Experiment 1

Laboratory Hardware and Tools

Each day, our lives become more dependent on 'embedded systems', digital information technology that is embedded in our environment. Try making a list and counting how many devices with embedded systems you use in a typical day. Here are some examples: if your clock radio goes off, and you hit the snooze button a few times in the morning, the first thing you do in your day is interact with an embedded system. Heating up some food in the microwave oven and making a call on a cell phone also involve embedded systems. That is just the beginning. Here are a few more examples: turning on the television with a hand held remote, playing a handheld game, using a calculator, and checking your digital wristwatch. All those are embedded systems devices that you interact with.

Exponentially increasing computing power, ubiquitous connectivity and the convergence of technology have resulted in hardware/software systems being embedded within everyday products and places. The last few years has seen a renaissance of hobbyists and inventors building custom electronic devices. These systems utilize off-the-shelf components and modules whose development has been fueled by a technological explosion of integrated sensors and actuators that incorporate much of the analog electronics which previously presented a barrier to system development by non-engineers. Microcontrollers with custom firmware provide the glue to bind sophisticated off-the-shelf modules into complex custom systems.

What are Embedded Systems?

Embedded systems are combination of hardware and software combined together to perform a dedicated task. Usually, they are used to control a device, a process or a larger system. Some examples of embedded systems include those controlling the structural units of a car, the automatic pilot and avionics of aircraft, telematic systems for traffic control, the chipset and software within a set-top box for digital TV, a pacemaker, chips within telecommunication switching equipment, ambient devices, and control systems embedded in nuclear reactors. The block diagram of embedded system is shown in Figure 1.1

Lab Objective

Development of an embedded system requires that combination of both hardware and software components should perform their assigned tasks under the predefined circumstances. This lab provides a series of experiments aimed at teaching hardware interfacing and embedded programming skills. We follow the bottom up approach by starting with simpler tasks and

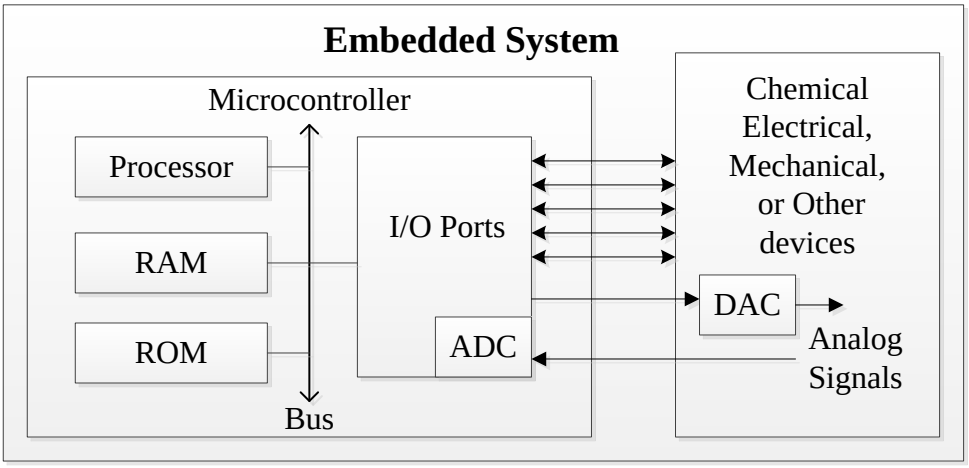


Figure 1.1: Block diagram of an embedded system

gradually building on that to develop a complete embedded system.

Prerequisites for Lab

This lab is designed for the students having some experience in 'C' programming, but no prior experience with embedded systems. In this lab, we assume that you have basic understanding of digital logic design and analog electronics.

Hardware Required

Hardware required for the experiments in this lab is listed below:

1. EK-TM4C123GXL - ARM Cortex-M4F Based microcontroller TM4C123G Tiva-C LaunchPad (Previously known as Stellaris Launchpad Board based on LM4F120H5QR microcontroller)
2. Expansion Board based on different electronic components required to perform lab assignments.

Tiva C Series TM4C123G LaunchPad

The key component used in the tutorials is the Tiva C Series TM4C123G (Stellaris) Launchpad board produced by Texas Instruments (TI). The board, illustrated in Figure 1.2, includes a user configurable TM4C123GH6PM micro-controller with 256 KB flash and 32 KB RAM as well as integrated circuit debug interface (ICDI). With appropriate software running on the host it is possible to connect to the TM4C123 (LM4F120) processor to download, execute and debug user code.

In Figure 1.2, there is a horizontal white line slightly above the the midpoint. Below the line are the TM4C123GH6PM, crystal oscillators, user accessible RGB LED, user accessible push-buttons and a reset push button. Above the line is the hardware debugger interface including

