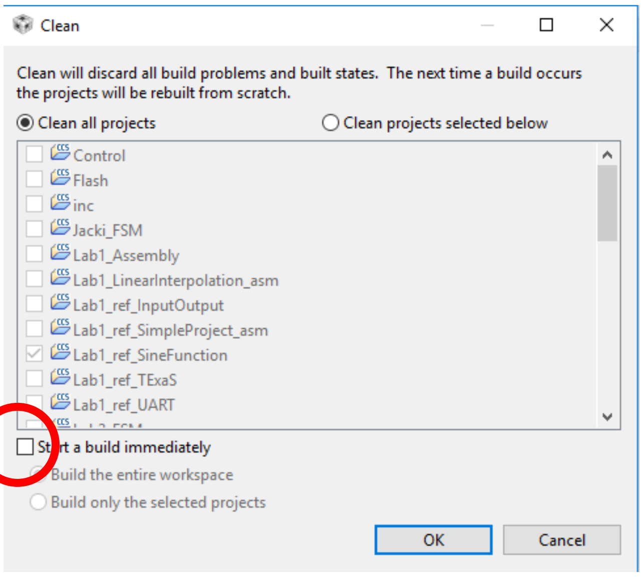
Click on the SineFunction project and open the view of the files in that project. Double click on SineFunction.c to see the source code. Make sure the desired project is in bold text before building or debugging. Notice in the figure below, the Project Explorer bolds the project and specifies "[Active-Debug]"


CE2007 - CCS Edit - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Project Explorer
Control
Flash
inc
Jacki_FSM
Lab1_Assembly
Lab1_LinearInterpolation_asm
Lab1_ref_InputOutput
Lab1_ref_SimpleProject_asm
Lab1_ref_SineFunction [Active - Debug]
Binaries
Includes

Click Build (Hammer icon) to build the project, there shouldn't be any errors. Next, click debug (Bug icon) to download the code to the MSP432 Launchpad. When you debug on the Launchpad for the first time CCS may prompt you to update the firmware. This step is recommended.

Note that the projects in the zip file is created under a different environment so it would be best to do a project clean for all the projects in your workspace before doing any compilation. In the Top menu, click on "Project"->"Clean…", select all projects to clean but you may want to uncheck the "Start a build immediately" else you'll be building every projects in your workspace! The cleaning will take a while as CCS is cleaning all the projects in the workspace.





Clean will discard all build problems and built states. The next time a build occurs the projects will be rebuilt from scratch.

[✓ Clean all projects] Clean projects selected below

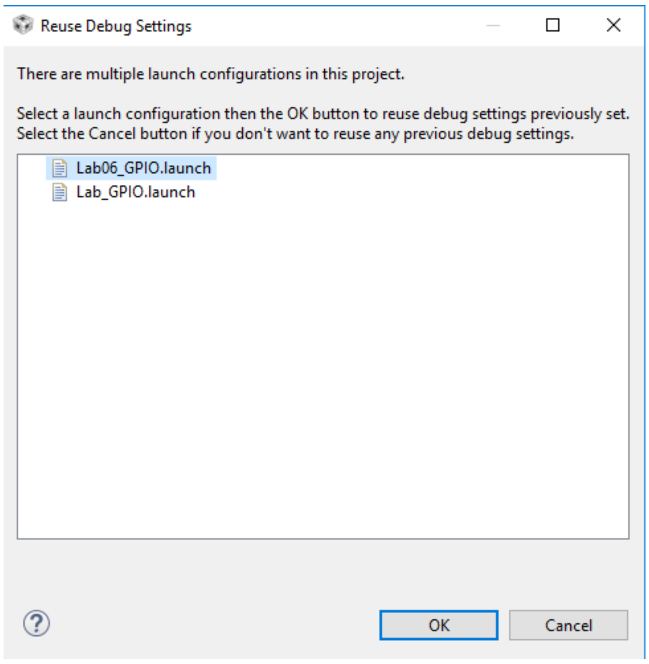
☐ Start a build immediately

Build the entire workspace

Build only the selected projects

The debug operation causes several actions to be done automatically

- Prompt to save source files
- Build the project (incrementally)
- Start the debugger (CCS will switch to the CCS Debug perspective)
- Connect CCS to the target
- Load (flash) the program on the target
- Run to main


Reuse Debug Settings

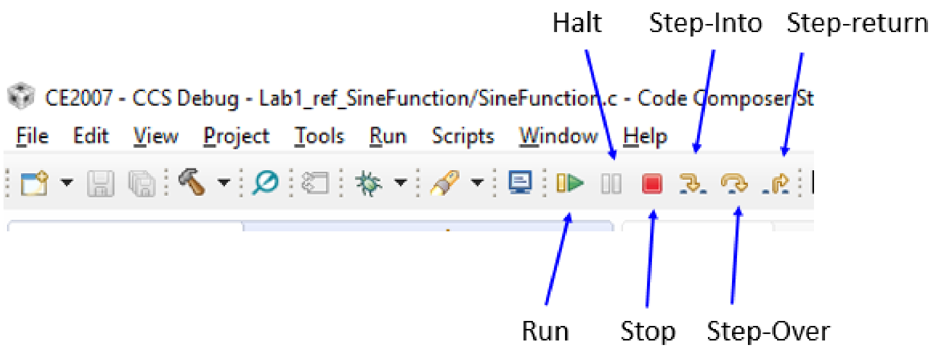
There are multiple launch configurations in this project.

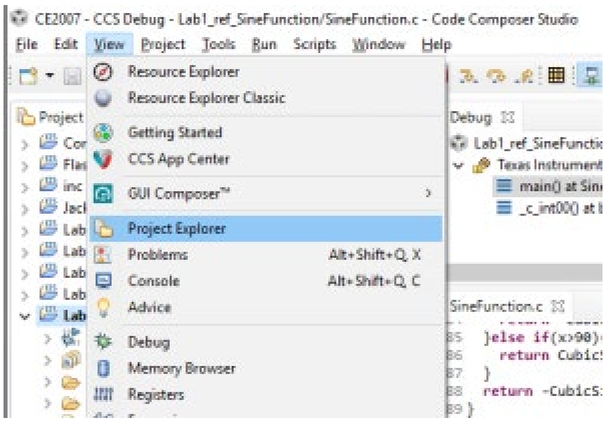
Select a launch configuration then the OK button to reuse debug settings previously set.
Select the Cancel button if you don't want to reuse any previous debug settings.

