Contents

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

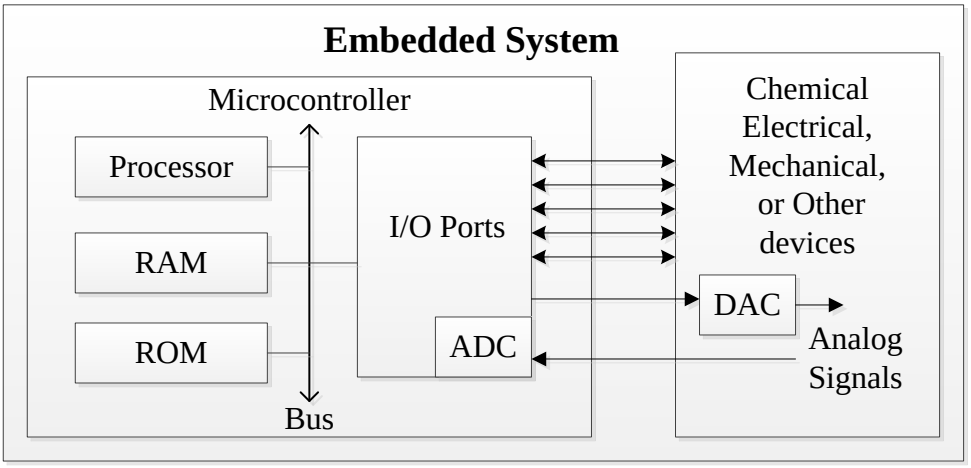


Figure 1.1: Block diagram of an embedded system

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

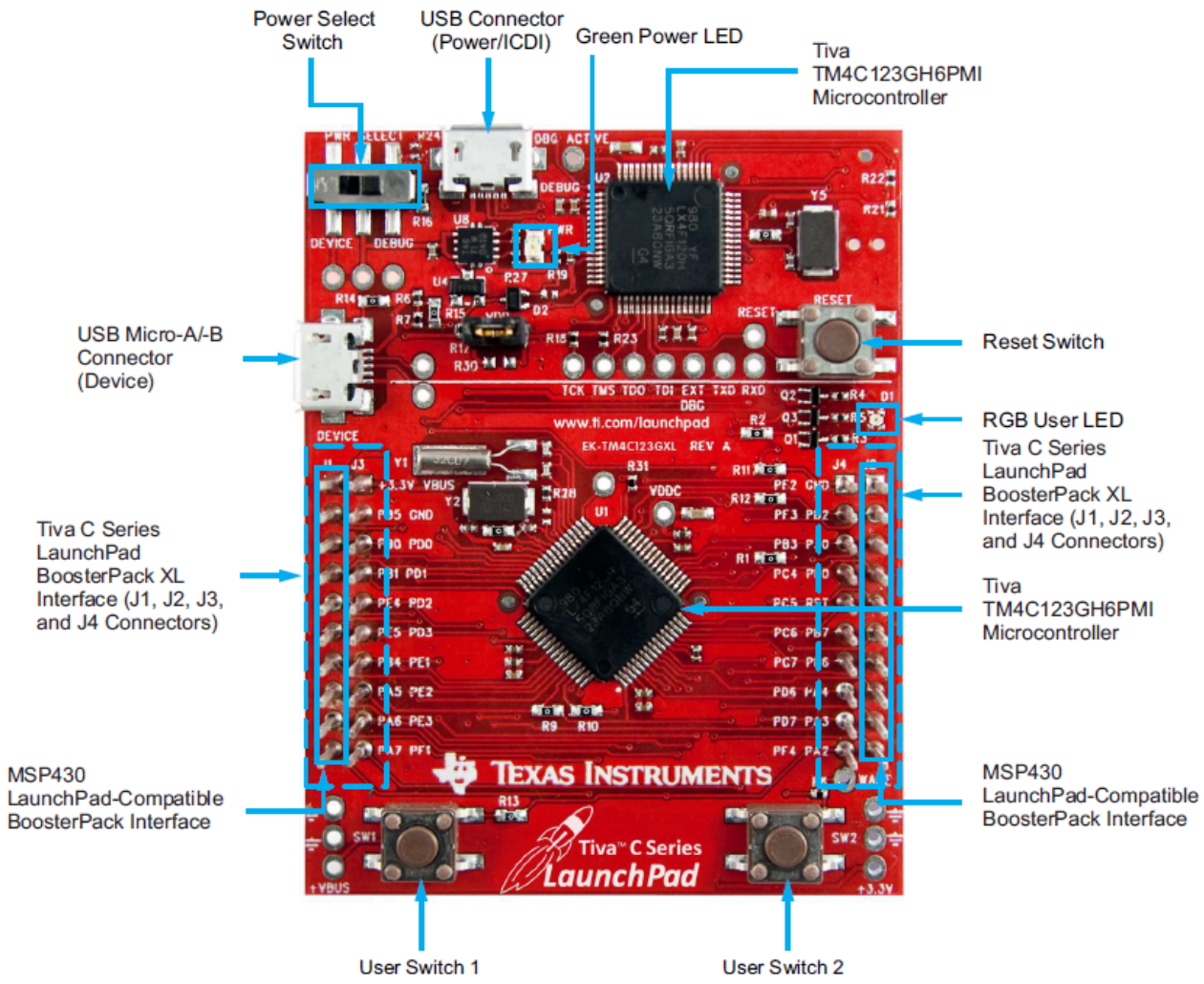


Figure 1.2: Launchpad Board

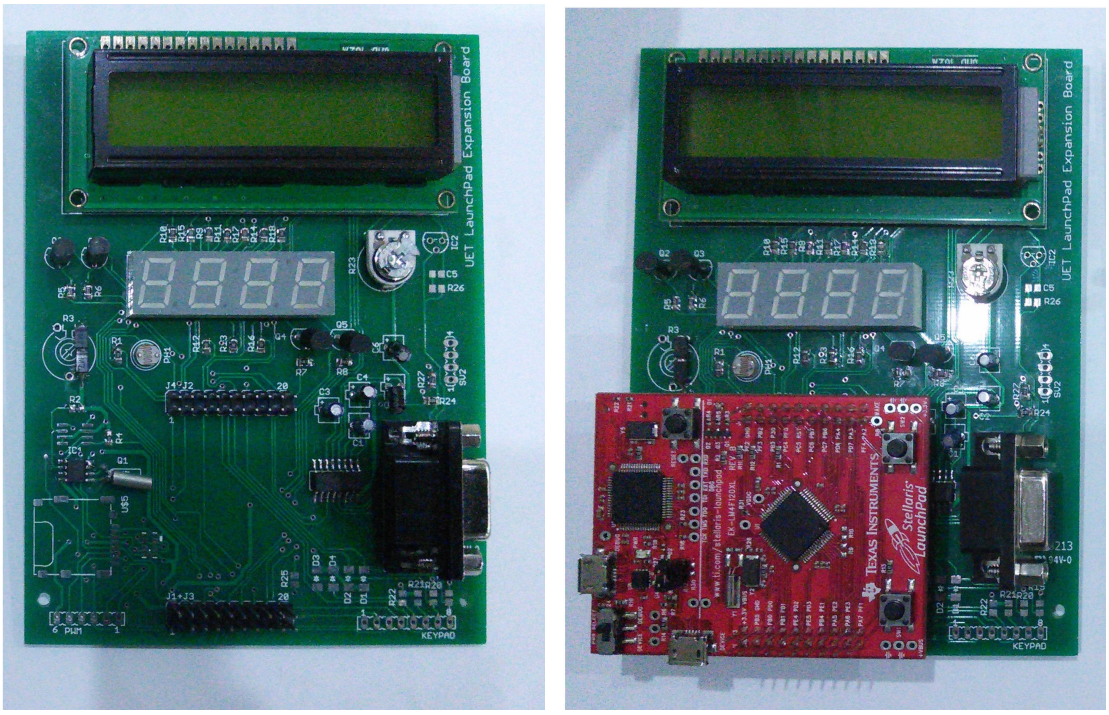
a 3.3V voltage regulator and other components. The regulator converts the 5V supplied by the USB connection to 3.3V for the processors and also available at the board edge connectors.

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.

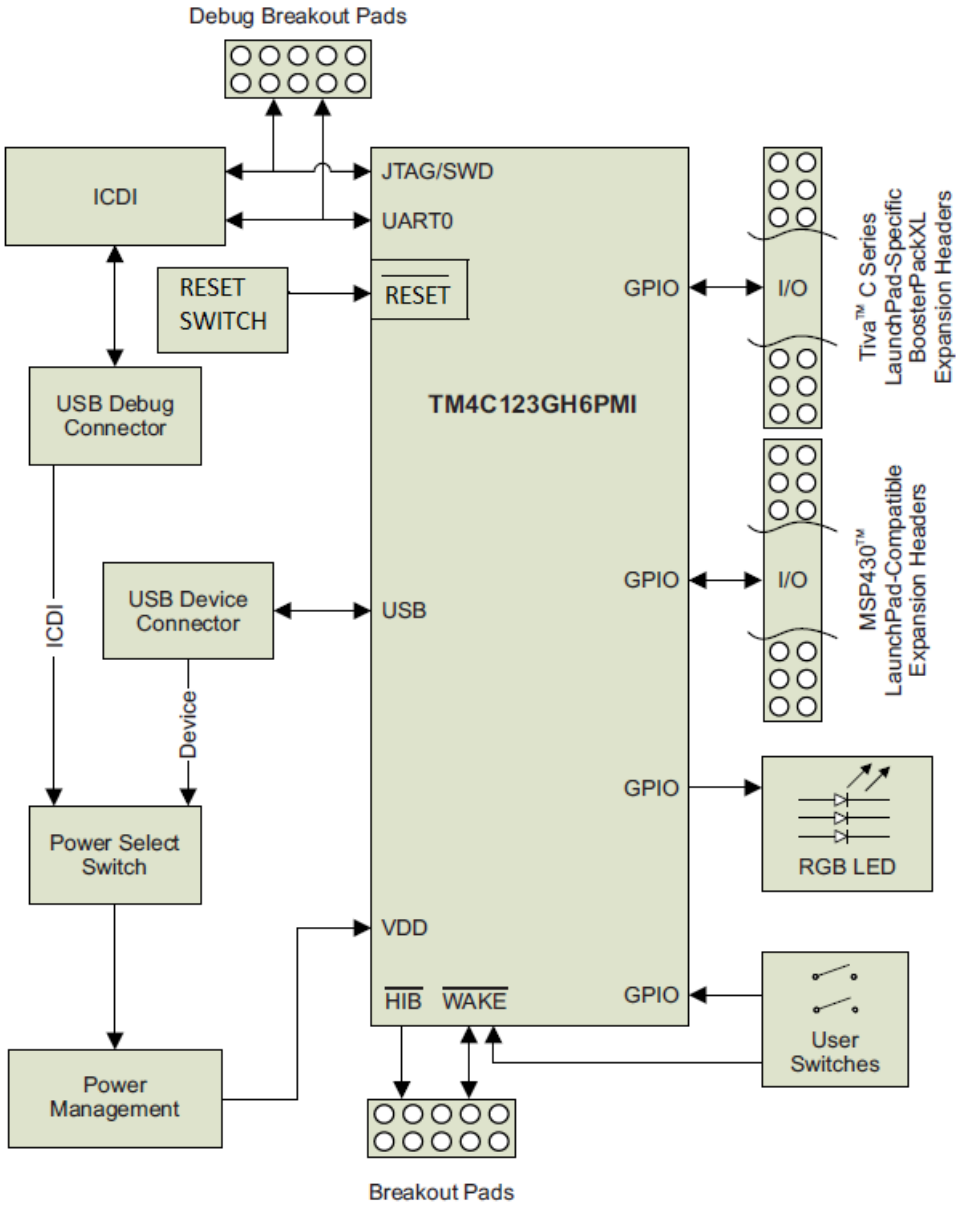


Figure 1.5: Block Diagram of TivaC Launchpad Board

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.