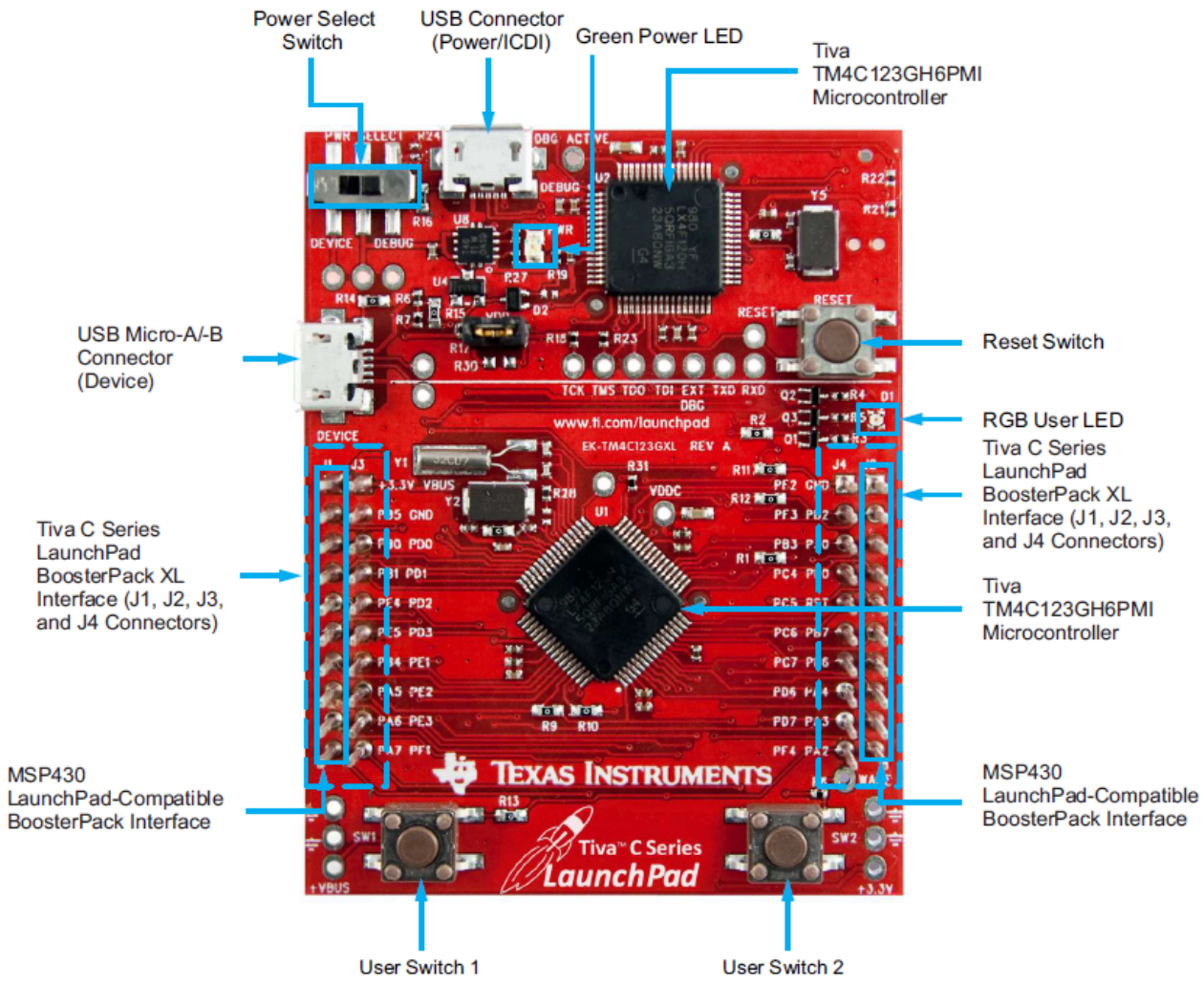


Figure 1.2: Launchpad Board

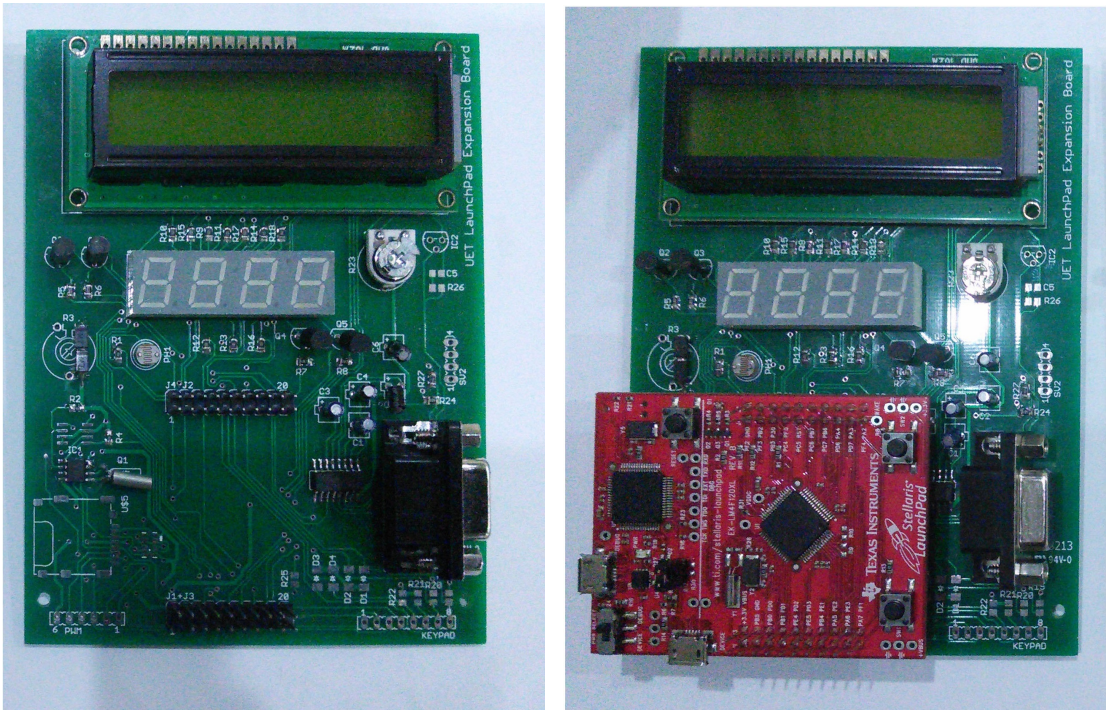
a 3.3V voltage regulator and other components. The regulator converts the 5V supplied by the USB connection to 3.3V for the processors and also available at the board edge connectors.

All the pins of Tiva C (Stellaris) Launchpad are brought out to well labeled headers – the pin labels directly correspond to the logical names used throughout the documentation rather than the physical pins associated with the particular part/package used. This use of logical names is consistent across the family and greatly simplifies the task of designing portable software.

The TM4C123GH6PM (LM4F120H5QR) is a member of the TIVA-C series (Stellaris) processors and offers 80 MHz Cortex-M4 processor with FPU, a variety of integrated memories and multiple programmable GPIO. This board provides far more computation and I/O horsepower than is required for the tasks performed in the lab. Furthermore, the TM4C123GH6PM (LM4F120H5QR) microcontroller is code-compatible to all members of the extensive Tiva C (Stellaris) family, providing flexibility to fit precise needs.

Expansion Board

The headers on the Launchpad can be used to connect the external peripherals and electronic devices to develop a custom application. The expansion board used in the lab for the launchpad to explore different applications that our MCU can support, is designed by EE faculty of UET Lahore. This board helps students get familiar with different peripherals of MCU by interacting with simple electronic components like seven segment display, 16x2 character LCD, temperature sensor (LM35), analog potentiometer, MAX232 and DB9 connector for interfacing UART using level shifter, real time clock (DS1307) for I2C interfacing. Figure 1.3 shows the expansion board with and without launchpad mounted on it.



(a) UET Launchpad expansion board (b) Launchpad mounted on expansion board

Figure 1.3: UET Launchpad Expansion board

Tiva C Series Overview

The TM4C123 (LM4F120) microcontrollers are based on the ARM Cortex-M4F core. The Cortex-M4 differs from previous generations of ARM processors by defining a number of key peripherals as part of the core architecture including interrupt controller, system timer and, debug and trace hardware (including external interfaces). This additional level of integration means that system software such as real-time operating systems and hardware development tools such as debugger interfaces can be common across the family of processors.
The TM4C (LM4F) microcontroller provides a wide range of connectivity features such as CAN, USB Device, SPI/SSI, I2C, UARTs. It supports high performance analog integration by providing two 1MSPS 12-bit ADCs and analog and digital comparators. It has best-in-class power consumption with currents as low as 370μA/MHz, 500μs wakeup from low-power modes

and RTC currents as low as 1.7μA. This Stellaris series offers a solid road map with higher speeds, larger memory and ultra low currents.

TM4C123GH6PM Microcontroller Overview

The TivaC TM4C123GH6PM microcontroller combines complex integration and high performance with the features shown in Figure 1.4.