Lab06_GPIO.launch
Lab_GPIO.launch

If you encounter something similar to the prompt on the left, just click ‘Cancel’ to ignore all previous settings.

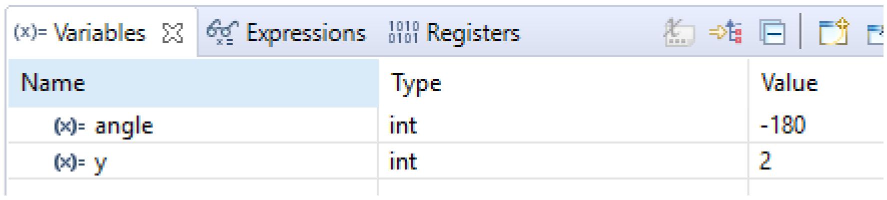
Once the flash is erased, and the image of this project is loaded, you will see the green triangle appear. That is the run icon, don’t click run yet, but seeing this icon means the system is ready to debug. The other key icons and their function are indicated in the figure below. Step over executes one line of C. If that line has a function, step over will execute the entire function. You can also experiment with Step in (which executes one line of C, and if that line has a function, it will step into that function. If you have stepped into a function, Step return will complete that function and stop at the spot the function was called. Ask around if you are not sure what these functions do.


Halt
Step-Into
Step-return
CE2007 - CCS Debug - Lab1_ref_SineFunction/SineFunction.c - Code Composer St
File Edit View Project Tools Run Scripts Window Help
Run
Stop
Step-Over

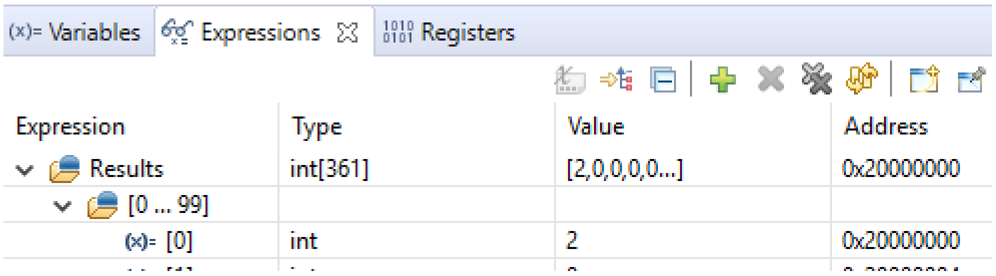

CE2007 - CCS Debug - Lab1_ref_SineFunction/SineFunction.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Resource Explorer
Resource Explorer Classic
Getting Started
CCS App Center
GUI Composer™
Project Explorer
Problems Alt+Shift+Q, X
Console Alt+Shift+Q, C
Advice
Debug
Memory Browser
Registers

If for some reason, the project explore window is not visible, you can enable it via View->Project Explore.

Single step the program by executing **Step Over** icon multiple times. Observe the local variables in the **Variables** window. Observe global variables in the Expressions window. Type **Results** in the "Add new expression" field and hit <enter>. Expand the Results field to see its data. The display format of the expression **Value** can be configured via right click. Examples of display format are decimal, hex, binary etc.



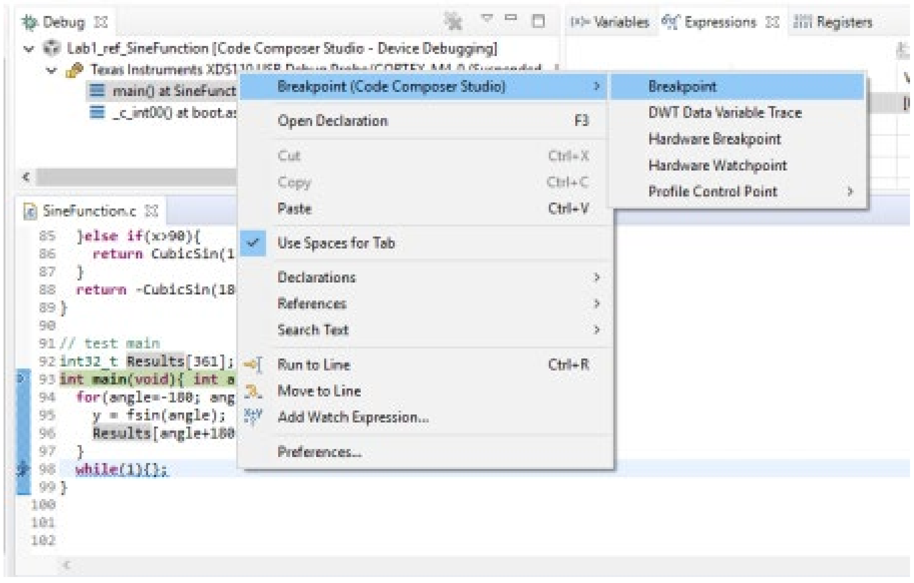
Name	Type	Value
(x)= angle	int	-180
(x)= y	int	2



Expression	Type	Value	Address
Results	int[361]	[2,0,0,0,0...]	0x20000000
[0 ... 99]			
(x)= [0]	int	2	0x20000000

Set a breakpoint at the line 98 "While(1) {}" statement. You can set a breakpoint by right clicking on the while statement and select "Breakpoint". A blue ball will appear at the blue column at the leftmost edge of the window (See figure below). Double clicking on the left blue column will toggle the breakpoint, check it out.