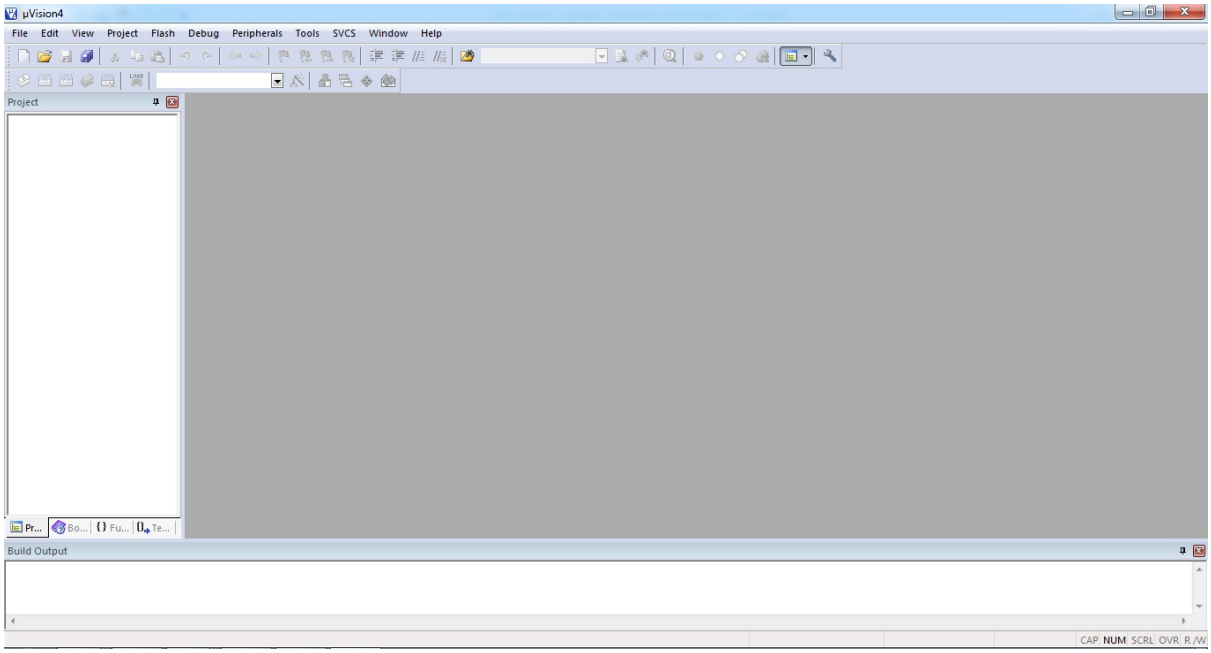


Figure 1.7: Keil interface on start

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

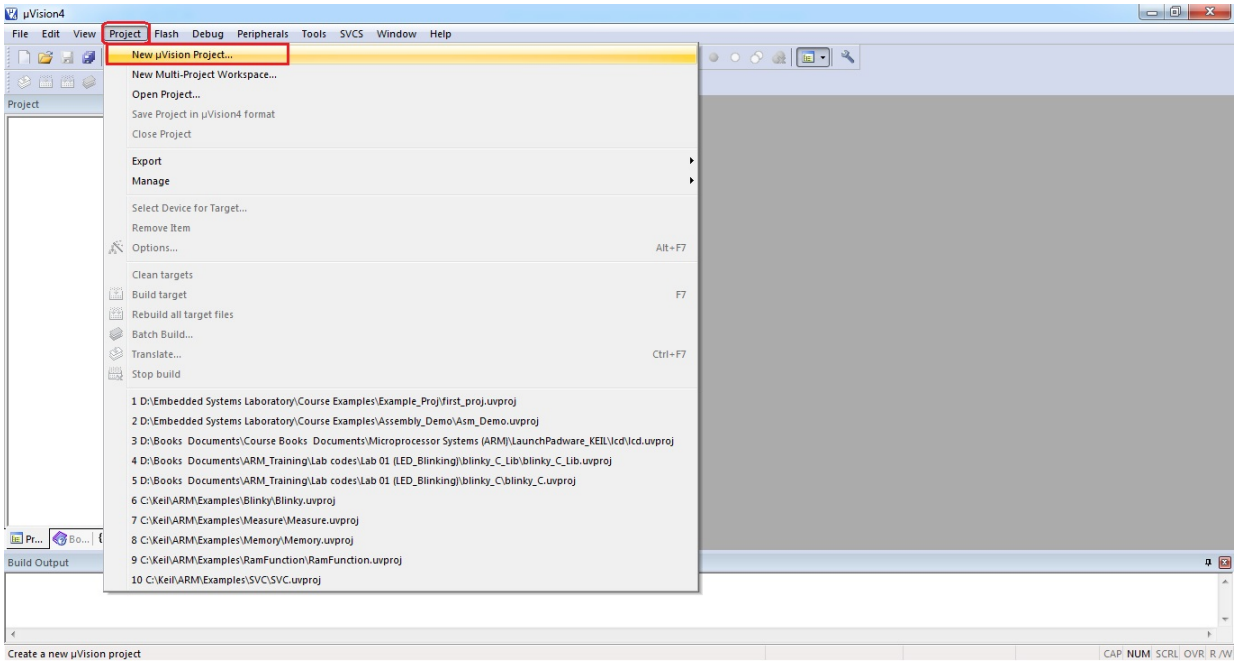


Figure 1.8: Create new μVision project

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).

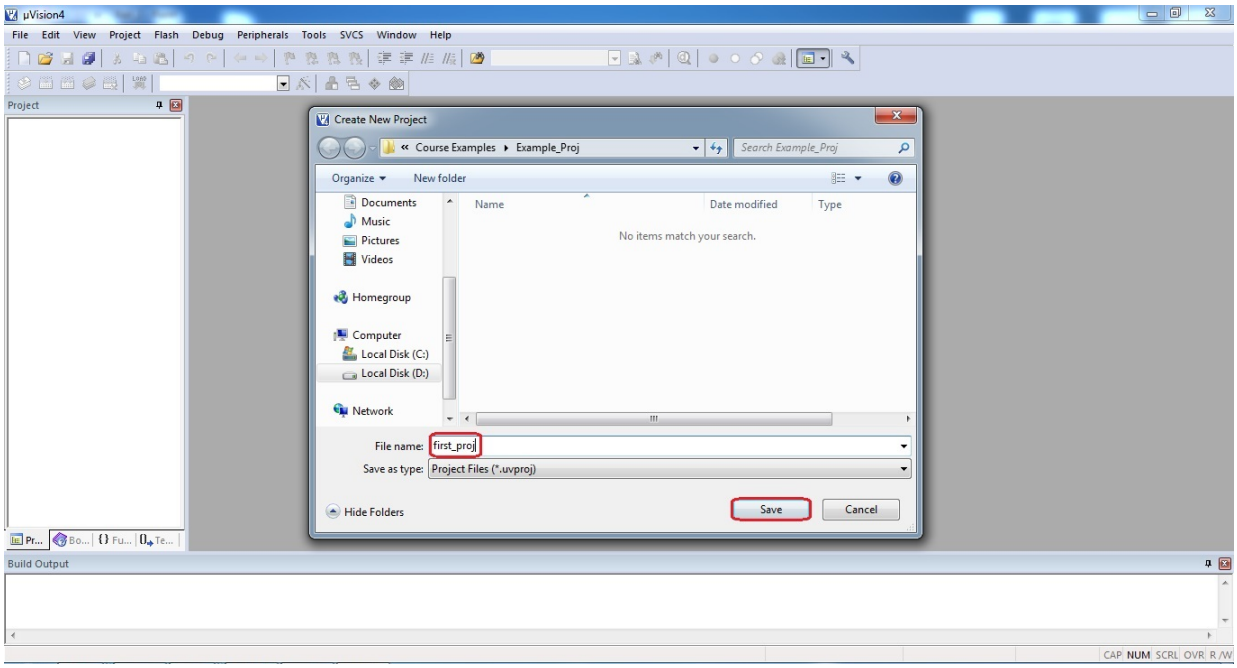


Figure 1.9: Type the name of the project in Keil and save it

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)