Feature	Description
Performance	
Core	ARM Cortex-M4F processor core
Performance	80-MHz operation; 100 DMIPS performance
Flash	256 KB single-cycle Flash memory
System SRAM	32 KB single-cycle SRAM
EEPROM	2KB of EEPROM
Internal ROM	Internal ROM loaded with TivaWare™ for C Series software
Security	
Communication Interfaces	
Universal Asynchronous Receivers/Transmitter (UART)	Eight UARTs
Synchronous Serial Interface (SSI)	Four SSI modules
Inter-Integrated Circuit (I^2C)	Four I^2C modules with four transmission speeds including high-speed mode
Controller Area Network (CAN)	Two CAN 2.0 A/B controllers
Universal Serial Bus (USB)	USB 2.0 OTG/Host/Device
System Integration	
Micro Direct Memory Access (µDMA)	ARM® PrimeCell® 32-channel configurable µDMA controller
General-Purpose Timer (GPTM)	Six 16/32-bit GPTM blocks and six 32/64-bit Wide GPTM blocks
Watchdog Timer (WDT)	Two watchdog timers
Hibernation Module (HIB)	Low-power battery-backed Hibernation module
General-Purpose Input/Output (GPIO)	Six physical GPIO blocks
Advanced Motion Control	
Pulse Width Modulator (PWM)	Two PWM modules, each with four PWM generator blocks and a control block, for a total of 16 PWM outputs.
Quadrature Encoder Interface (QEI)	Two QEI modules
Analog Support	
Analog-to-Digital Converter (ADC)	Two 12-bit ADC modules, each with a maximum sample rate of one million samples/second
Analog Comparator Controller	Two independent integrated analog comparators
Digital Comparator	16 digital comparators
JTAG and Serial Wire Debug (SWD)	One JTAG module with integrated ARM SWD
Package Information	
Package	64-pin LQFP
Operating Range (Ambient)	Industrial (-40°C to 85°C) temperature range Extended (-40°C to 105°C) temperature range

Figure 1.4: TivaC TM4C123GH6PM Microcontroller Features

The Cortex-M4 core architecture consists of a 32-bit processor with a small set of key peripherals. The Cortex-M4 core has a Harvard architecture meaning that it uses separate interfaces

to fetch instructions and data. This helps ensure the processor is not memory starved as it permits accessing data and instruction memories simultaneously. From the perspective of the CM4, everything looks like memory – it only differentiates between instruction fetches and data accesses. The interface between the Cortex-M4 and manufacturer specific hardware is through three memory buses – ICode, DCode, and System – which are defined to access different regions of memory.

The block diagram of TivaC Launchpad evaluation board in Figure 1.5 gives an overview of how the Stellaris ICDI and other peripherals are interfaced with microcontroller.

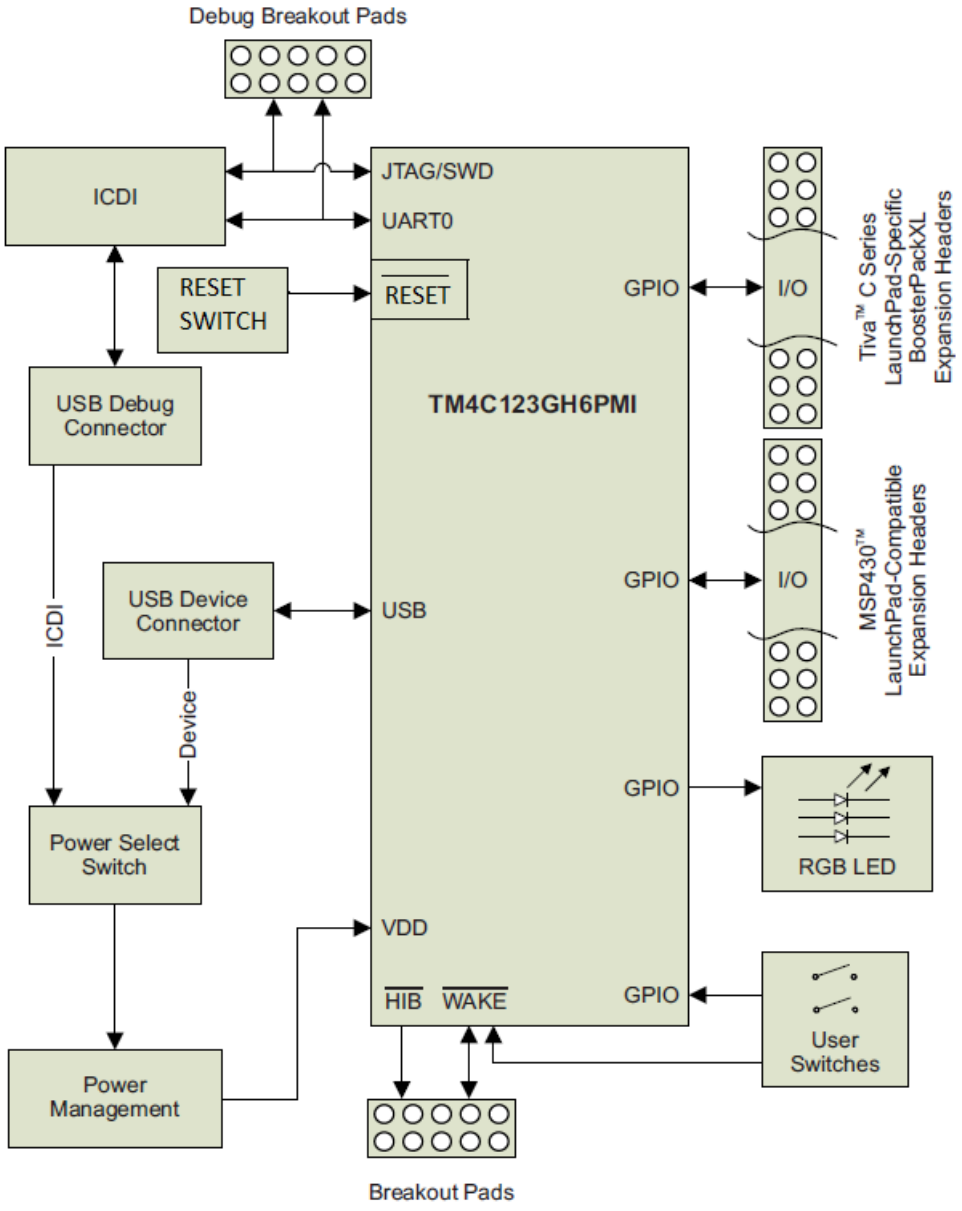


Figure 1.5: Block Diagram of TivaC Launchpad Board

Development Tools

To develop an application and run it on TivaC Launchpad, a software is required to write our code, debug it and download it to the device. Fortunately, many IDEs are available for the application development of TivaC Launchpad. Figure 1.6 shows different IDEs available for development. In this lab, we will use Keil μVision4 as our development tool.

	mentor embedded	IAR SYSTEMS	ARM KEIL	Code Composer Studio
Eval Kit License	**30-day full function. Upgradeable**	**32KB code size limited. Upgradeable**	**32KB code size limited. Upgradeable**	**Full function. Onboard emulation limited**
Compiler	**GNU C/C++**	**IAR C/C++**	**RealView C/C++**	**TI C/C++**
Debugger / IDE	**gdb / Eclipse**	**C-SPY / Embedded Workbench**	**µVision**	**CCS/Eclipse-based suite**

Figure 1.6: Development Tools

Setup Keil μVision to Write Code

1. Run the software by clicking the icon on desktop, if available, or by clicking on **Start** → **All Programs** →**Keil μVision**. An interface similar to one shown in Figure 1.7 will open.