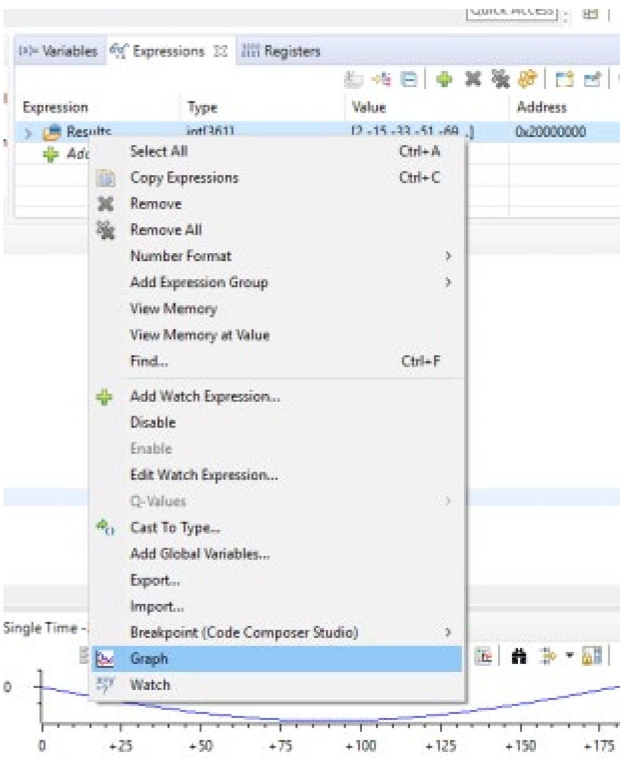
Hit the Run button, program will run till it hit the breakpoint at line 98, after filling in the result array. The status of the expressions/variables in the debug window will be refreshed every time a breakpoint is hit. You can verify that by setting a breakpoint at line 96. One element of the Results[] array will be updated at every breakpoint hit.



```
  }else if(x>90){
    return CubicSin(1
  }
  return -CubicSin(18
}

// test main
int32_t Results[361];
int main(void){ int a
  for(angle=-180; ang
    y = fsin(angle);
    Results[angle+180
  }
  while(1){};
}
```

Result array now contains values that will form a sine wave. View the graphical visualization by right clicking on Result in the expression window and select Graph at the bottom of the pop-up menu.



Check out the other execution options under the Run menu.

Halt the debugger after exploring by pressing on the STOP button. CCS will return to the editor view.

7.3 Interacting with hardware

In this section, you'll use the register view feature to correlate the behavior of the GPIO hardware with their status and control register.

Lab1ref_InputOutput is a simple project that showcases some the features of the LaunchPad. For example, it will input from the two switches on the LaunchPad and output to the LED. Compile and load this project onto MSP432 LaunchPad.

Run the project and interact with the two switches on the LaunchPad. You should observe this simple behavior:

- No switches No LEDs on
- Just SW1 Red LED is on, color LED is blue
- Just SW2 Red LED is on, color LED is red
- Both SW1,SW2 Red LED is on, color LED is blue+red=purple

Figure 3 below is the connection diagram of the GPIO to the switches and LEDs.

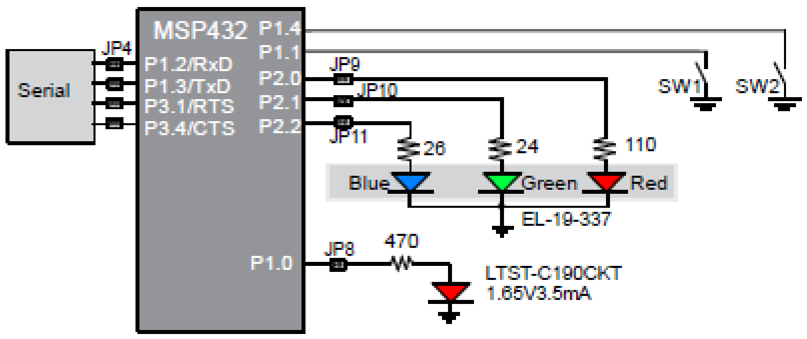
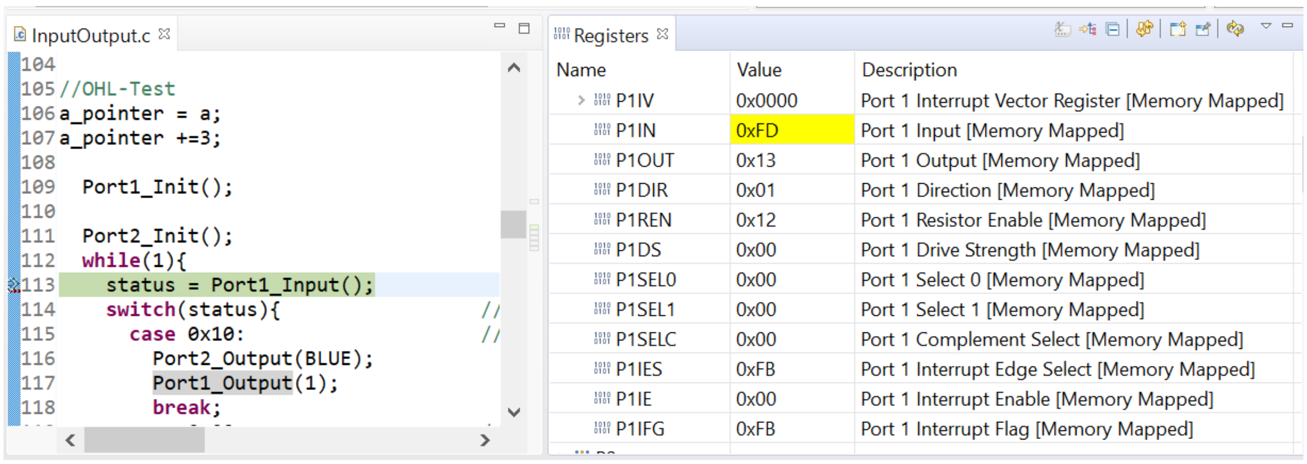


Figure 3: MSP432 Launchpad LED and SW connections

Set a breakpoint at the line status = Port1_Input() and click on the Resume/Run button, the program will run to the breakpoint and stop. Observe the I/O Port registers. First select the Registers tab, then select P1 (I/O Port 1). Assert and hold the switches on the Launchpad and Click on Resume/Run. You will see the P1IN value changes (See figure in the next page)

What happened here is that you have previously stopped just before executing Port1_Input(). Clicking on the Resume button causes the program to start executing, so Port1_Input() will read status of the 8 pins from GPIO Port1 and if P1.1 and/or P1,4 is asserted (logic '0' for this case), the corresponding LED will be lighted. Program will loop back to Port1_Input() again to sample the Input, but since a breakpoint is set on this line, the program will halt until user issue another command. As mentioned earlier, status of the various windows in the IDE will be updated when a breakpoint is hit.



```
//OHL-Test
a_pointer = a;
a_pointer +=3;

  Port1_Init();

  Port2_Init();
  while(1){
    status = Port1_Input();
    switch(status){                //
      case 0x10:                   //
        Port2_Output(BLUE);
        Port1_Output(1);
        break;
```