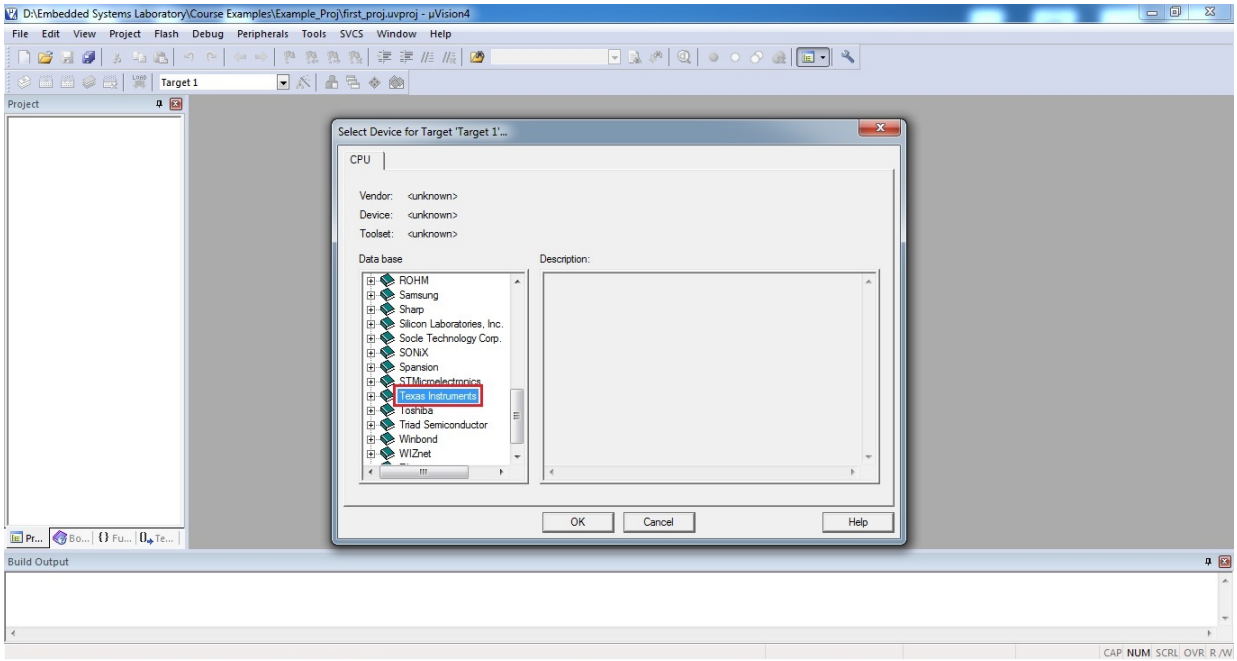


Figure 1.10: Select the manufacturer of your microcontroller

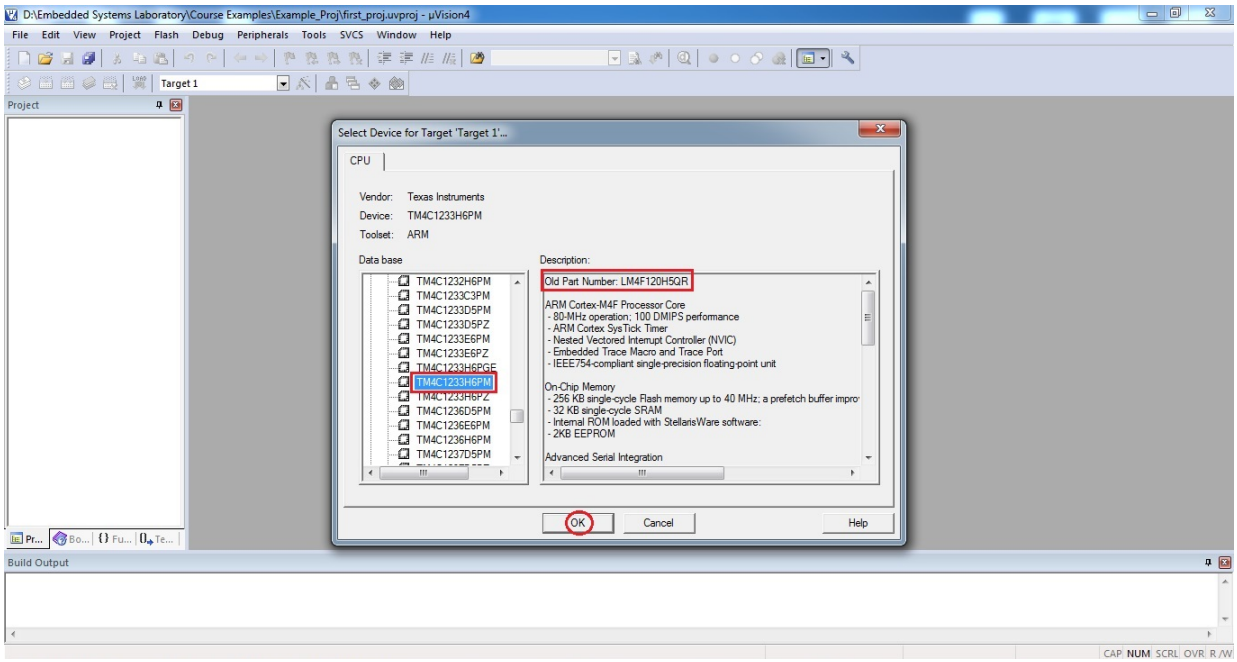


Figure 1.11: Select the part number for your microcontroller

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).

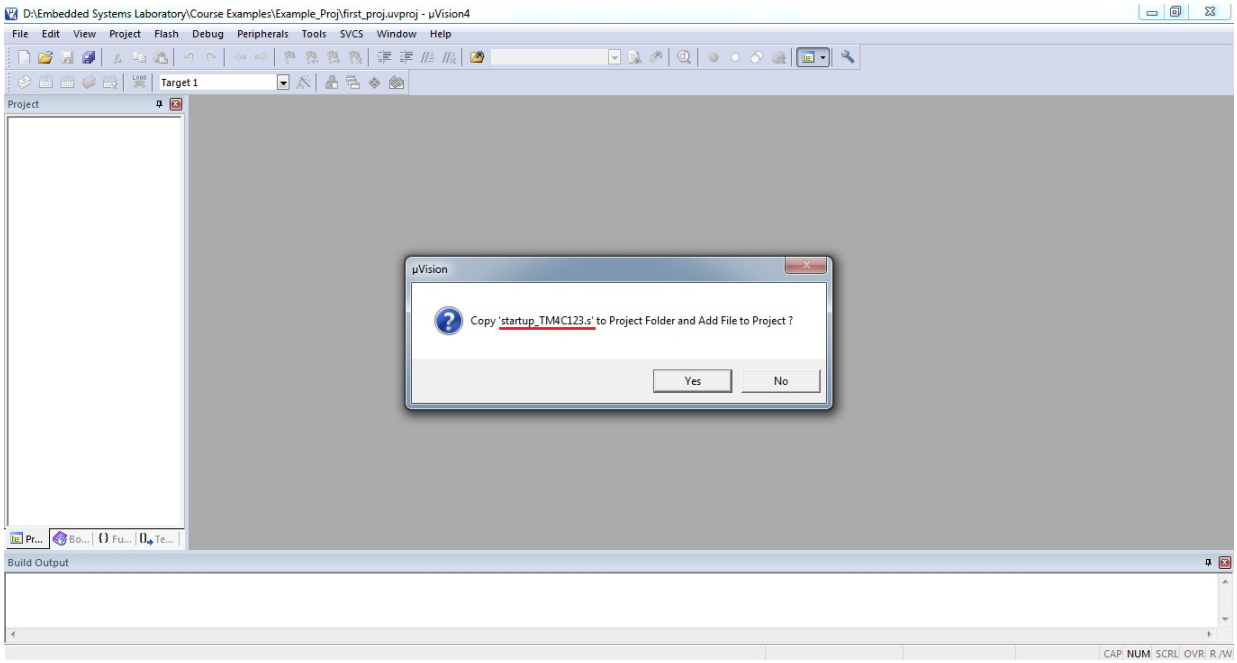


Figure 1.12: Add or discard the startup file to the project

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 ??).

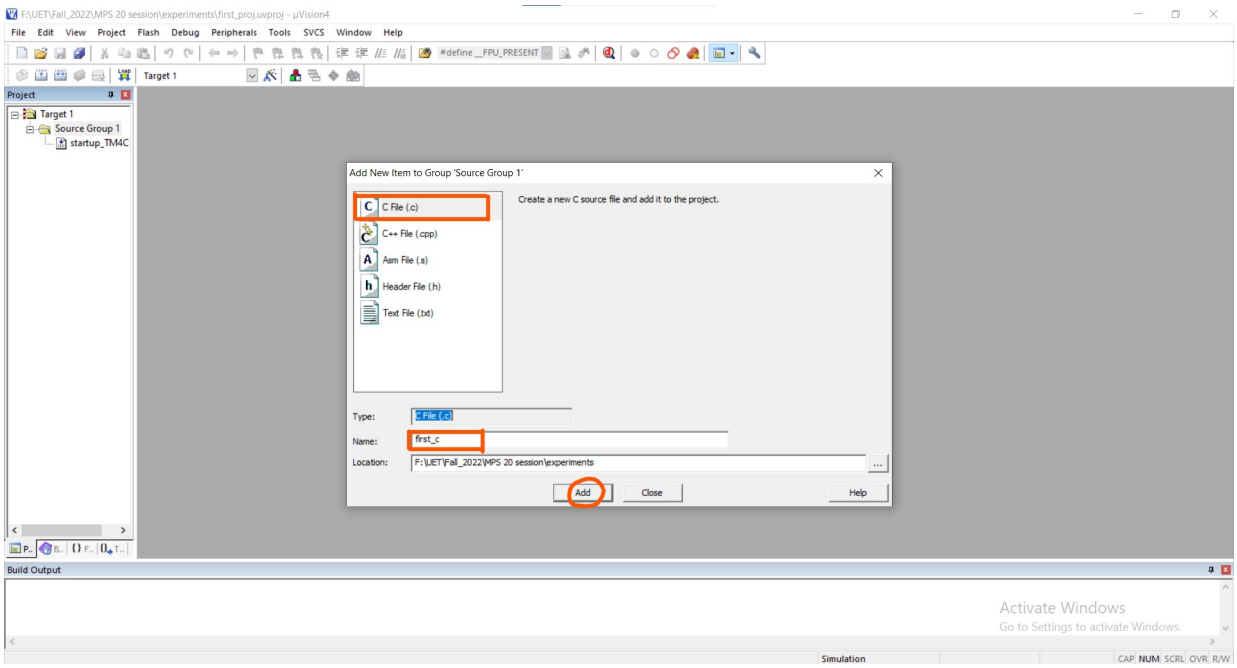


Figure 1.13: Add and save a new file to the project

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 ??).

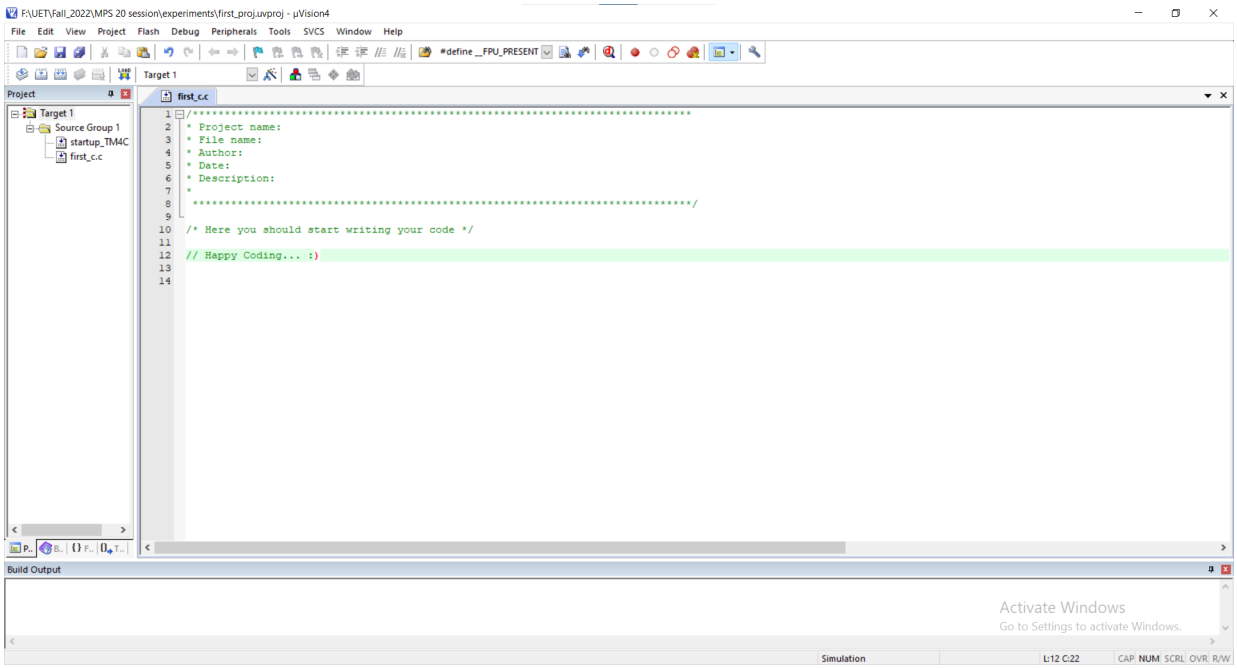


Figure 1.14: Edit the file in the text editor window

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 ??).

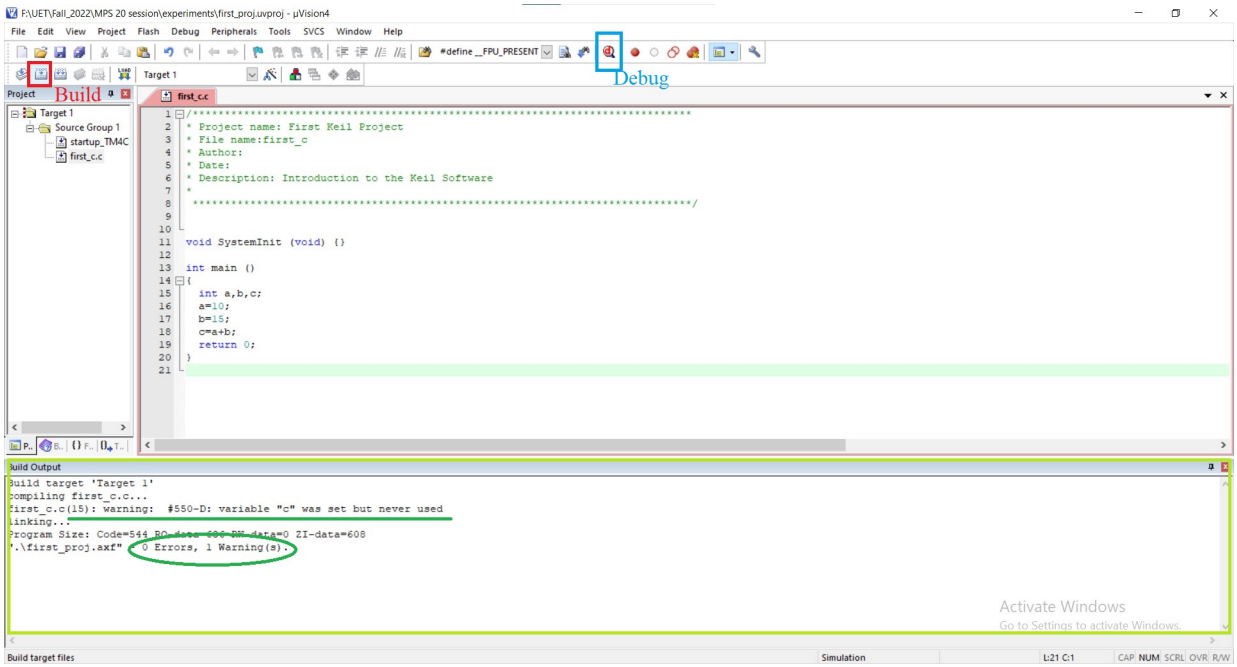


Figure 1.15: Build the target and enter Debug Mode

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 ??).