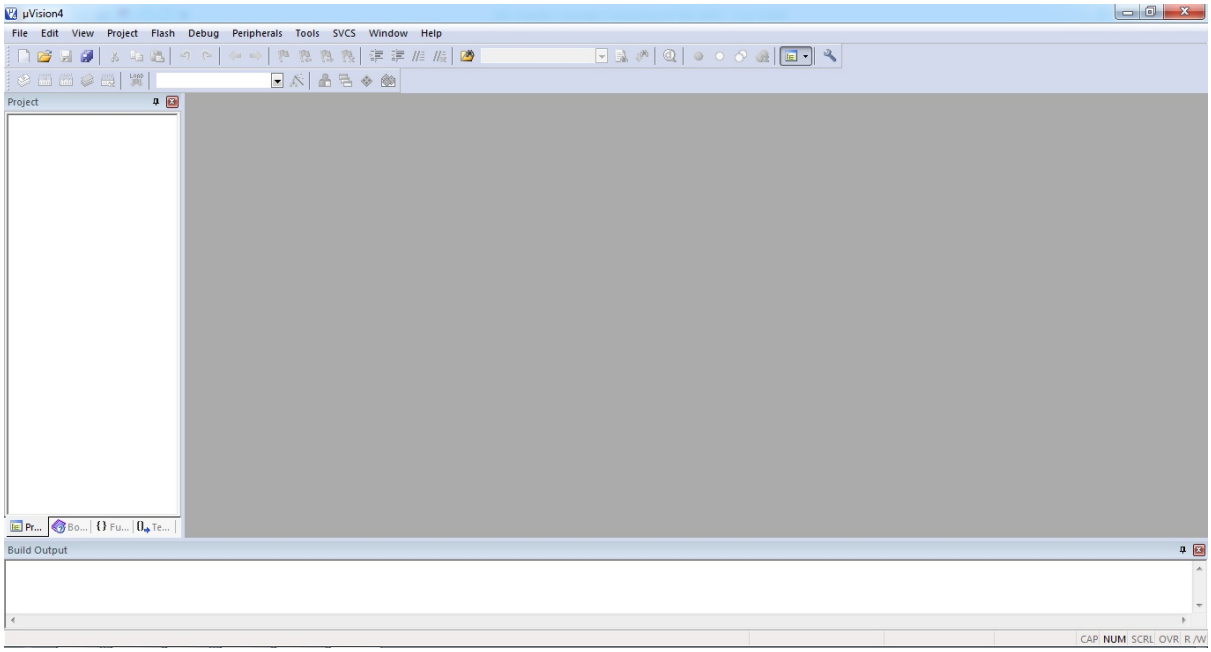


Figure 1.7: Keil interface on start

2. Click on *Project* tab and choose **New μVision Project** from the drop-down list as shown in Figure 1.8

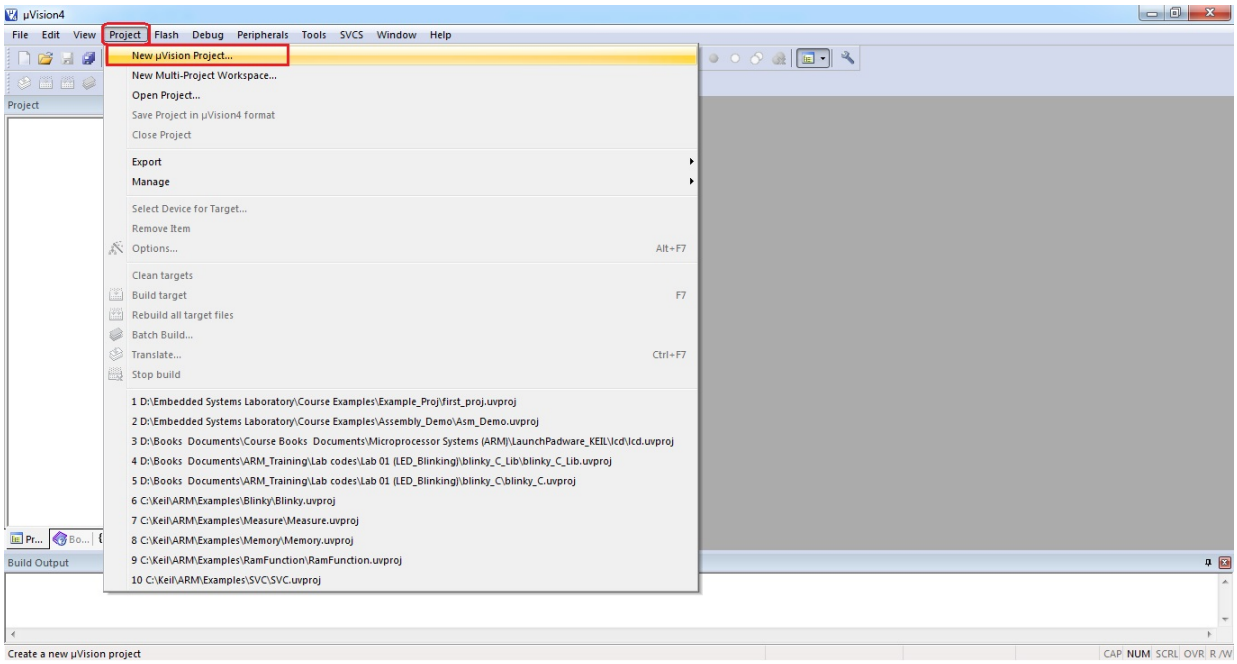


Figure 1.8: Create new μVision project

3. Select and create a directory, then assign a name to your project (project name can be different from folder name) then click on **Save**. **Do not make a directory, file or project name with a space in it**. A space will prevent simulation from working properly. (Figure 1.9).

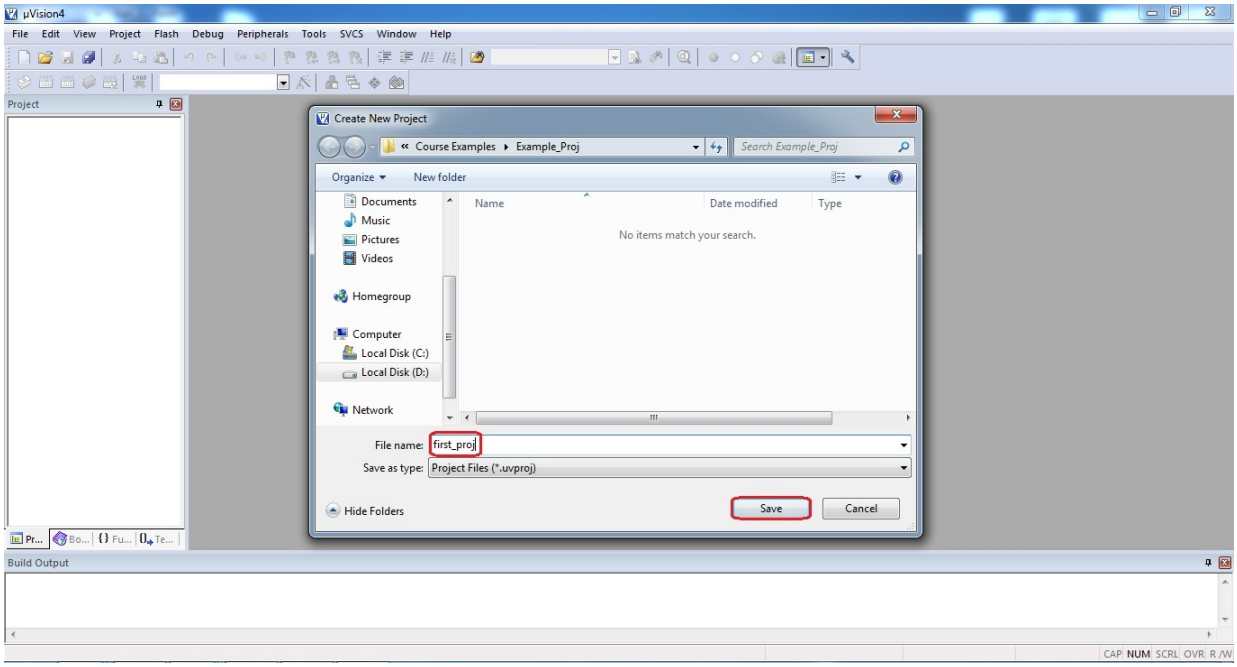


Figure 1.9: Type the name of the project in Keil and save it

4. To select a microcontroller double click on *Texas Instruments* and select **TM4C123GH6PM** or **TM4C1233H6PM** depending upon your microcontroller. Click *OK*.(See Figure 1.10 and 1.11)