Name	Value	Description
P1IV	0x0000	Port 1 Interrupt Vector Register [Memory Mapped]
P1IN	0xFD	Port 1 Input [Memory Mapped]
P1OUT	0x13	Port 1 Output [Memory Mapped]
P1DIR	0x01	Port 1 Direction [Memory Mapped]
P1REN	0x12	Port 1 Resistor Enable [Memory Mapped]
P1DS	0x00	Port 1 Drive Strength [Memory Mapped]
P1SEL0	0x00	Port 1 Select 0 [Memory Mapped]
P1SEL1	0x00	Port 1 Select 1 [Memory Mapped]
P1SELC	0x00	Port 1 Complement Select [Memory Mapped]
P1IES	0xFB	Port 1 Interrupt Edge Select [Memory Mapped]
P1IE	0x00	Port 1 Interrupt Enable [Memory Mapped]
P1IFG	0xFB	Port 1 Interrupt Flag [Memory Mapped]

Figure above shows the status when SW1 is asserted. SW1 is connected to P1.1, whose status can be read from Bit1 of P1IN register. 0xFD in binary is 1111 1101. Bit1 is logic ‘0’. You can also explore P2OUT register to see the status of the P2.0, P2.1 and P2.2 pins. E.g. if SW1 is asserted, BLUE LED is lighted and P2OUT should have a value of 0x4. 0x4 in binary is 0000 0100. Bit2 is logic ‘1’ so P2.2 outputs a logic ‘1’ to light up the BLUE LED. We will investigate details of GPIO operation in subsequent lectures.

7.4 C Language Programming

You will be introduced to some basic but important techniques on C programming in this section, techniques that will be useful in your subsequent labs. We will use the same Lab1ref_InputOutput project you use in section 7.3.

Bit Manipulation

Bit manipulation is a common technique used in C program, especially embedded system where you typically need to deal with control and status registers, variables or flags where each bit represent a specific function or status.

Load the project Lab1_C_Programming, put a break point at Bit_Manipulation() and run the program. Step Into Bit_Manipulation() function. Read the comments associated with each C statements in the function before single stepping through the code.

In general, there are three bitwise logical operations that you can use to perform bit manipulation in C.

- ‘|’. OR operation is used to set a particular bit in the target destination without affecting other bits in the destination.
- ‘&’. AND operation is used to clear a particular bit in the target destination without affecting other bits in the destination.
- ‘^’. XOR operation is used to toggle a particular bit in the target destination without affecting other bits in the destination.

- '~'. Bitwise inversion. Used to invert each bit in the bit-mask. Typically used together with '&' allow the same bit mask to be used for all three bitwise logical operation.
- Examples:
    - X |= 0x2; // Setting bit1 of variable X. Bit-mask = 0x2 (0010 in binary).
    - Y &= ~0x2; // Clearing bit1 of variable Y.
    - Z ^= 0x3; // Toggle bit0 and bit1 of variable Z.

Bit Shifting and Extraction

Bit shifting and extraction is often used to extract a particular bit or a few targeted bits from a variable for processing or data reformatting purpose. E.g. when you want to test the status of say bit2 of a variable, or when you want to pack bit0, bit2, bit4, bit6 of the variable together, i.e. 0101 0101 becomes 0000 1111.

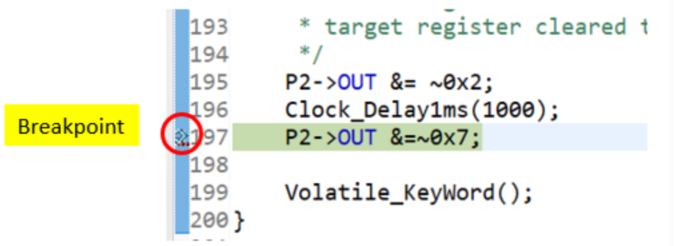
Using the same project Lab1_C_Programming, put a break point at Bit_ShiftExtract() and run the program. Step Into Bit_ShiftExtract() function. Read the comments associated with each C statements in the function before single stepping through the code.

The function Bit_ShiftExtract() extract bit2 of each element in the test[] array and increment the count variable if the bit2='1'.

Volatile Key Word

Volatile key word is used to inform the compiler that the variable may change outside the context of the program. One effect is that compiler will assign a memory location for the variable. Without the volatile key word, the compiler may assign the variable to a temporary register or even remove the variable during the code optimization process.

Reload the program, put a breakpoint at (P2->OUT &= ~0x7) in Bit_Manipulation(), and Run the program.



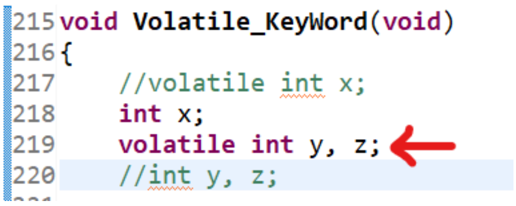
```
     * target register cleared t
     */
    P2->OUT &= ~0x2;
    Clock_Delay1ms(1000);
    P2->OUT &=~0x7;

    Volatile_KeyWord();
}
```

When the breakpoint is hit, check the Disassembly Window, can you find the call to Volatile_KeyWord()?

What happened was that the "main.c" file in this project is compiled with code optimizer enabled. When code optimization is enabled, the C compiler will perform modifications to the user code at assembly level to produce a program that is more efficient in execution cycle or code size. Note that the rest of the codes in all projects for Lab1 to Lab5 are compiled with code optimizer OFF. Even at the lowest optimization level, the optimizer assess that the code in Volatile_KeyWord() is not doing anything useful so it removed the entire function!

Next, insert a 'volatile' qualifier to variable y and z in the Volatile_KeyWord() and re-build the project, load and run the program.

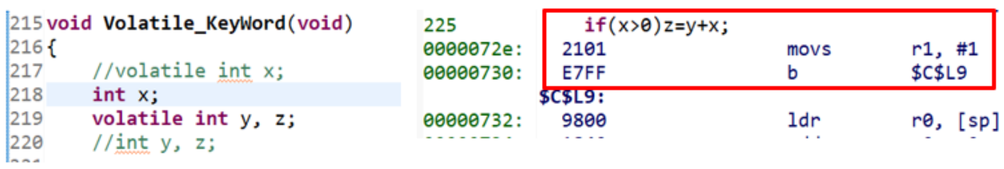


```
void Volatile_KeyWord(void)
{
    //volatile int x;
    int x;
    volatile int y, z;
    //int y, z;
```

Check whether the function is visible now. The volatile qualifier informed the optimizer that variables y and z may change outside the context of the program and all code/functions involving variable y and z should not be removed.