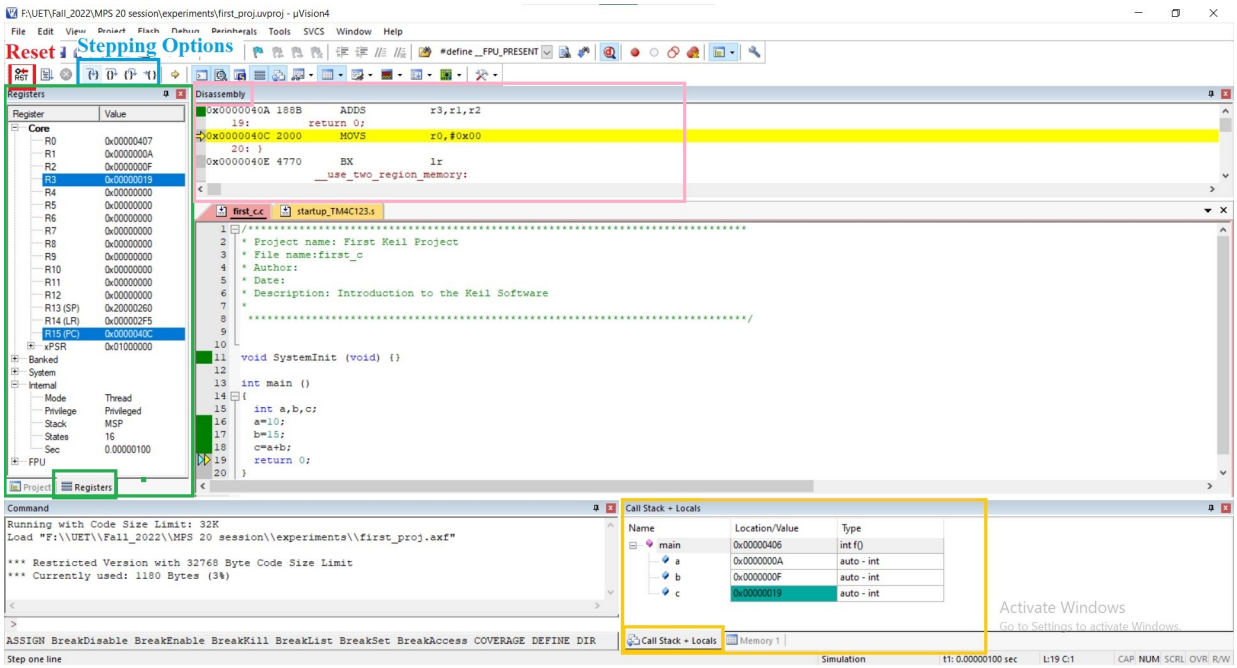


Figure 1.16: Debug the code using different stepping options by observing values in Register window and CallStack + Locals window

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 ??).

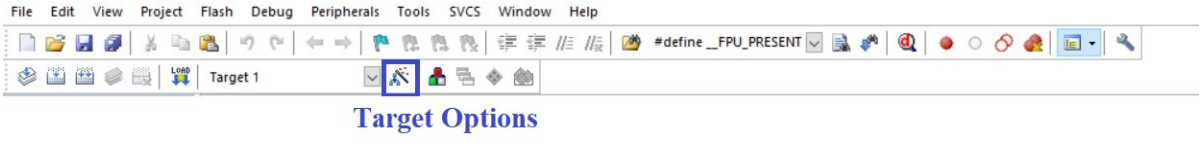


Figure 1.17: Target Options

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

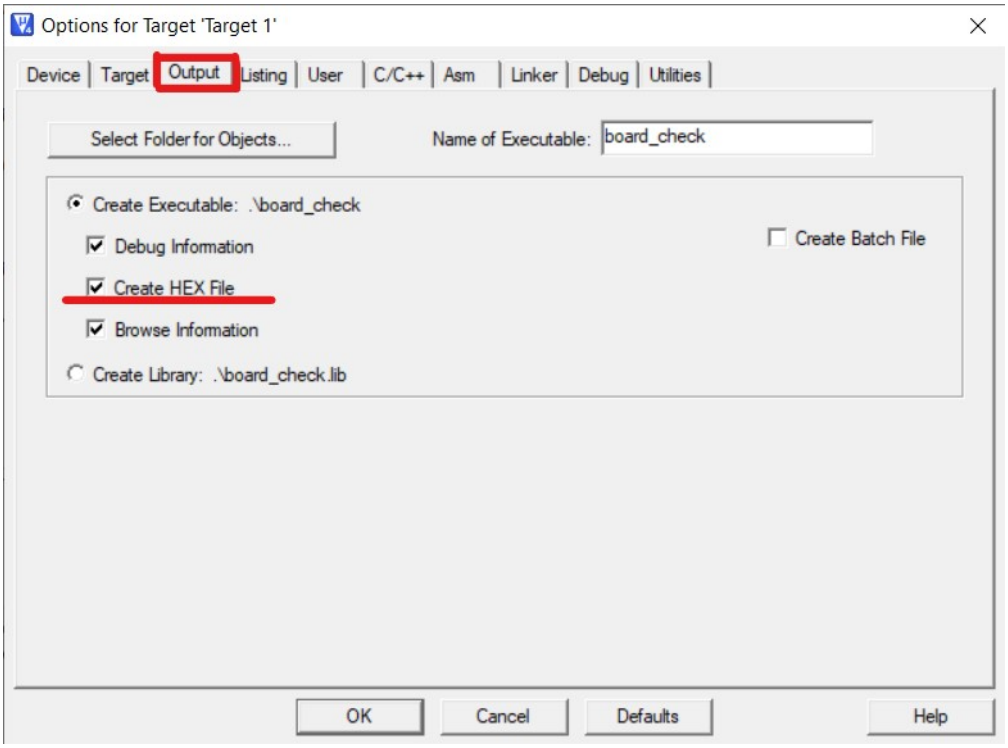


Figure 1.18: Select option Create Hex File

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

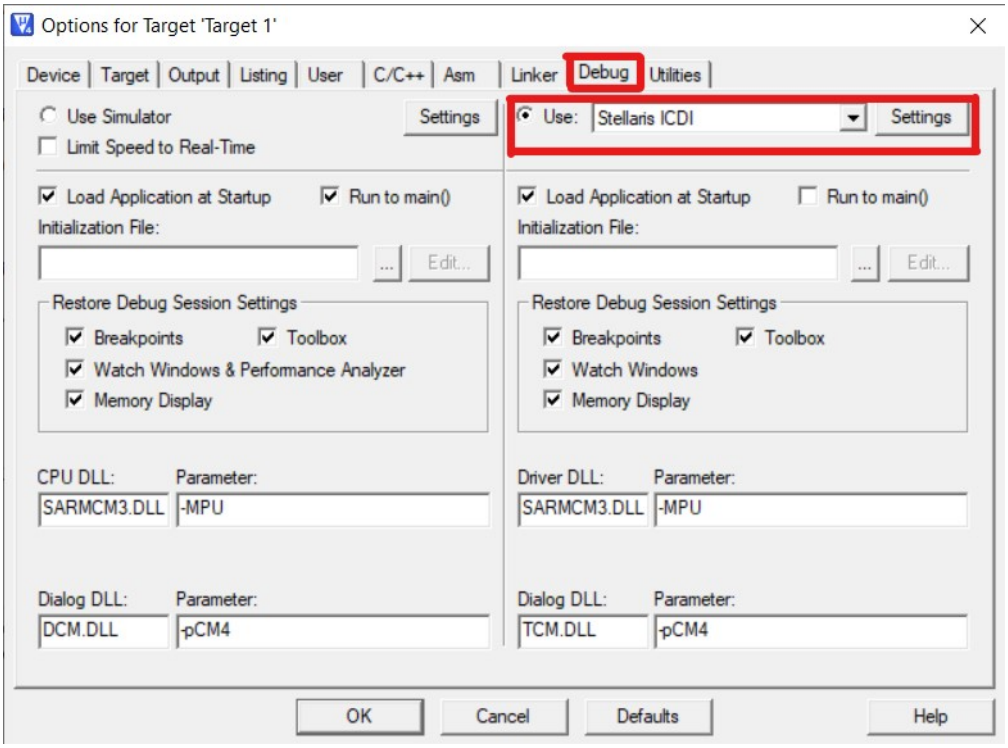


Figure 1.19: Change Simulation option to Use Stellaris ICDI

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

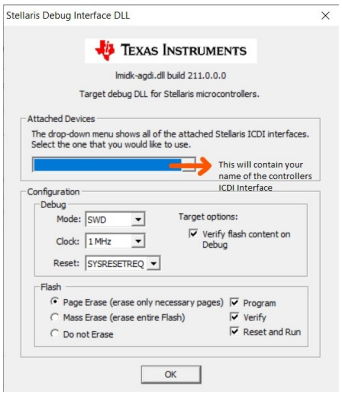


Figure 1.20: Settings for Using Stellaris ICDI

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

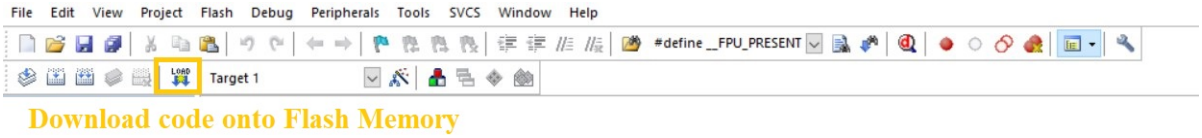


Figure 1.21: Load the code into Flash Memory

Experiment 2

C Language Programming

During the infancy years of microprocessor based systems, programs were developed using assemblers and fused into the EPROMs. There used to be no mechanism to find what the program was doing. LEDs, switches, etc. were used to check correct execution of the program. Some 'very fortunate' developers had In-circuit Simulators (ICEs), but they were too costly and were not quite reliable as well. As time progressed, use of microprocessor-specific assembly-only as the programming language reduced and embedded systems moved onto C as the embedded programming language of choice. C is the most widely used programming language for embedded processors/controllers. Assembly is also used but mainly to implement those portions of the code where very high timing accuracy, code size efficiency, etc. are prime requirements.