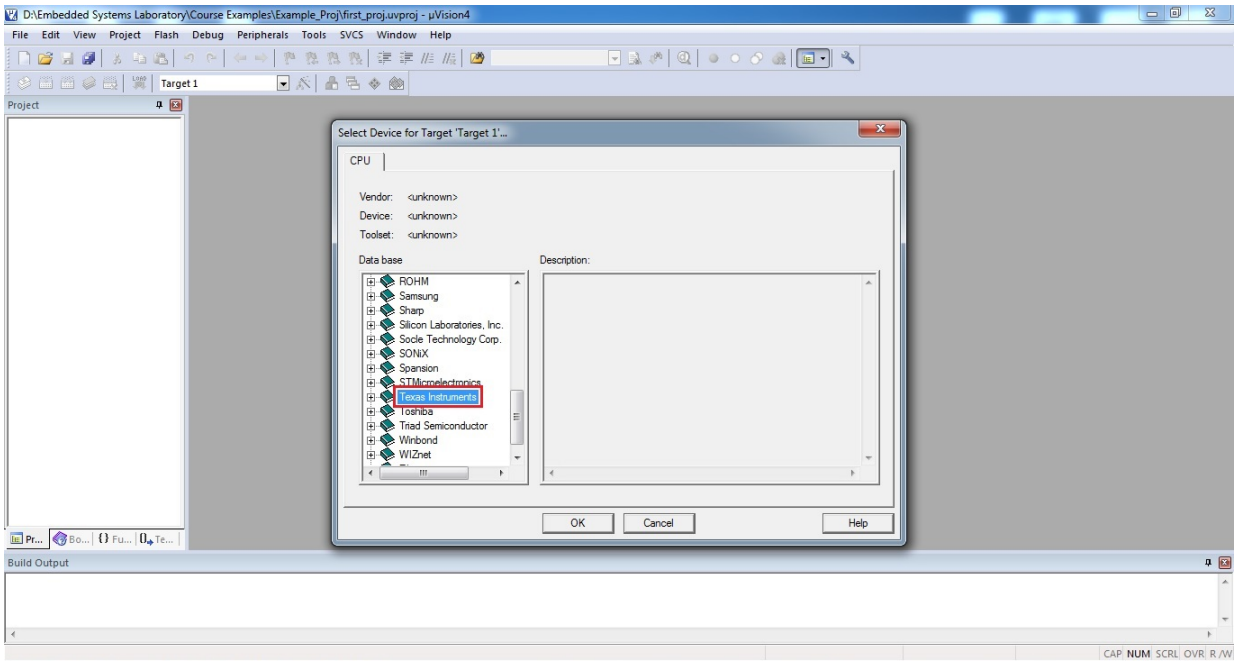


Figure 1.10: Select the manufacturer of your microcontroller

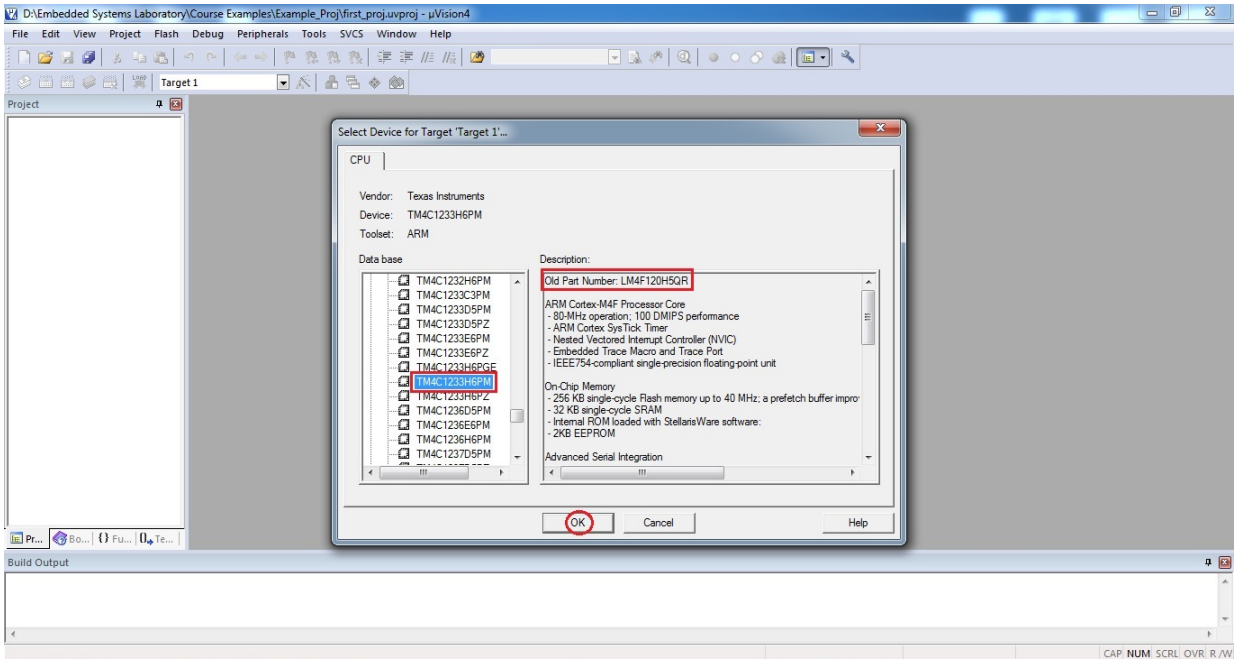


Figure 1.11: Select the part number for your microcontroller

5. When prompted to copy **'Startup_TM4C123.s to project folder'** click on *Yes* or *No* according to the requirement of your project (Figure 1.12).