Next, look at the assembly implementation for the C statement "if(x>0)z=y+x;". Compare it with the assembly implementation if variable x is declared with a volatile qualifier. You can see that if x is not a volatile variable, the compare will not happen because to the code optimizer, x value did not change after being initialize to 1 in the code, hence testing is unnecessary. But if x is a volatile variable, comparison must be done as its value may change outside the program context.

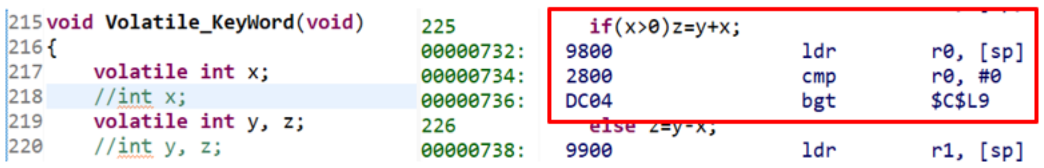
"int x;"



```
void Volatile_KeyWord(void)
{
    //volatile int x;
    int x;
    volatile int y, z;
    //int y, z;
```

```
225            if(x>0)z=y+x;
0000072e:    2101          movs      r1, #1
00000730:    E7FF          b         $C$L9
          $C$L9:
00000732:    9800          ldr       r0, [sp]
```

"volatile int x;"



```
void Volatile_KeyWord(void)
{
    volatile int x;
    //int x;
    volatile int y, z;
    //int y, z;
```

```
225            if(x>0)z=y+x;
00000732:    9800          ldr       r0, [sp]
00000734:    2800          cmp       r0, #0
00000736:    DC04          bgt       $C$L9
226            else z=y-x;
00000738:    9900          ldr       r1, [sp]
```

7.5 Assembler

There are two reasons for learning the assembly language of the computer that we are using. Sometimes, but not often, we wish to optimize our application for maximum execution speed or minimum memory size, and thus writing pieces of our code in assembly language is one approach to such optimizations. The most important reason, however, is that by observing the assembly code generated by the compiler for our C code we can truly understand what our software is doing, which is what you did in section 7.4 by observing the behaviour of the optimizer when volatile keyword is used. Based on this understanding, we can evaluate, debug, and optimize our system. So, the goal of this module is not for you to become proficient in assembly language, but rather to learn enough so you can interpret the assembly code generated by the C compiler.

An assembler is system software that converts low-level assembly language program (human readable format) into object code (machine readable format). Typically, one line of assembly language creates one machine instruction. Writing in assembly exposes the low-level details of the architecture.

The following task requires you to implement an assembly routine call Convert. The code of the main routine that calls the convert routine is given in the Lab1_Assembly Project. You can reference the code in Lab1ref_SimpleProject_asm project to see how the '.field' directive is used to load a large constant. Remember that ARM assembly instruction has certain limitation in loading large immediate constants.

#Task:

- Click on Lab1_Assembly Project and complete the missing lines in convert.asm.
- The sub-routine 'Convert' which take in the ADC output value (n) and convert it to distance (D) in mm via an equation D = 1195172/(n – 1058).
- The conversion routine saturates the output to 800mm as this is the maximum measurement distance of the IR Sensor. Note that there is a division in the conversion equation, so a SDIV instruction is required.

You can use the main program delivered as part of the Lab1_Assembly project to test your Convert function. This testing approach is Black Box functional testing. This testing methodology just sets inputs and observes outputs. In other words, we look at the outside of the software, and not probe any of the internal details of the function. Black box testing looks at the overall functionality of what software does without know of how it works. This test program contains 16 test cases (inputs and expected outputs). The expected results are plotted as in Figure 4 below.

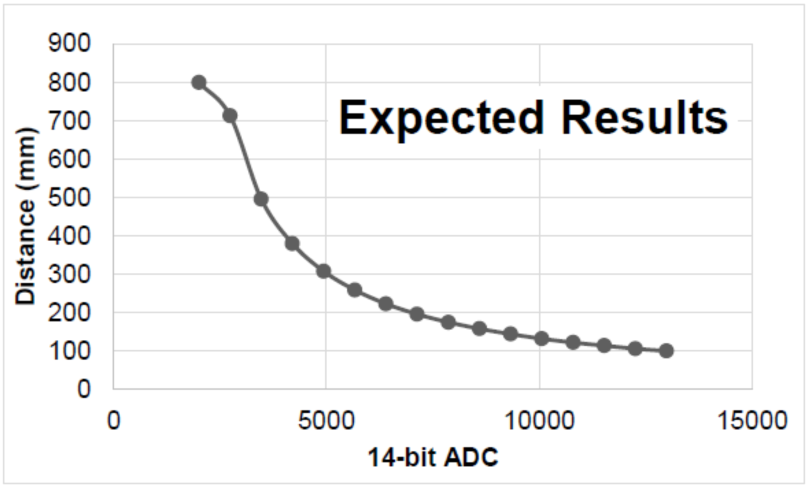
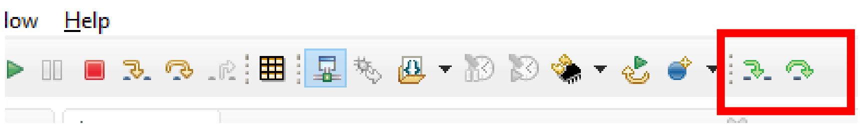