Embedded Systems Programming

Embedded programs must work closely with the specialized components and custom circuitry that makes up the hardware. Unlike programming on top of a full-function operating system, where the hardware details are removed as much as possible from the programmer's notice and control, most embedded programming acts directly with and on the hardware. This includes not only the hardware of the CPU, but also the hardware which makes up all the peripherals (both on-chip and off-chip) of the system. Thus an embedded programmer must have a good knowledge of hardware, at least as it pertains to writing software that correctly interfaces with and manipulates that hardware. This knowledge will often extend to specifying key components of the hardware (microcontroller, memory devices, I/O devices, etc.) and in smaller organizations will often go as far as designing and laying out the hardware as a printed circuit board. An embedded programmer will also need to have a good understanding of debugging equipment such as multimeters, oscilloscopes and logic analyzers.

Another difference from more general purpose computers is that most embedded systems are quite limited as compared to the former. The microcomputers used in embedded systems may have program memory sizes of a few thousand to a few hundred thousand bytes rather than the gigabytes in the desktop machine, and will typically have even less data (RAM) memory than program memory.

There are many factors to consider when developing a program for embedded systems. Some of them are:

- Efficiency - Programs must be as short as possible and memory must be used efficiently.
- Speed - Programs must run as fast as possible.

- Ease of implementation.
- Maintainability
- Readability

C Programming Language for Embedded Systems

Embedded systems are commonly programmed using C, assembly and BASIC. C is a very flexible and powerful programming language, yet it is small and fairly simple to learn. C gives embedded programmers an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages so its compilers are available for almost every processor in use today and there is a very large body of experienced C programmers.

C used for embedded systems is slightly different as compared to C used for general purpose programming (under a PC platform). Programs that are developed for embedded systems are usually expected to monitor and control external devices and directly manipulate and use the internal architecture of the processor such as interrupt handling, timers, serial communications and other available features. C compilers for embedded systems must provide ways to examine and utilize various features of the microcontroller's internal and external architecture; this includes interrupt service routines, reading from and writing to internal and external memories, bit manipulation, implementation of timers/counters and examination of internal registers etc. Standard C compiler, communicates with the hardware components via the operating system of the machine but the C compiler for the embedded system must communicate directly with the processor and its components. For example, consider the following C language statements:

```
printf("C Programming for Embedded Systems\n");
c = getchar();
```

In standard C running on a PC platform, the *printf* statement causes the string inside the quotation to be displayed on the screen. The same statement in an embedded system causes the string to be transmitted via the serial port pin (i.e., TxD) of the microcontroller provided the serial port has been initialized and enabled. Similarly, in standard C running on a PC platform *getchar()* causes a character to be read from the keyboard on a PC. In an embedded system the instruction causes a character to be read from the serial pin (i.e., RxD) of the microcontroller.

Template for Embedded C Program

```
#include "lm4f120h5qr.h"

void main(void)
{
// body of the program goes here
}
```

Template for Embedded C Program

- The first line of the template is the C directive. This tells the compiler that during compilation, it should look into this file for symbols not defined within the program.
- The next line in the template declares the beginning of the body of the main part of the program. The main part of the program is treated as any other function in C program. Every C program should have a main function.
- Within the curly brackets you write the code for the application you are developing.

Preprocessor Directives

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. Preprocessor directives begin with a hash symbol (#) in the first column. As the name implies, preprocessor commands are processed first.i.e., the compiler parses through the program handling the preprocessor directives.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control etc. Here we discuss only two important preprocessor directives:

Macro Definitions (#define)

A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. Object-like macros resemble data objects when used, function-like macros resemble function calls.

Object-like Macros

An object-like macro is a simple identifier which will be replaced by a code fragment. It is called object-like because it looks like a data object in code that uses it. They are most commonly

used to give symbolic names to numeric constants. To define preprocessor macros we can use #define. Its format is:

```
#define identifier replacement
```

Syntax to define macros in C

When the preprocessor encounters this directive, it replaces any occurrence of *identifier* in the rest of the code by *replacement*. This *replacement* can be an expression, a statement, a block or simply anything. The preprocessor does not understand C, it simply replaces any occurrence of *identifier* by *replacement*.

```
#define DELAY 20000
```

Wherever, DELAY is found as a token, it is replaced with 20000.

Function-like Macros

Macros can also be de

ned which look like a function call. These are called function-like macros. To de

ne a function-like macro, the same '#defi

ne' directive is used, but with a pair of parentheses immediately after the macro name. For example:

```
#define SUM(a, b, c)  a + b + c
#define SQR(c)  ((c) * (c))
```

Advantages Of Using A Macro

- The speed of the execution of the program is the major advantage of using a macro.
- It saves a lot of time that is spent by the compiler for invoking / calling the functions.
- It reduces the length of the program

Including Files (#include)

It is used to insert the contents of another file into the source code of the current file. There are two slightly different ways to specify a file to be included:

```
#include "filename"
```

Syntax to define macros in C

```
#include "lm4f120h5qr.h"
```

This include directive will include the file named "lm4f120h5qr" at this point in the program. This file will define all the I/O port names for LM4F120 microcontroller.

Structures

Structures provide a way of storing many different values in variables of potentially different types under the same name. This makes it a more modular program, which is easier to modify because its design makes things more compact. Structs are generally useful whenever a lot of data needs to be grouped together–for instance, they can be used to hold records from a database or to store information about contacts in an address book. In the contacts example, a struct could be used that would hold all of the information about a single contact–name, address, phone number, and so forth.

Defining a Structure

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

Syntax to define structure in C

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

Example 2.1.

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use struct keyword to define variables of structure type. Following is the example to explain usage of structure:

Example 2.2.

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */

   strcpy( Book1.title , "C Programming");
   strcpy( Book1.author , "Richard C. Dorf");
   strcpy( Book1.subject , "C Programming Tutorial");
   Book1.book_id = 6495407;


   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Richard C. Dorf
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
```

Type Casting

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the cast operator as follows:

```
(type_name) expression
```

Syntax to typecast a variable

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation:

Example 2.3.

```
#include <stdio.h>

main()
{
   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of mean : 3.400000
```

It should be noted here that the cast operator has precedence over division, so the value of sum is first converted to type double and finally it gets divided by count yielding a double value. Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Unsigned Character	unsigned char	1	0 to 255
Integer	int	2	-32678 to 32767
Short Integer	short int	2	-32678 to 32767
Long integer	long int	4	-2147483648 to 2147483647
Unsigned Integer	unsigned int	2	0 to 65535
Unsigned Short Integer	unsigned short int	2	0 to 65535
Unsigned Long Integer	unsigned long int	4	0 to 4294967295
Float	float	4	1.2E-38 to 3.4E38
Double	double	8	2.2E-308 to 1.8E308
Long Double	long double	10	3.4E-4932 to 1.1E+4932

Type Specifiers

Type Qualifiers

Although the idea of const has been borrowed from C++. Let us get one thing straight: the concepts of const and volatile are completely independent. A common misconception is to imagine that somehow const is the opposite of volatile and vice versa. The table 2.1 provides a list of various type specifiers along with information regarding their memory allotment. Notice that the keywords 'volatile' and 'const' are not present. This is because these are type qualifiers and not type specifiers. We will now look at these two type specifiers in some detail now.

Const Keyword

A data object that is declared with const as a part of its type specification must not be assigned to, in any way, during the run of a program. It is very likely that the definition of the object will contain an initializer (otherwise, since you can't assign to it, how would it ever get a value?), but this is not always the case. For example, if you were accessing a hardware port at a fixed memory address and promised only to read from it, then it would be declared to be const but not initialized.

Volatile Keyword

The reason for having this type qualifier is mainly to do with the problems that are encountered in real-time or embedded systems programming using C. What volatile keyword does is that it tells the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the program itself, and forces every reference to such an object to be a genuine reference. It is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time - without any action being taken by the code the compiler finds nearby.

To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition. For instance, both of these declarations will declare foo to be a volatile integer:

```
volatile int foo;
int volatile foo;
```

Now, it turns out that pointers to volatile variables are very common, especially with memory-mapped I/O registers. Both of these declarations declare pReg to be a pointer to a volatile unsigned 8-bit integer:

```
volatile uint8_t * pReg;
uint8_t volatile * pReg;
```

Proper Use of Volatile

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

- Memory-mapped peripheral registers
- Global variables modified by an interrupt service routine
- Global variables accessed by multiple tasks within a multi-threaded application

Pointers

Pointers are extremely powerful programming tool. They can make some things much easier, help improve your program's efficiency, and even allow you to handle unlimited amounts of

data. For example, using pointers is one way to have a function modify a variable passed to it. It is also possible to use pointers to dynamically allocate memory, which means that you can write programs that can handle nearly unlimited amounts of data on the fly - you don't need to know, when you write the program, how much memory you need. As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

Example 2.4.

```
#include <stdio.h>

int main ()
{
   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

How to declare a pointer?

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. (a) we define a pointer variable (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

Example 2.5.

```
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Double Pointers

Double pointer is a pointer to a pointer. In general, pointers store the address of a variable. Whereas pointer to pointer is a pointer stores the address of another pointer, and this second pointer has the address of the variable. Double pointers are declared as:

```
int    **ip;    /* pointer to a pointer to an integer */
```

Consider the following example.

Example 2.6.

```c
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   int ** ptr;  /* double pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/
   ptr = &ip;  /* store address of ip in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   /* access the value using double pointer */
   printf("Value of **ptr variable: %d\n", **ptr );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
Value of **ptr variable: 20
```

Functions

A function is a group of statements that together perform a task. Every C program has at least one function which is main(), and all the most trivial programs can define additional functions. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. A function is known with various names like a method or a sub-routine or a procedure etc.

The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

Syntax to define functions in C

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

Example 2.7.

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Passing Arguments by Value

By default, arguments in C are passed by value. When arguments are passed by value, a copy of the argument is passed to the function. Therefore, changes made to the formal parameter by the called function have no effect on the corresponding actual parameter. Consider the following code:

Example 2.8.

```
#include <stdio.h>

void foo(int y)
{
    printf("y = %d\n", y);
}

int main()
{
    foo(5); // first call

    int x = 6;
    foo(x); // second call
    foo(x+1); // third call
```

```
    return 0;
}
```

In the first call to foo(), the argument is the literal 5. When foo() is called, variable y is created, and the value of 5 is copied into y. Variable y is then destroyed when foo() ends. In the second call to foo(), the argument is the variable x. x is evaluated to produce the value 6. When foo() is called for the second time, variable y is created again, and the value of 6 is copied into y. Variable y is then destroyed when foo() ends. In the third call to foo(), the argument is the expression x+1. x+1 is evaluated to produce the value 7, which is passed to variable y. Variable y is once again destroyed when foo() ends. Thus, this program prints:

```
y = 5
y = 6
y = 7
```

Because a copy of the argument is passed to the function, the original argument can not be modified by the function. This is shown in the following example:

Example 2.9.

```
#include <stdio.h>

void foo(int y)
{
    printf("y = %d\n", y);

    y = 6;
    printf("y = %d\n", y);
} // y is destroyed here

int main()
{
    int x = 5;
    printf("x = %d\n", x);

    foo(x);
    printf("x = %d\n", x);

    return 0;
}
```

The output of the program is:

```
x = 5
y = 5
y = 6
x = 5
```

At first, x is 5. When foo() is called, the value of x (5) is passed to variable y inside foo(). y is assigned the value of 6, and then destroyed. The value of x is unchanged, even though y was changed.

Arguments passed by value can be variables (e.g., x), literals (e.g., 6) or expressions (e.g., x+1). There are two reasons to call a function by value: side effects and privacy. Unwanted side effects are usually caused by inadvertent changes made to a call by reference parameter. Mostly data is required to be private and, if allowed, only someone calling the function is permitted to change it. However, passing large structures or data by value can take a lot of time and memory to copy, and this can cause a performance penalty, especially if the function is called many times. So, it is better to use a call by value by default and only use call by reference if data changes are expected.

Passing Arguments by Reference

When passing arguments by value, the only way to return a value back to the caller is using the return statement. While this is suitable in many cases, there are a few cases where better options are available. One such case is when writing a function that needs to modify the values of an array (e.g., sorting an array). In this case, it is more efficient and more clear to have the function modify the actual array passed to it, rather than trying to return something back to the caller. One way to allow functions to modify the value of argument is by using pass by reference. In pass by reference, we declare the function parameters as references rather than normal variables:

```
void AddOne(int *y)
{
    *y = *y + 1;
}
```

When the function is called, y will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument! The following example shows this in action:

Example 2.10.