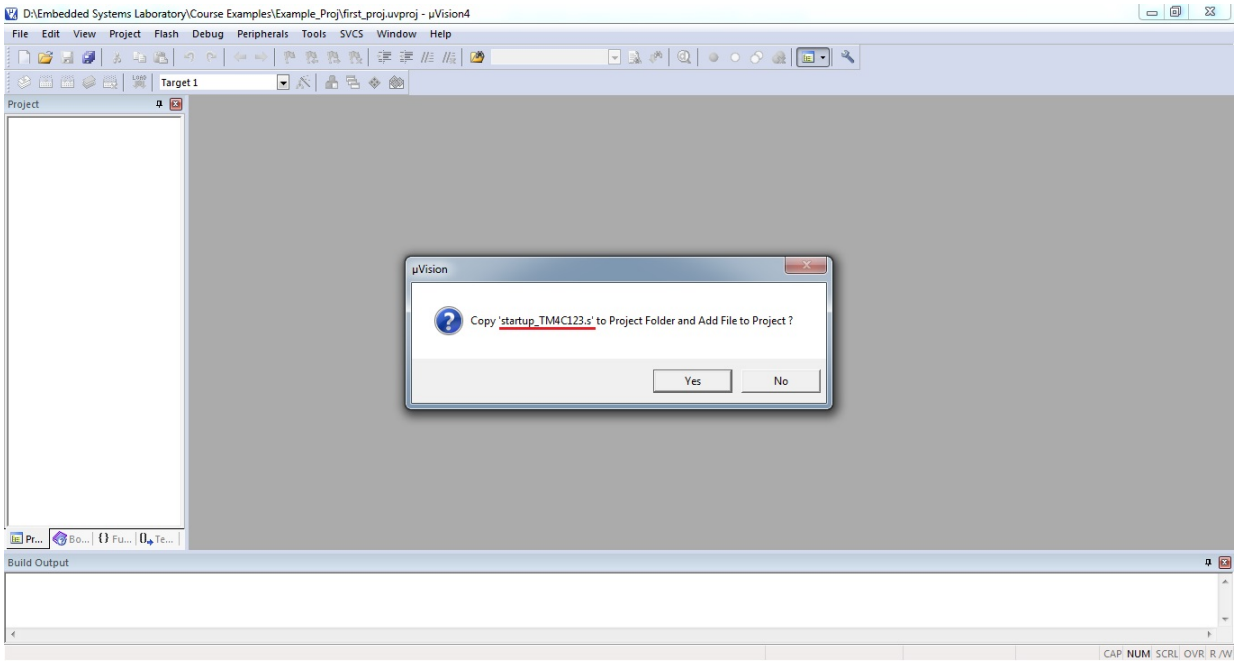


Figure 1.12: Add or discard the startup file to the project

6. Right click on *Source Group 1* under *Target 1*, click on **Add New Item to Group 'Source Group 1'...** and select the type of file you want to add (**.c** for C file), write its name in given space and click *OK* (Figure 1.13).

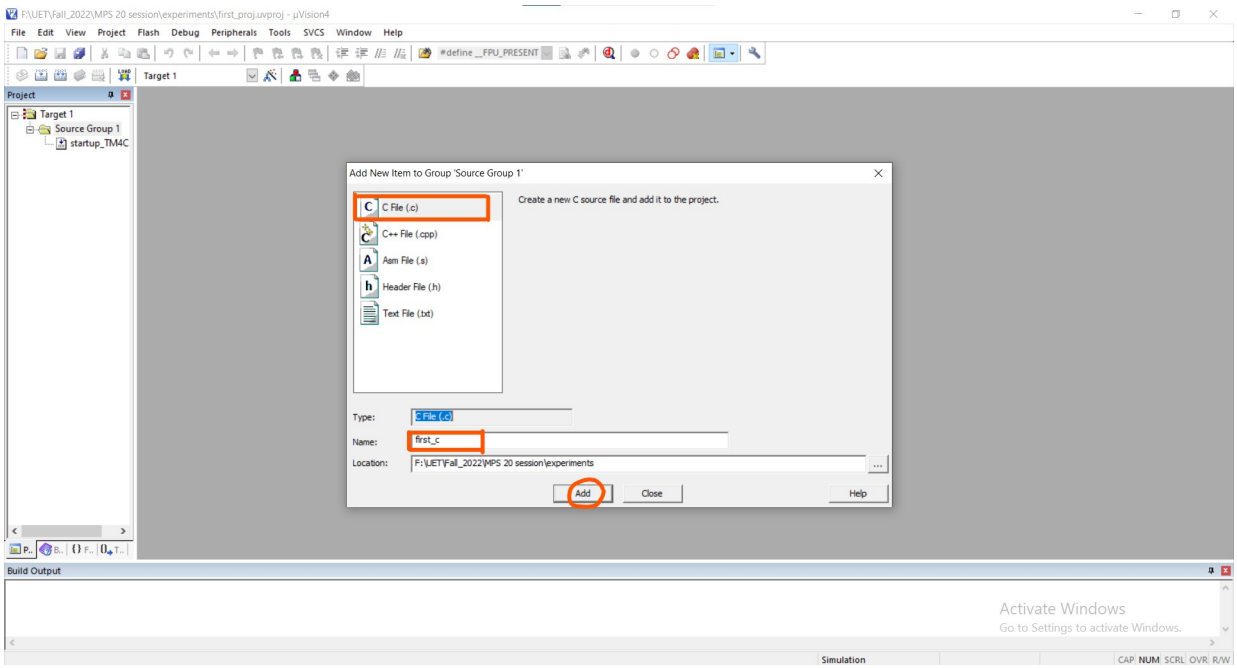


Figure 1.13: Add and save a new file to the project

7. Double click on the file name under *Source Group 1* in *Project* window to open it in the editor pane. Here, you can write and edit the code (Figure 1.14).

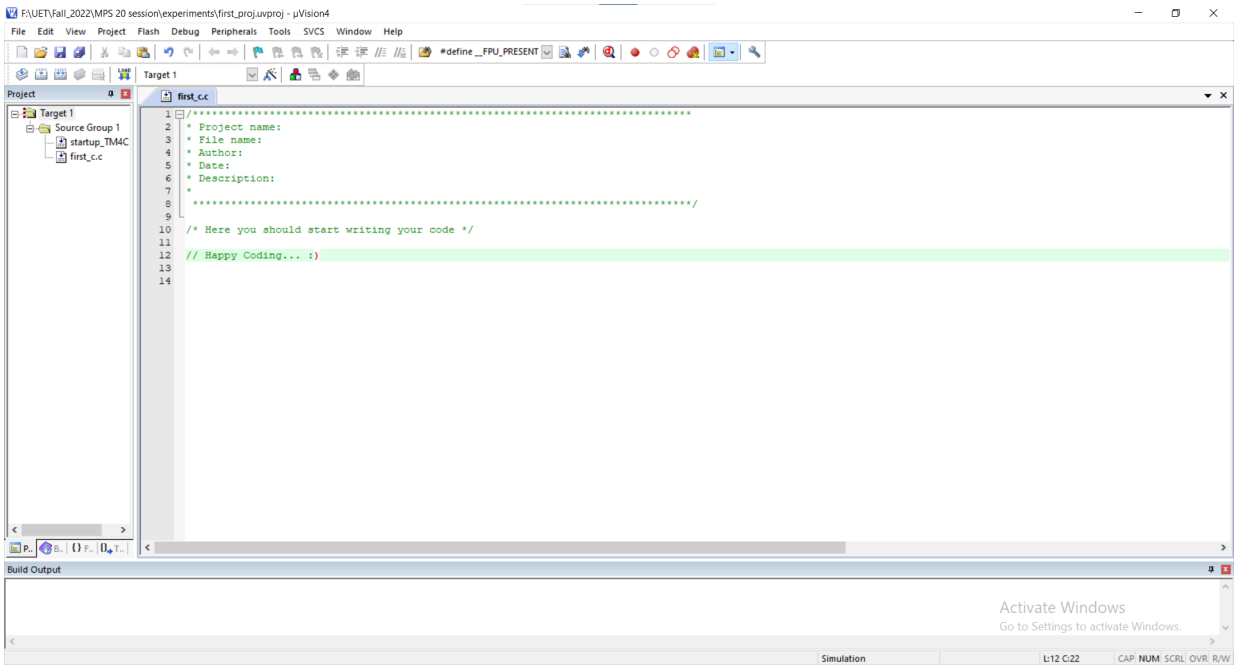


Figure 1.14: Edit the file in the text editor window

8. After writing your code, click *Build* to build the target files. *Build Output* window will provide information about *errors and warnings* in your code. After successful build, go into the debugger by clicking *Debug* (Figure 1.15).