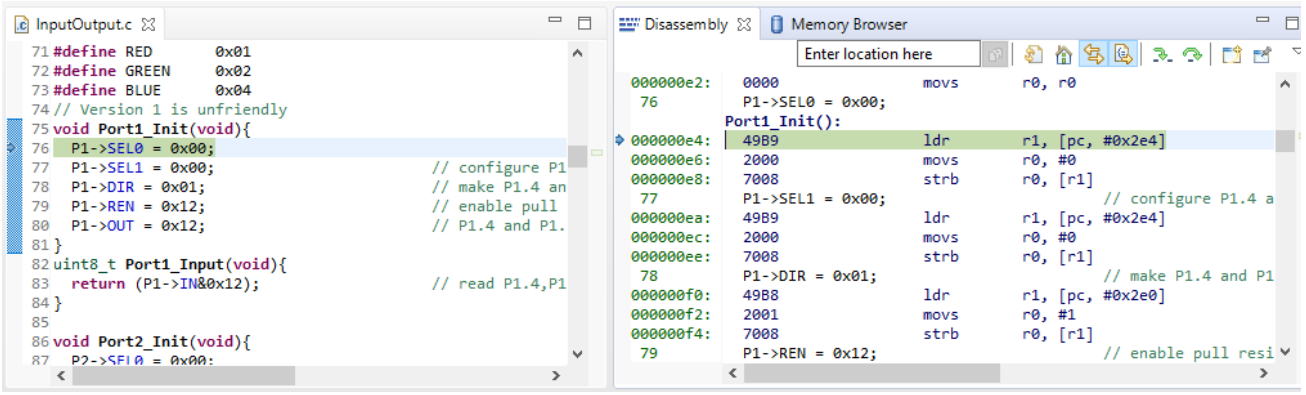
Figure 4: Distance vs ADC values

Run main and compare your results with expected values. Note that the graph in Figure 4 is only for illustration purpose, do not 'compare' your results using the graph. The code used to perform the validation of your result is in the Lab1_Assemblymain.asm, check the code in the file to find out which register(s) you need to observe to see if your result is correct or not. It is ok if your results differ by ±1 (which could be due to rounding). Note that you should use the green arrows for single stepping in Assembly. The yellow arrows are for single stepping in C program.



Dissecting C statements

In this section, you will look other aspects of how C statements are implemented in ASM. Open the Lab1ref_InputOutput project and Click on View->Disassembly to bring up the Disassembly Window. Step through the program to see how each C statement is implemented in ASM. Step into Port1_Init() and you'll see something similar to the figure below. Take note of how the various Port1 registers are initialized in ASM.



```
71 #define RED        0x01
72 #define GREEN      0x02
73 #define BLUE       0x04
74 // Version 1 is unfriendly
75 void Port1_Init(void){
76   P1->SEL0 = 0x00;
77   P1->SEL1 = 0x00;                          // configure P1
78   P1->DIR = 0x01;                           // make P1.4 an
79   P1->REN = 0x12;                           // enable pull
80   P1->OUT = 0x12;                           // P1.4 and P1.
81 }
82 uint8_t Port1_Input(void){
83   return (P1->IN&0x12);                     // read P1.4,P1
84 }
85
86 void Port2_Init(void){
87   P2->SEL0 = 0x00;
```

```
000000e2:   0000              movs      r0, r0
 76         P1->SEL0 = 0x00;
          Port1_Init():
000000e4:   49B9              ldr       r1, [pc, #0x2e4]
000000e6:   2000              movs      r0, #0
000000e8:   7008              strb      r0, [r1]
 77         P1->SEL1 = 0x00;                      // configure P1.4 a
000000ea:   49B9              ldr       r1, [pc, #0x2e4]
000000ec:   2000              movs      r0, #0
000000ee:   7008              strb      r0, [r1]
 78         P1->DIR = 0x01;                       // make P1.4 and P1
000000f0:   49B8              ldr       r1, [pc, #0x2e0]
000000f2:   2001              movs      r0, #1
000000f4:   7008              strb      r0, [r1]
 79         P1->REN = 0x12;                       // enable pull resi
```

Questions:

- What does “ldr r1, [pc, #0x2e4]” do? Note that actual offset value (#0x2e4) may be different on your version of the code, but essentially, both SEL0 and SEL1 initialization will have the same offset value.
- Why is the same instruction used in the initialization of the Port1’s SEL0 and SEL1 registers? Does that mean these instructions are all writing to the same location?
- Go to the exact memory location to see what content is stored there. You can use View->Memory Browser.
- Which register stores the return value for the function Port1_Input()? Write down the ASM instruction involved.
- Which register contains the input argument of the function Port2_Output()? Write down the ASM instruction involved.

When exploring how data are exchanged between calling and called routines in C, you need to look at the programming standard used. Note that the sub-routines examples given adhere to a programming standard, called ARM Architecture Procedure Call Standard (AAPCS). There are many components of this standard, but the ones relevant to this lab include:

- If there is one input parameter, it is passed in R0
- If there are two input parameters, they are passed in R0, R1
- If there are three input parameters, they are passed in R0-R2
- If there are four input parameters, they are passed in R0-R3
- If there is an output parameter, it is returned in R0
- The function can modify R0-R3, R12 freely
- If a function wishes to use R4-R11 then it must save and restore them using the stack.
- If a function calls another function, then it must save and restore LR
- Functions must balance the stack

7.6 Linker

A linker builds a single software system by connecting (linking) software components. In CCS, the build command performs both assembly and linking. In an embedded system, the loader will program object code into on-chip flash. We place object code in non-volatile memory such as Flash because it can retain information even if power is removed. In CCS, the Debug command performs a load operation and starts the debugger.

The linker assigned absolute addresses to various software sections according to the configuration listed in the linker command file. In the process, a map file is generated which contains many useful information. In this section, you’ll explore the map file to check out what information is available. The map file can be found under the Debug folder of the Lab1ref_InputOutput project directory. Filename Lab1ref_InputOutput.map.