```
#include <stdio.h>

void foo(int *y)
{
    printf("y = %d\n", *y);

    *y = 6;
    printf("y = %d\n", *y);
}

int main()
{
```

```
        int x = 5;
        printf("x = %d\n", x);

        foo(&x);
        printf("x = %d\n", x);

        return 0;
    }
```

This program is the same as the one we used for the pass by value example, except foo's parameter is now a reference instead of a normal variable. When we call foo(&x), *y becomes a reference to x. This example produces the output:

```
x = 5
y = 5
y = 6
x = 6
```

Note that the value of x was changed by the function.

Bitwise Operations in C

The byte is the lowest level at which we can access data; there's no "bit" type, and we can't ask for an individual bit. In fact, we can't even perform operations on a single bit – every bitwise operator will be applied to, at a minimum, an entire byte at a time. This means we will be considering the whole representation of a number whenever we talk about applying a bitwise operator. Table 2.3 summarizes the bitwise operators available in C.

Operator	Description	Example	Result
&	Bitwise AND	0x88 & 0x0F	0x08
$\wedge$	Bitwise XOR	0x0F $\wedge$ 0xFF	0xF0
|	Bitwise OR	0xCC | 0x0F	0xCF
$\ll$	Left Shift	0x01 $\ll$ 4	0x10
$\gg$	Right Shift	0x80 $\gg$ 6	0x02

Bitwise Operators in C

Bit Masking

Bitwise operators treat every bit in a word as a Boolean (two-value) variable, apply a column-wise Boolean operator, and generate a result. Unlike binary math, there is no carry or borrow and every column is independent.

Turning Off Bits

Bit masking is using the bits in one word to "mask off" or select part of the range of bits in another word, using the bitwise Boolean AND operator. The 1 bits in the "mask" select

which bits we want to keep in the other word, and the zero bits in the mask turn all the other corresponding bits to zeroes. In other words, the 1 bits are the "holes" in the mask that let the corresponding bits in the other word flow through to the result.

Turning On Bits

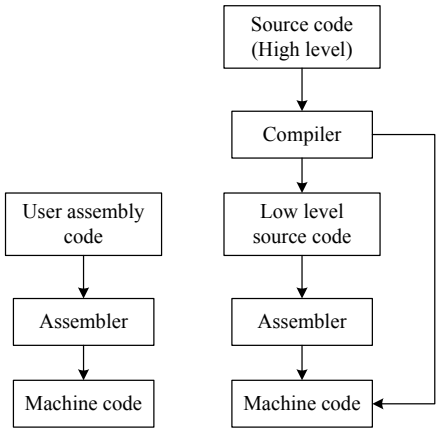
The opposite of masking (turning off bits) is "ORing in" bits, where we use the bitwise Boolean OR to "turn on" one or more bits in a word. We select a value that, when ORed with some other value, "turns on" selected bits and leaves the other bits unchanged.

Toggling Bits

Sometimes it does not really matter what the value is, but it must be made the opposite of what it currently is. This can be achieved using the XOR (Exclusive OR) operation. XOR returns 1 if and only if an odd number of bits are 1. Therefore, if two corresponding bits are 1, the result will be a 0, but if only one of them is 1, the result will be 1. Therefore inversion of the values of bits is done by XORing them with a 1. If the original bit was 1, it returns 1 XOR 1 = 0. If the original bit was 0 it returns 0 XOR 1 = 1. Also note that XOR masking is bit-safe, meaning that it will not affect unmasked bits because Y XOR 0 = Y, just like an OR.

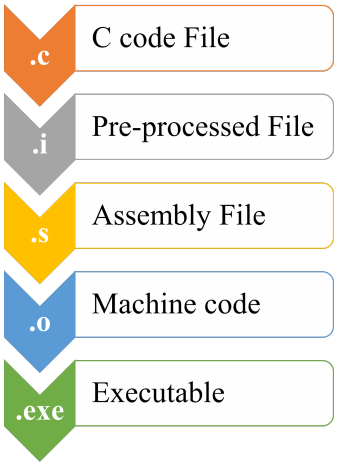
Compilation of C Program

The tool used in the compilation process is called either a compiler or an assembler depending on the type of the user program source file. If the user program is written in assembly language, then we use an assembler to convert it to an object code, which is also called machine code. In this process, an assembly instruction is converted to its equivalent opcode or machine code. Assembly language programs are also called low level programs. If the user program is written in a high level language, we will take the example of a C program, then we use a compiler to convert it to the machine code. Another possibility is that the compiler converts the code to an equivalent assembly program, which is converted to machine code by the assembler. However, in general, the compilers give the machine code as an output.



Conversion into Machine code

When writing a program in C, we also use preprocessor directives. These are resolved before conversion into assembly code. During the preprocessing step, the files or libraries added via *#include* directive are placed in the same file and macros are replaced by their values.



Step-wise Compilation of C code

Stepwise compilation using gcc

To view the stepwise outputs of the compilation process, different options are used along with the gcc command. Table 2.1 lists these options and how to observe the outputs. Using gcc with *-o* option allows us to name the output.

Step	Command	Output	Viewing the output
Pre-processing	gcc -E lab4a.c -o lab4a.i gcc -E lab4b.c -o lab4b.i	lab4a.i lab4b.i	Files can easily be viewed in text editor Notepad++
Compilation	gcc -S lab4a.i lab4b.i	lab4a.s lab4b.s	Files can easily be viewed in text editor Notepad++
Assembling	gcc -c lab4a.s lab4b.s	lab4a.o lab4b.o	In command prompt using: objdump -D lab4a.o objdump -D lab4b.o
Linking	gcc -o out lab4a.o lab4b.o	out.exe	In command prompt: out

Commands for stepwise compilation for files named lab4a.c and lab4b.c

Assembly of the code can also be viewed in the disassembly window in *Debug Mode* in Keil μVision.

Helping Material

The following links can be used as a guide for using gcc.

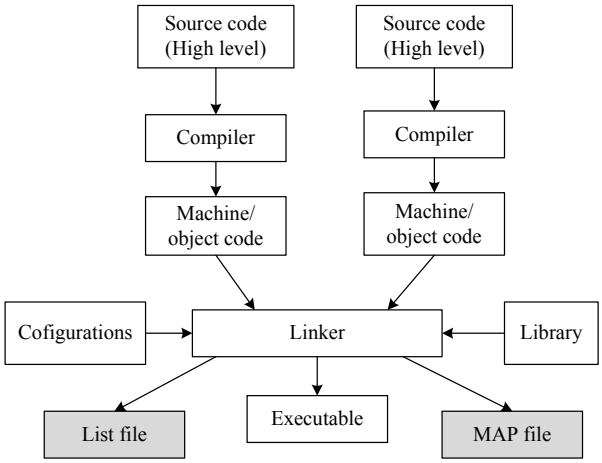
- https://fresh2refresh.com/c-programming/c-environment-setup-using-gcc/
- https://www.edureka.co/blog/how-to-compile-c-program-in-command-prompt/
- https://www.youtube.com/watch?v=vewJP5f6PVw (This video is in Urdu)

Linker

As the linker combines object files, it performs the following tasks:

- Allocates sections into the target system's configured memory.
- Combine object file sections
- Define or redefine global symbols at link time

In addition, some of the functions used in the program are not implemented by the programmer, rather they are provided by a library. For instance, taking the log of a variable will require the use of a math library. It is also the job of a linker to get the desired modules from different libraries and integrate them in the process of constructing the executable. A possible depiction of the functioning of the linker is shown in Figure 2.3. The linker, in addition to building the executable, can generate, optionally, many other files containing useful information for the programmer. For instance, the linker can generate a map file that provides an overall summary of memory usage for code space as well as global data variables. Similarly, the list file can also be generated by the tools, which provides the disassembly of the code that might be helpful to the programmer in debugging. The blocks in Figure 2.3 showing the linker generated MAP and list files are colored to mark that they are optional outputs from the linker.



Creating Executable from Linker

The linker command language controls memory configuration, output section definition, and address binding. Linker scripts that are used to place the vector table, code segment, data segment initializers, and data segments in the appropriate locations in memory. These scripts have different extensions for different toolchains.

- .ld – GNU GCC and Sourcery CodeBench
- .sct – Keil MDK
- .xcl – IAR Embedded Workbench
- .cmd – TI Code Composer Studio

Basic idea of different directives of the linker command file are explained here and here. Two powerful directives in Linker command file, MEMORY and SECTIONS, allow to:

- Allocate sections into specific areas of memory.
- Relocates symbols and sections to assign them to final addresses.
- Resolves undefined external references between input files.

The scatter file for Keil μVision can either be written using the syntax mentioned here or can be configured in *Target Options* like done here. After specifying the scatterfile, build the project again. .sct file and .map files generated can be viewed in text editor like Notepad++.

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x00000000 0x00040000  {    ; load region size_region
  ER_IROM1 0x00000000 0x00040000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x00008000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

The listing option is not available for C files in the evaluation version. Information regarding them can be found here.

Experiment 3

Assembly Language Programming

Every computer, no matter how simple or complex, has a microprocessor that manages the computer's arithmetical, logical and control activities. A computer program is a collection of numbers stored in memory in the form of ones and zeros. CPU reads these numbers one at a time, decodes them and perform the required action. We term these numbers as machine language.

Although machine language instructions make perfect sense to computer but humans cannot comprehend them. A long time ago, someone came up with the idea that computer programs could be written using words instead of numbers and a new language of mnemonics was developed and named as assembly language. An assembly language is a low-level programming language and there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions.

We know that computer cannot interpret assembly language instructions. A program should be developed that performs the task of translating the assembly language instructions to machine language. A computer program that translates the source code written with mnemonics into the executable machine language instructions is called an assembler. The input of an assembler is a source code and its output is an executable object code. Assembly language programs are not portable because assembly language instructions are specific to a particular computer architecture. Similarly, a different assembler is required to make an object code for each platform.

The ARM Architecture

The ARM is a *Reduced Instruction Set Computer*(RISC) with a relatively simple implementation of a load/store architecture, i.e. an architecture where main memory (RAM) access is performed through dedicated load and store instructions, while all computation (sums, products, logical operations, etc.) is performed on values held in registers. ARM supports a small number of addressing modes with all load/store addresses being determined from registers and instruction fields only.

The ARM architecture provides 16 registers, numbered from R0 to R15, of which the last three have special hardware significance.

R13, also known as SP, is the stack pointer, that is it holds the address of the top element of the program stack. It is used automatically in the PUSH and POP instructions to manage storage and recovery of registers in the stack.