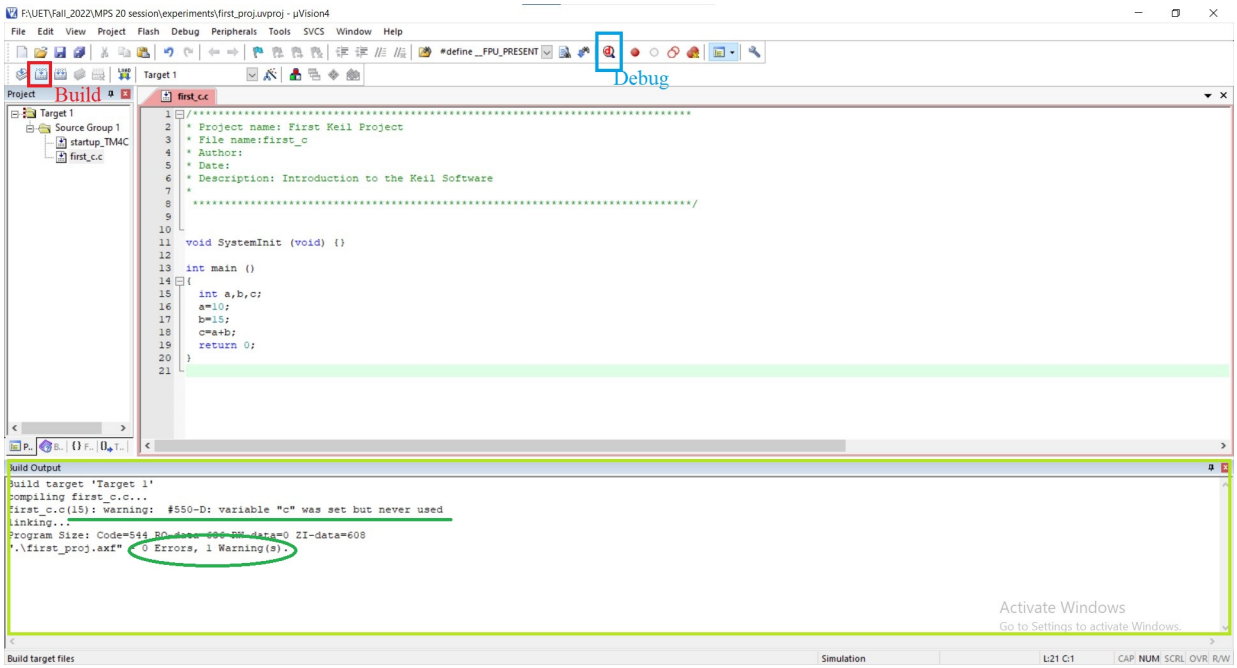


Figure 1.15: Build the target and enter Debug Mode

9. In Debug mode, step wise execution is used for viewing results of the code. Values in *Registers* window and *Call Stack + Locals* window will provide insight into the execution of the code. Assembly of the code is also available in the *Disassembly* window (Figure 1.16).

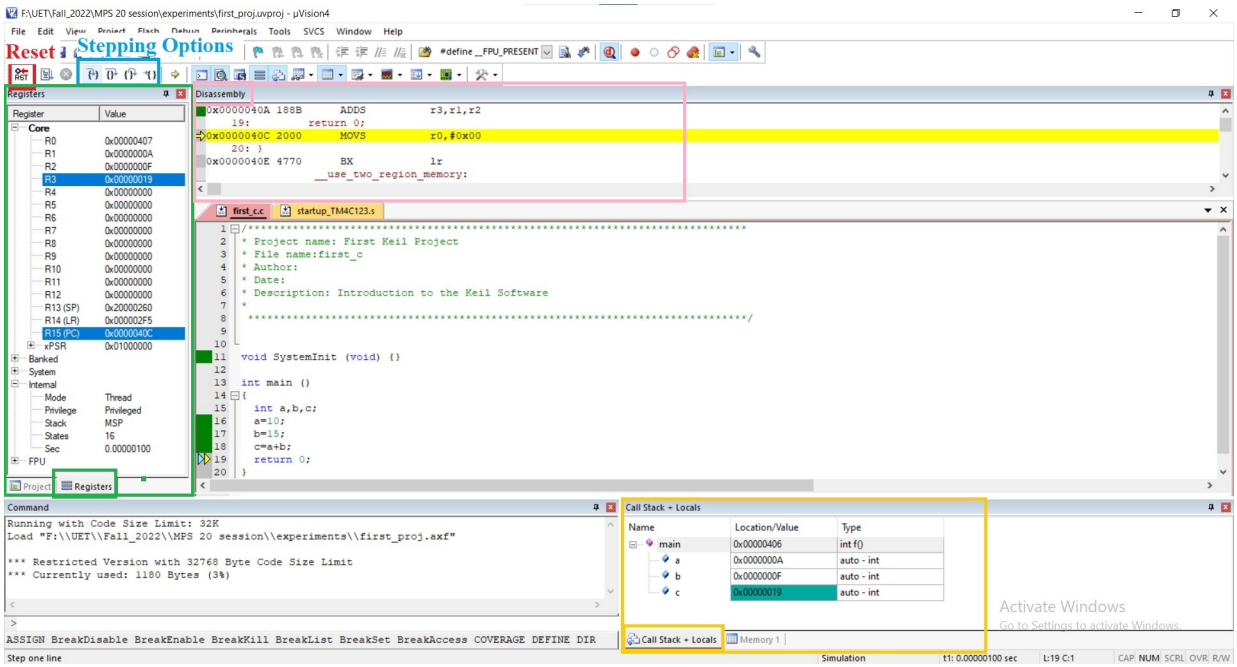


Figure 1.16: Debug the code using different stepping options by observing values in Register window and CallStack + Locals window

Using Keil μVision with TivaC Launchpad

For running the code on the launchpad, few steps need to be performed.

1. When Tiva Board is connected first time with your computer, its drivers need to be installed. Drivers are available at Texas Instruments site. For this purpose:

   (a) Go to *Device Manager* and expand *Other Devices*. When Tiva is connected for the first time, two *In-Circuit Debug Interface* will appear.

   (b) Right click on these and select *Update Driver*. When prompted, choose *Browse my computer for drivers* and provide the path to the folder where the drivers downloaded have been **extracted**.

   (c) Repeat this for the second *In-Circuit Debug Interface*.

   (d) After completing this procedure, Device Manager will have Stellaris ICDI drivers installed.