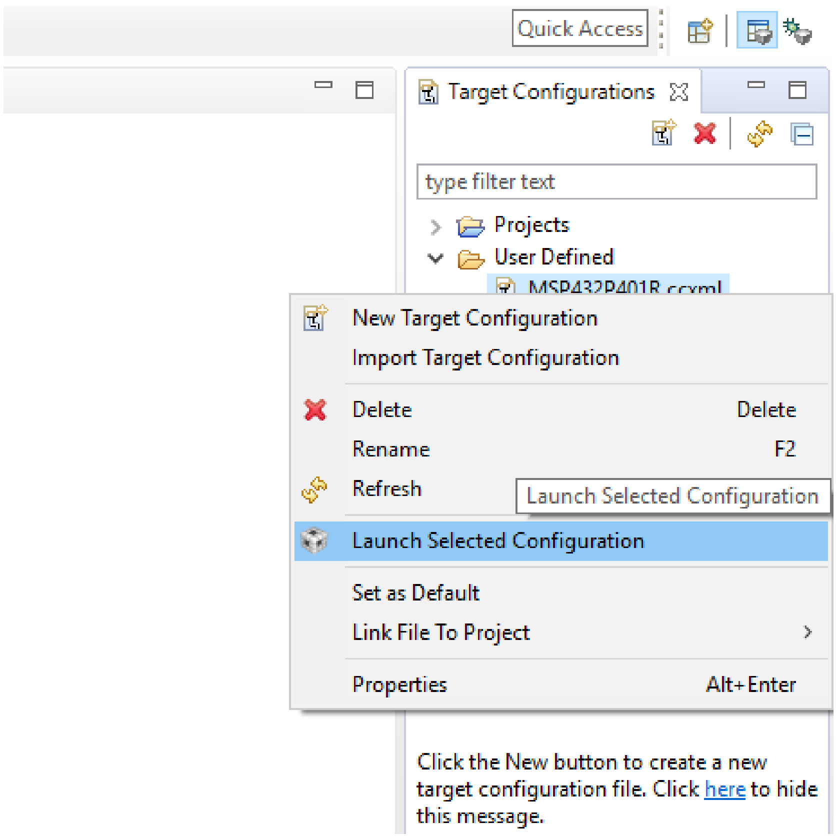
Some questions to ponder when going through the map file:

- How large is the code size for this project?
- Which file consume the largest code size for this project?
- How much SRAM memory is left for program/data expansion?
- What is the starting address of Port1_Init()? Compare this with the address you see in the Disassembly Window, are they the same? If not, why? HINT: Its related to the way ARM processor handle the processor state (ARM, Thumb).
- Locate the global symbols that can be found in the map file.
- Explore how compiler allocates software sections (.bss, .data etc) for different variables by declaring different type of variables in the program and check the map file on their allocation. E.g. global arrays (initialized and uninitialized), static arrays etc. Large arrays can be used so it is easier for you to identify changes in the map file. Stack allocation will be more difficult to observe.

7.7 RSLK-MAX Self-Test (good to re-visit when developing application for Lab5).

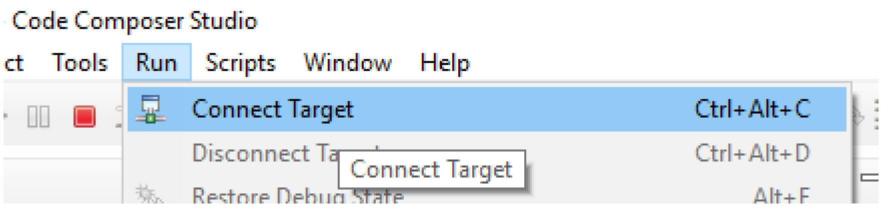
You can test out the RSLK-MAX with the RSLK-MAX-SelfTest.out located in the root folder of the zip file. Since you only have a .out file with no associated project source, you will need to start a Project-Less Debug Session to download the .out file into the hardware.

Click View->Target Configurations. Right Click on MSP432P401R.ccxml and select "Launch Selected Configuration"

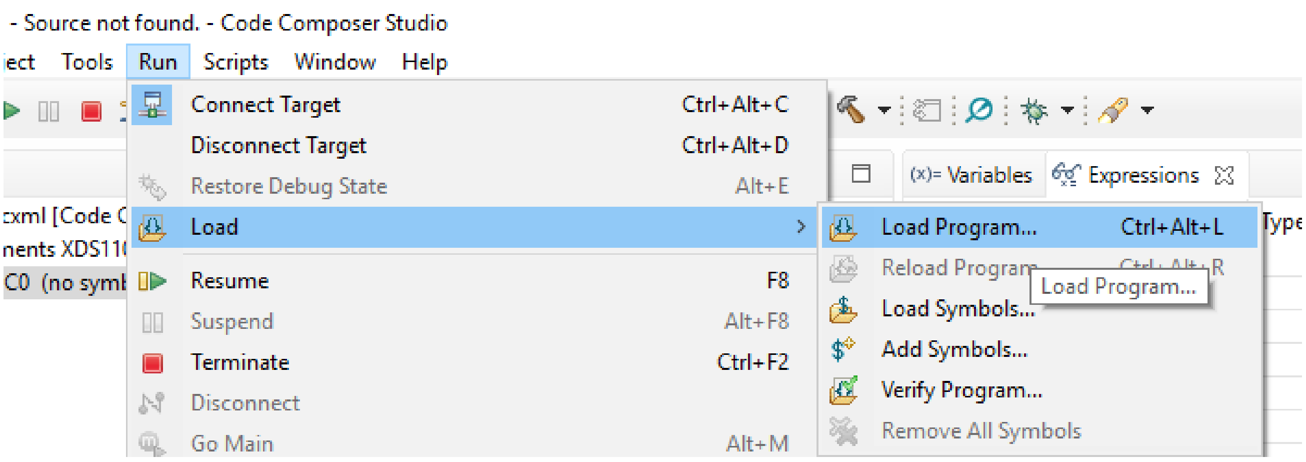


Click the New button to create a new target configuration file. Click here to hide this message.

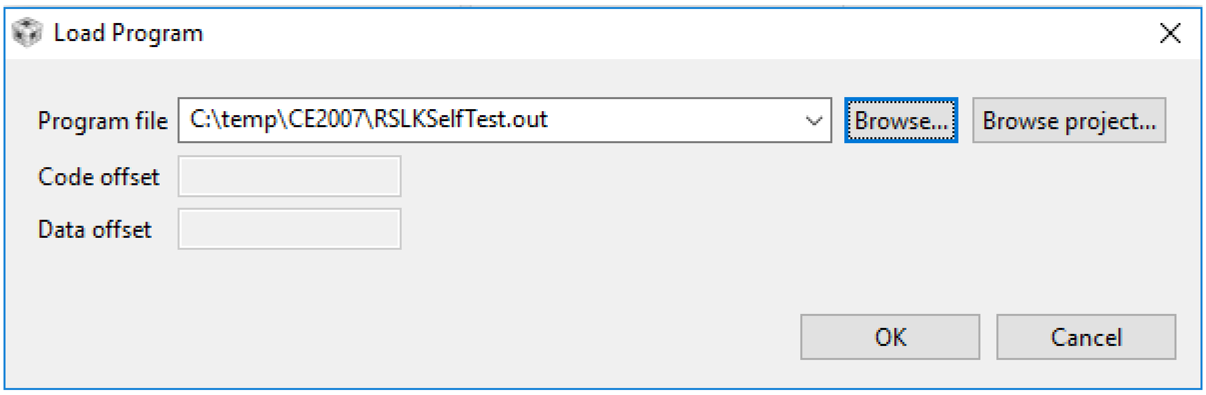
Click on Run-> Connect Target


Code Composer Studio
ct Tools Run Scripts Window Help
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
Connect Target
Restore Debug State Alt+F