```
label
        opcode    operand1,  operand2, …   ; Comment
```

R14, also known as LR is the link register, and holds the return address, that is the address of the first instruction to be executed after returning from the current function.

R15, also known as PC is the program counter, and holds the address of the next instruction.

Assembly Language Syntax

ARM assembler commonly uses following instruction format:

Normally, the first operand is the destination of the operation. The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different.

Label

Label is an optional first field of an assembly statement. Labels are alphanumeric names used to define the starting location of a block of statements. Labels can be subsequently used in our program as an operand of some other instruction. When creating the executable file the assembler will replace the label with the assigned value. Labels must be unique in the executable file because an identical label encountered by the Assembler will generate an error.

ARM assembler has reserved first character of a line for the label field and it should be left blank for the instructions with no labels. In some compilers, labels can be optionally ended with colon(:) but it is not accepted by the ARM assembler as an end of the label field.

Defining appropriate labels makes your program look more legible. Program location can be easily found and remembered using labels. It is easier to use certain functionality of your program in an entirely different code .i.e., your code can be used as a library program. You do not have to figure out memory addresses because it is a tedious task especially when the instruction size in your microcontroller is not fixed.

Opcode (Mnemonics)

Opcode is the second field in assembly language instruction. Assembly language consists of mnemonics, each corresponding to a machine instruction. Using a mnemonic you can decide what operation you want to perform on the operands. Assembler must translate each mnemonic opcode into their binary equivalent.

Operands

Next to the opcode is the operand field which might contain different number of operands. Some of the instructions in Cortex-M3 will have no operand while other might have as many as four operands. The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different. Normally, the first operand is the destination of the operation.

Comments

Comments are messages intended only for human consumption. They have no effect on the translation process and indeed are not acted on by the ARM Assembler. The comment field of an assembly language instruction is also optional. A semicolon signifies that the rest of the line is a comment and is to be ignored by the assembler. If the semicolon is the first non-blank character on the line, the entire line is ignored. If the semicolon follows the operands of an instruction, then only the comment is ignored by the assembler. The purpose of comments is to make the program more comprehensible to the human reader. If you want to modify the program in response to a product update and you haven't used that code before then comments go a long way to helping comprehension.

A First Assembly Program

In this section we will learn to write very simple assembly language program with and without the startup file.

Add the function of startup file.

Assembly Program with Startup File

```
; This is the first ARM Assembly language program. Describe the
; functionality of the program at the top. You can also include
; the info regarding the author and the date

;;; Directives
      PRESERVE8
      THUMB                   ; Marks the THUMB mode of operation

;;;;;;;;;; The user code (program) is placed in CODE AREA

      AREA    |.text|, CODE, READONLY, ALIGN=2

      ENTRY                   ; ENTRY marks the starting point of
                              ; the code execution
      EXPORT __main

__main
```

```
; User Code Starts from the next line

      MOV     R5, #0x1234     ; Let us take some arbitrary numbers
                              ; to start with
      MOV     R3, #0x1234

      ADD     R6, R5, R3      ; Add the values in R5 and R3 and store the
                              ; result in R6
STOP
      B       STOP            ; Endless loop

      END                     ; End of the program, matched with
                              ; ENTRY keyword
```

Assembly Language Program without Startup File

```
;;; Directives
      PRESERVE8
      THUMB                   ; Marks the THUMB mode of operation

Stack   EQU     0x00000100  ; Define stack size to be 256 byes

; Allocate space for the stack.

      AREA    STACK, NOINIT, READWRITE, ALIGN=3

StackMem  SPACE   Stack

; Initialize the two entries of vector table.

      AREA    RESET, DATA, READONLY
      EXPORT  __Vectors

__Vectors
      DCD  StackMem + Stack    ; stack pointer value when stack is empty
      DCD  Reset_Handler       ; reset vector

      ALIGN

;;;;;;;;; The user code (program) is placed in CODE AREA

      AREA    |.text|, CODE, READONLY, ALIGN=2

      ENTRY                   ; ENTRY marks the starting point of
                              ; the code execution
      EXPORT Reset_Handler

Reset_Handler

; User Code Starts from the next line
```

```
        MOV     R5, #0x1234     ; Let us take some arbitrary numbers
                                ; to start with
        MOV     R3, #0x1234

        ADD     R6, R5, R3      ; Add the values in R5 and R3 and store the
                                ; result in R6
STOP
        B       STOP            ; Endless loop

        END                     ; End of the program, matched with
                                ; ENTRY keyword
```

Assembler Directives

Every program to be executed by a computer is a sequence of statements that can be read by the humans. An assembler must differentiate between the statements that will be converted into executable code and that instruct the assembler to perform some specific function.

Assembler directives are commands to the assembler that direct the assembly process. Assembler directives are also called pseudo opcodes or pseudo-operations. They are executed by the assembler at assembly time not by the processor at run time. Machine code is not generated for assembler directives as they are not directly translated to machine language. Some tasks performed by these directives are:

1. Assign the program to certain areas in memory.
2. Define symbols.
3. Designate areas of memory for data storage.
4. Place tables or other fixed data in memory.
5. Allow references to other programs.

ARM assembler supports a large number of assembler directives but in this lab manual we will discuss only some important directives which are required for this lab.

AREA Directive

AREA directive allows the programmer to specify the memory location to store code and data. Depending on the memory configuration of your device, code and data can reside in different areas of memory. A name must be specified for an area directive. There are several optional comma delimited attributes that can be used with AREA directive. Some of them are discussed below.

Example 3.1.

```
AREA    Example, CODE, READONLY    ; An example code section.
        ; user code
```

Attribute	Explanation
CODE	Contains machine instructions. READONLY is the default.
DATA	Contains data, not instructions. READWRITE is the default.
READONLY	Indicates that this area should not be written to.
READWRITE	Indicates that this area may be read from or written to.
NOINIT	Indicates that the data area is initialized to zero. It contains only reservation directives with no initialized values.

ENTRY and END Directives

The first instruction to be executed within an application is marked by the ENTRY directive. Entry point must be specified for every assembly language program. An application can contain only a single entry point and so in a multi-source module application, only a single module will contain an ENTRY directive.

This directive causes the assembler to stop processing the current source file. Every assembly language source module must therefore finish with this directive.

Example 3.2.

```
        AREA    MyCode, CODE, READONLY
        ENTRY       ; Entry point for the application

Start

        ; user code

        END         ; Informs the assembler about the end of a source file
```

EXPORT and IMPORT Directives

A project may contain multiple source files. You may need to use a symbol in a source file that is defined in another source file. In order for a symbol to be found by a different program file, we need to declare that symbol name as a global variable. The EXPORT directive declares a symbol that can be used in different program files. GLOBAL is a synonym for EXPORT.

The IMPORT directive provides the assembler with a name that is not defined in the current assembly.

Example 3.3.

```
AREA    Example, CODE, READONLY
        IMPORT  User_Code       ; Import the function name from
                                ; other source file.
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external
                                ; modules.
```

```
DoAdd   ADD     R0,R0,R1
```

ARM, THUMB Directives

The ARM directive instructs the assembler to interpret subsequent instructions as 32-bit ARM instructions. If necessary, it also inserts up to three bytes of padding to align to the next word boundary. The ARM directive and the CODE32 directive are synonyms.

The THUMB directive instructs the assembler to interpret subsequent instructions as 16-bit Thumb instructions. If necessary, it also inserts a byte of padding to align to the next halfword boundary.

In files that contain a mixture of ARM and Thumb code Use THUMB when changing from ARM state to Thumb state. THUMB must precede any Thumb code. Use ARM when changing from Thumb state to ARM state. ARM must precede any ARM code.

Example 3.4.

```
AREA    ChangeState, CODE, READONLY
        ARM
                                ; This section starts in ARM state
        LDR     r0,=start+1     ; Load the address and set the
                                ; least significant bit
        BX      r0              ; Branch and exchange instruction sets

                                ; Not necessarily in same section
        THUMB                   ; Following instructions are Thumb
start   MOV     r1,#10          ; Thumb instructions
```

ALIGN Directive

Use of ALIGN ensures that your code is correctly aligned. By default, the ALIGN directive aligns the current location within the code to a word (4-byte) boundary. ALIGN 2 can also be used to align on a halfword (2-byte) boundary in Thumb code. As a general rule it is safer to use ALIGN frequently through your code.

PRESERVE8 Directive

The PRESERVE8 directive specifies that the current file preserves 8-byte alignment of the stack. LDRD and STRD instructions (double-word transfers) only work correctly if the address they access is 8-byte aligned. If your code preserves 8-byte alignment of the stack, use PRESERVE8 to inform the linker. The linker ensures that any code that requires 8-byte alignment of the stack is only called, directly or indirectly, by code that preserves 8-byte alignment of the stack.

Data Reservation Directives (DCB, DCD, DCW)

ARM assembler supports different data definition directives to insert constants in assembly code. This directive allows the programmer to enter fixed data into the program memory and treats that data as a permanent part of the program. Different variants of these directives are:

1. DCB (Define Constant Byte) to define constants of byte size.
2. DCW (Define Constant Word) allocates one or more halfwords of memory, aligned on two-byte boundaries.
3. DCD (Define Constant Data) allocates one or more words of memory, aligned on four-byte boundaries.

Example 3.5.

```
data    DCD     0, 0, 0       ; Defines 3 words initialized to zeros
```

SPACE Directive

The SPACE directive reserves a zeroed block of memory. ALIGN directive must be used to align any code following a SPACE directive.

Example 3.6.

```
AREA    MyData, DATA, READWRITE
data1   SPACE   255         ; defines 255 bytes of zeroed store
```

Experiment 4

Digital Input/Output Interfacing and Programming

Objective

The objective of this lab is to give you a hands-on exposure to the programming of I/O, which when executed by the microcontroller (TI TM4C123, an ARM Cortex-M4 based microcontroller) simply blinks LED on the development board.

Introduction to GPIO

A microcontroller communicates with the outside world either by setting the voltage on the pin high (usually 5V) or low (usually 0V) or reading the voltage level of an input pin as being high (1) or low (0). We refer to these pins as general purpose input output (GPIO) pins. Any GPIO pin can be configured through software to be either a digital input or a digital output. GPIO outputs let you translate logical values within your program to voltage values on output pins and voltage outputs help your microcontroller exert control over the system in which it is embedded.

Configuring Peripherals

The fundamental initialization steps required to utilize any of the peripheral are:

1. Enable clocks to the peripheral
2. Configure pins required by the peripheral
3. Configure peripheral hardware

Structure of the Program

The overall structure of this program is illustrated below. The program begins by including the addresses relevant peripheral registers. Main routine follows the initialization steps described above and then enters an infinite loop which toggles an LED and waits for sometime.

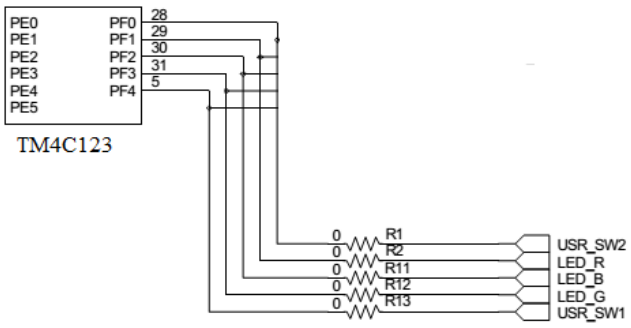
```
#deine register_name  (*(volatile unsigned long *) register_address)