2. In Keil μVision, *Target Options* need to be changed for loading code on the microcontroller (Figure 1.17).

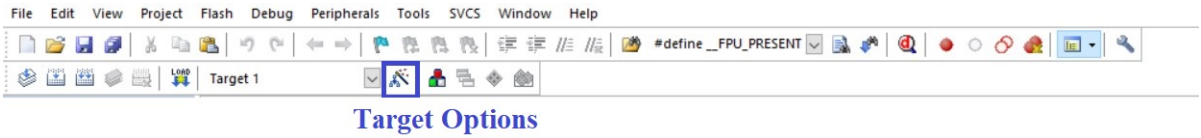


Figure 1.17: Target Options

3. In *Output* tab, tick **Create Hex File** to create the binary to be loaded (Figure 1.18).

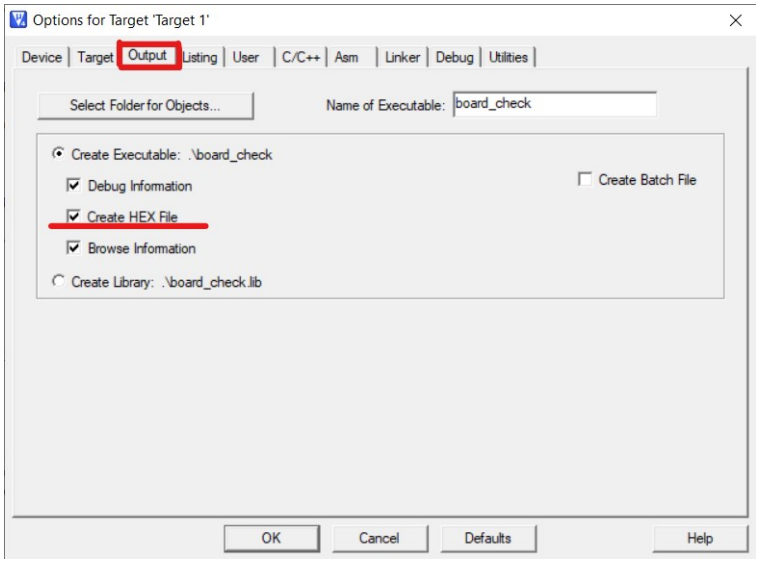


Figure 1.18: Select option Create Hex File

4. In *Debug* tab, choose *Use* instead of Simulator and change the drivers to **Stellaris ICDI** (Figure 1.19).

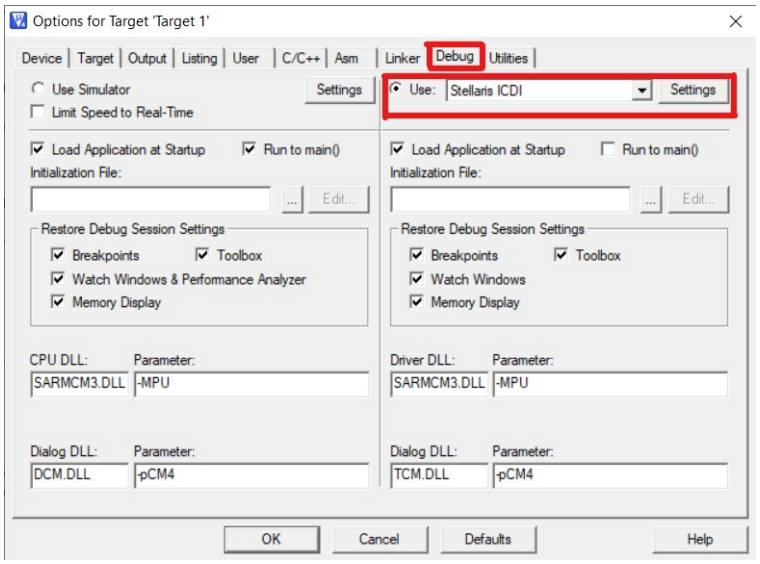


Figure 1.19: Change Simulation option to Use Stellaris ICDI

5. Click on *Settings* next to it and make sure they match Figure 1.20.

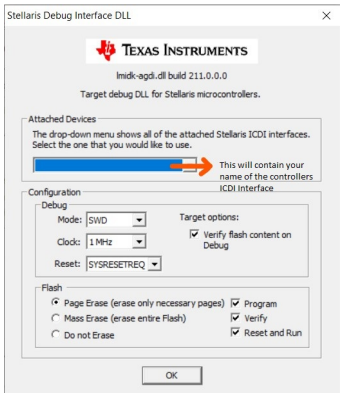


Figure 1.20: Settings for Using Stellaris ICDI

6. Click on *Load* to download code to the microcontroller's Flash Memory (Figure 1.21) Microcontroller will start executing the code loaded onto it.

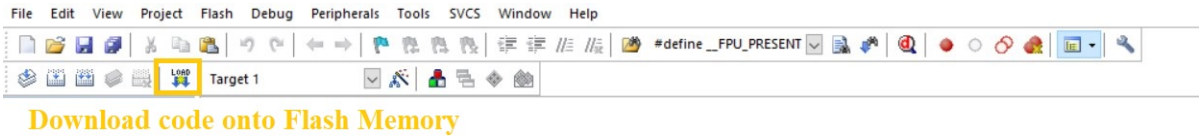


Figure 1.21: Load the code into Flash Memory

Enabling FPU for __main function

The function __main is the entry point to the C library. The __main function is responsible for zeroing out zero initialized regions and initializing global variables (copying variables from FLASH to proper positions in RAM) among other tasks. Then it jumps to your main function. Further details can be seen at https://developer.arm.com/documentation/dai0241/b/ and https://developer.arm.com/documentation/100748/0618/Embedded-Software-Development/Application-sta

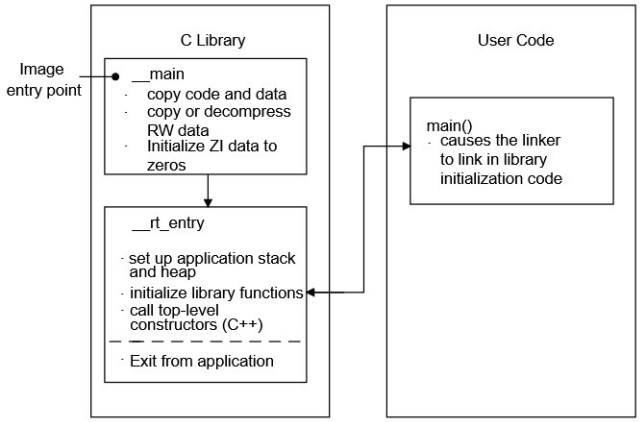


Figure 1.22: Default initialization sequence

This function requires Floating Point Unit which needs to be enabled. Either you can do it in

Reset Handler in startup file or in SystemInit Function in C.

```
Reset_Handler PROC
          EXPORT  Reset_Handler             [WEAK]
          IMPORT  SystemInit
          IMPORT  __main
          ; CPACR is located at address 0xE000ED88
          LDR.W R0, =0xE000ED88
          ; Read CPACR
          LDR R1, [R0]
          ; Set bits 20-23 to enable CP10 and CP11 coprocessors
          ORR R1, R1, #(0xF << 20)
          ; Write back the modified value to the CPACR
          STR R1, [R0]; wait for store to complete
          DSB
          ;reset pipeline now the FPU is enabled
          ISB
          LDR     R0, =SystemInit
          BLX     R0
          LDR     R0, =__main
          BX      R0
          ENDP
```

Modified Reset Handler to enable FPU

```
#include "TM4C123.h"

void SystemInit (void)
{
  /* ----------------------FPU settings ------------------------*/
  #if (__FPU_USED == 1)
  /* set CP10, CP11 Full Access*/
  SCB->CPACR |= ((3UL << 10*2) | (3UL << 11*2) );
  #endif

}
```

SystemInit function to enable FPU

Also remember to add breakpoint on the first line of your main code so you can do step by step debugging. Otherwise, it will just run the entire program after coming out of __main.