Click on Run->Load-> Load Program …

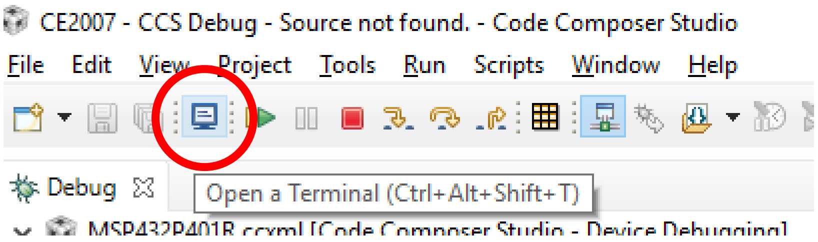

- Source not found. - Code Composer Studio
ect Tools Run Scripts Window Help
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
Restore Debug State Alt+E
Load
Resume F8
Suspend Alt+F8
Terminate Ctrl+F2
Disconnect
Go Main Alt+M
Load Program... Ctrl+Alt+L
Reload Program Ctrl+Alt+R
Load Program...
Load Symbols...
Add Symbols...
Verify Program...
Remove All Symbols
(x)= Variables
Expressions

Load the RSLKSelfTest.out from the root project directory

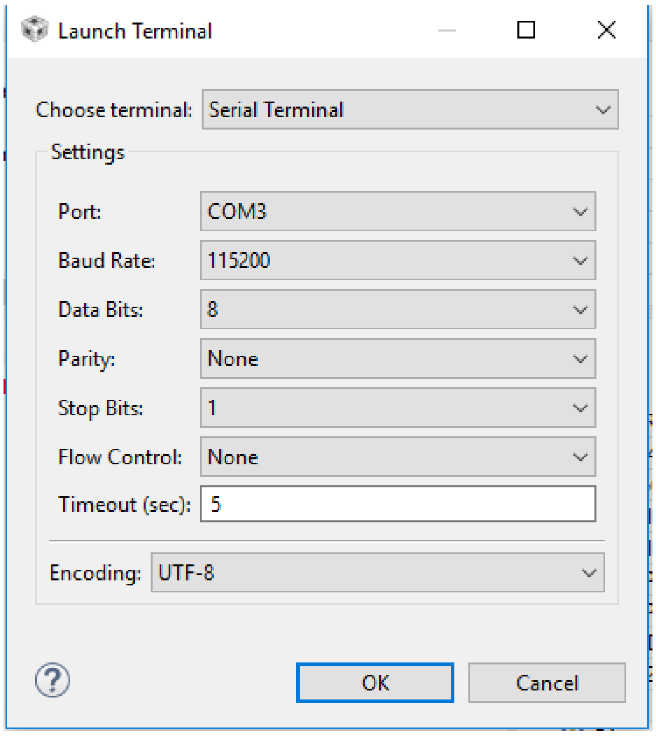


Program file C:\temp\CE2007\RSLKSelfTest.out
Code offset
Data offset

Open a Terminal Window in CCS


CE2007 - CCS Debug - Source not found. - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
Open a Terminal (Ctrl+Alt+Shift+T)
MSP432P401R.ccxml [Code Composer Studio - Device Debugging]

Configure the UART connection. The example below uses COM3, it may differ on your PC.

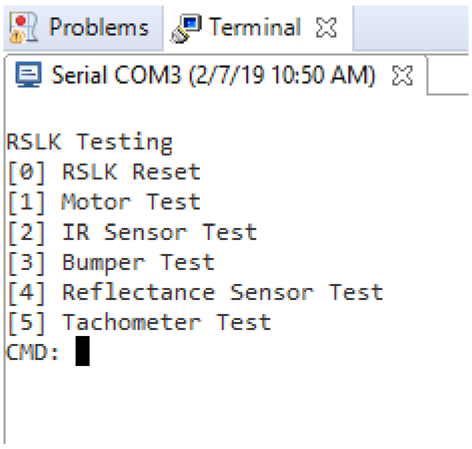


Choose terminal: Serial Terminal

Settings

Port: COM3
Baud Rate: 115200
Data Bits: 8
Parity: None
Stop Bits: 1
Flow Control: None
Timeout (sec): 5
Encoding: UTF-8

After bringing up the Terminal Window, run the RSLKSelfTest Program and you are good to go to do the testing.



```
RSLK Testing
[0] RSLK Reset
[1] Motor Test
[2] IR Sensor Test
[3] Bumper Test
[4] Reflectance Sensor Test
[5] Tachometer Test
CMD:
```

Go through all the tests. You can use this .out file to check the hardware in future, if you suspect the issue you are facing is due to faulty hardware.

8. Difference between RSLK and RSLK-MAX (to be added if RSLK-MAX is used)

RSLK	RSLK-MAX	Remarks
One IR LED controlled by P5.3	Two IR LEDs controlled by P5.3 and P9.2. Need to set both GPIO pins to HIGH to enable IR LED for the Reflectance Read Process.	Lab2 Reflectance Sensor Interface Reflectance.c #define RSLK_MAX 1
P1.7 (Motor DIR L) P1.6 (Motor DIR R)	P5.4 (Motor DIR L) P5.5 (Motor DIR R)	Lab3 Motor. Lab4 Tachometer. Lab5. Motor.c TA3InputCapture.c #define RSLK_MAX 1
P8.2. Timer Capture Input connecting to Left Motor Tachometer. Timer A3 Ch2.	P10.5. Timer A3 Ch1.	Lab4 Tachometer. Lab5. TA3InputCapture.c #define RSLK_MAX 1

9. ASSIGNMENT

- Complete the Lab1 handout and submit before your next lab.