int main(void) {
  //Enable peripherals
  ... (1) ...
  //Configure pins
  ... (2) ...
  while(1) {
    //Turn ON LED
    ... (3)...
    //Delay for a bit
    ... (4) ...
    //Turn OFF LED
    ... (5) ...
  }
}
```

Pseudo code to blink LED

Where is LED?

The TivaC LaunchPad comes with an RGB LED. This LED can be configured for any custom application. Table 3.1 shows how the LED is connected to the pins on the microcontroller. Figure 4.1 shows the physical connection of the onboard LED.



LED Schematic

GPIO pin	Pin Function	USB Device
PF1	GPIO	RGB LED (Red)
PF2	GPIO	RGB LED (Blue)
PF3	GPIO	RGB LED (Green)

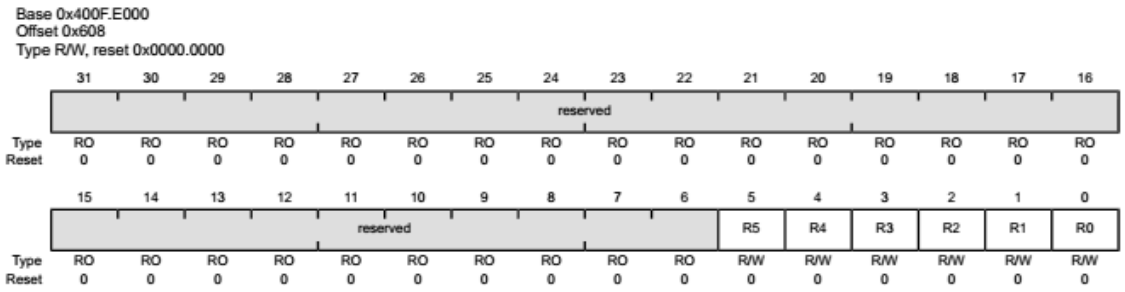
RGB LED Signals

LED Configuration

We will follow the steps stated above to configure the on-board LED.

Enabling the Clock

The RCGCGPIO register provides software the capability to enable and disable GPIO modules in Run mode. When enabled, a module is provided a clock and access to module registers. When disabled, the clock is disabled to save power and accessing a module register generates a bus fault. This register is shown in Figure 4.2. The clock can be enabled for the GPIO port F by asserting the 6th bit of RCGCGPIO register.



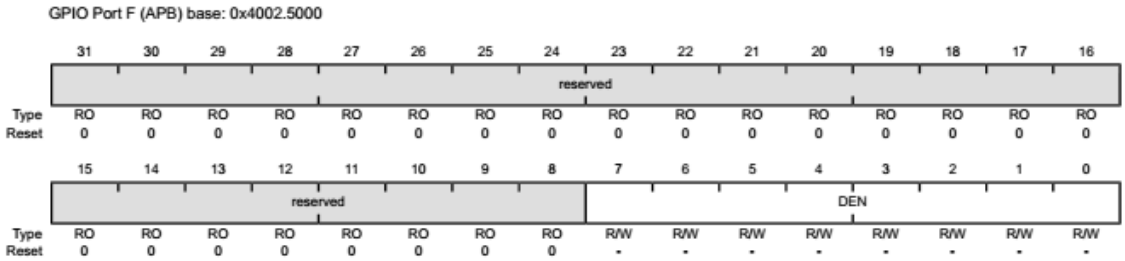
General-Purpose Input/Output Run Mode Clock Gating Control (RCGCGPIO)

Following command can be used to enable clock signal for GPIO port F

```
SYSCTL_RCGCGPIO_R = 0x20;      // (1)
```

Configuring the Pin as Output

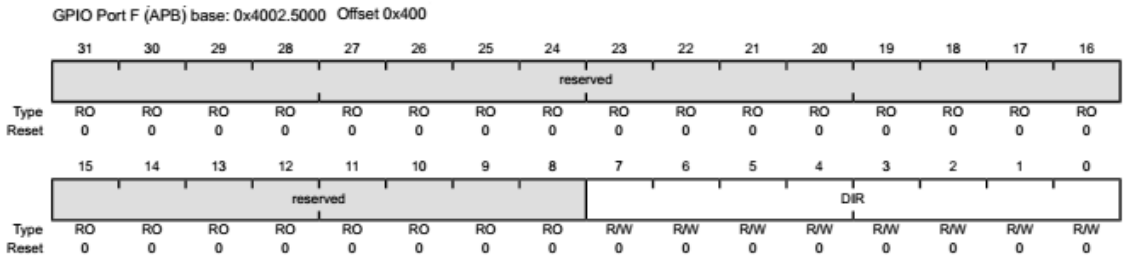
After enabling the clock, it is necessary to configure the required pins. In this case, a single pin (PF3) must be configured as an output. To use the pin as a digital input or output, the corresponding bit in the GPIODEN (Figure 4.3) register must be set and then setting a bit in the GPIODIR (Figure 4.4) register configures the corresponding pin to be an output.



GPIO Digital Enable (GPIODEN)

The commands used to set the corresponding bits in GPIODEN and GPIODIR registers are given as follows

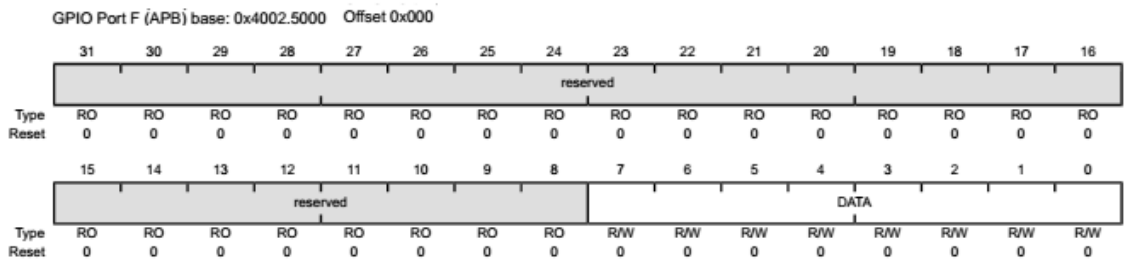
```
GPIO_PORTF_DIR_R = 0x08;      // (2)
GPIO_PORTF_DEN_R = 0x08;
```



GPIO Direction (GPIODIR)

Toggle the LED

After configuring the LED (pin PF3) as an output, now we want to toggle it after regular intervals. LED can be turned ON and OFF by setting and resetting the corresponding bits in the GPIODATA register.



GPIO Data (GPIODATA)

The commands for toggling LED are as follows

```
GPIO_PORTF_DATA_R = 0x08;      // (3)
GPIO_PORTF_DATA_R = 0x00;      // (5)
```

Introducing a Delay

We cannot observe the toggling of the LED because of very high frequency. So, we introduce a delay loop in order to observe the toggle sequence of the LED. The syntax for the loop is shown in the following

```
int counter = 0;
    while(counter < 200000){   // (4)
          ++counter;
    }
```

Source Code

The complete source code for the program is given as follows

Example 4.1.

```
#define SYSCTL_RCGCGPIO_R       (*((volatile unsigned long *)0x400FE608))

#define GPIO_PORTF_DATA_R       (*((volatile unsigned long *)0x400253FC))
#define GPIO_PORTF_DIR_R        (*((volatile unsigned long *)0x40025400))
#define GPIO_PORTF_DEN_R        (*((volatile unsigned long *)0x4002551C))

#define GPIO_PORTF_CLK_EN       0x20
#define GPIO_PORTF_PIN3_EN      0x08
#define LED_ON                  0x08
#define LED_OFF                 ~(0x08)
#define DELAY                   200000


int main(void)
{
    volatile unsigned long ulLoop;

    /* Enable the GPIO port that is used for the on-board LED. */
    SYSCTL_RCGCGPIO_R |= GPIO_PORTF_CLK_EN;

    /* Do a dummy read to insert a few cycles after enabling the
       peripheral. */
    ulLoop = SYSCTL_RCGCGPIO_R;


    /* Enable the GPIO pin for the LED (PF3).  Set the direction as output
        and enable the GPIO pin for digital function.*/

    GPIO_PORTF_DIR_R = GPIO_PORTF_PIN3_EN;
    GPIO_PORTF_DEN_R = GPIO_PORTF_PIN3_EN;

    // Loop forever.
    while(1)
    {

        // Turn on the LED.
        GPIO_PORTF_DATA_R |= LED_ON;

        // Delay for a bit.
        for(ulLoop = 0; ulLoop < DELAY; ulLoop++)
        {
        }

        // Turn off the LED.
        GPIO_PORTF_DATA_R &= LED_OFF;
```

```
            // Delay for a bit.
            for(ulLoop = 0; ulLoop < DELAY; ulLoop++)
            {
            }
        }
    }
```

Exercises

Repeat the above procedure for RED and BLUE LED.

Experiment 5

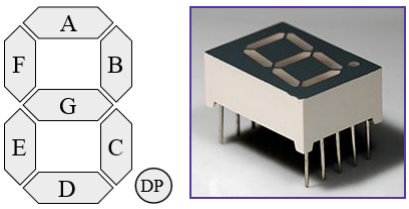
Parallel Interfacing: Interfacing Seven Segment Display

Objective

This lab provides an opportunity to learn about the use of a microcontroller (MCU) and its interfacing to external devices. The lab uses the Tiva-C TM4C123 LaunchPad that has an ARM Cortex-M4F based TM4C123GH6PM MCU in it. This lab uses C language programming to drive a 7-segment display.

7-Segment Display Construction

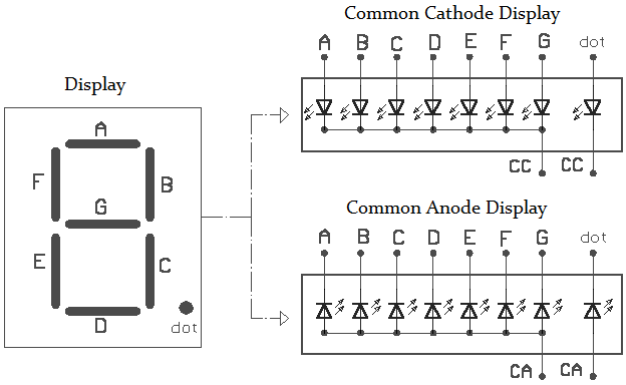
A 7-Segment display is a useful electronic component use to produce numeric, alphabetic and some non-alphabetic symbols using a specific arrangement of LEDs as shown in Figure 5.1.



7 Segment LED Display

A seven segment display consists of seven LEDs arranged in the form of a squarish '8' slightly inclined to the right and a single LED as the dot character. Different characters can be displayed by selectively glowing the required LED segments. Seven segment displays are of two types, common cathode and common anode. In common cathode type, the cathode of all LEDs are tied together to a single terminal which is usually labeled as 'com' and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g & dot. In common anode type, the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins. Both the configurations are shown in Figure 5.2

In this experiment, a GPIO port on the TM4C123GH6PM MCU transmits 8 bits of data. These bits are applied to the seven-segment display to cause it to illuminate the appropriate segments to display the proper number or character. The seven segment used in this experiment is of



Common Anode and Common Cathode Configurations

common-anode type. Segment intensity is dependent on the current flow and should not exceed the limit of the segment.

Digit Drive Pattern

Digit drive pattern of a seven segment LED display is simply the different logic combinations of its terminals. For a certain character, a combination of LED ON and LED OFF is generated to display the character for a short period of time. The pattern is loaded alternately to display other characters. For example, to display the number 2, LEDs a, b, d, e, and g are illuminated. Table 5.3 provides the display pattern for numbers(0-9) and characters A through F.

Characters	DP	G	F	E	D	C	B	A	Hexadecimal
0	1	1	0	0	0	0	0	0	0xC0
1	1	1	1	1	1	0	0	1	0xF9
2	1	0	1	0	0	1	0	0	0xA4
3	1	0	1	1	0	0	0	0	0xB0
4	1	0	0	1	1	0	0	1	0x99
5	1	0	0	1	0	0	1	0	0x92
6	1	0	0	0	0	0	1	0	0x82
7	1	1	1	1	1	0	0	0	0xF8
8	1	0	0	0	0	0	0	0	0x80
9	1	0	0	1	0	0	0	0	0x90
A	1	0	0	0	1	0	0	0	0x88
B	1	0	0	0	0	0	1	1	0x83
C	1	1	0	0	0	1	1	0	0xC6
D	1	0	1	0	0	0	0	1	0xA1
E	1	0	0	0	0	1	1	0	0x86
F	1	0	0	0	1	1	1	0	0xs8E

7 Segment Decoder Table

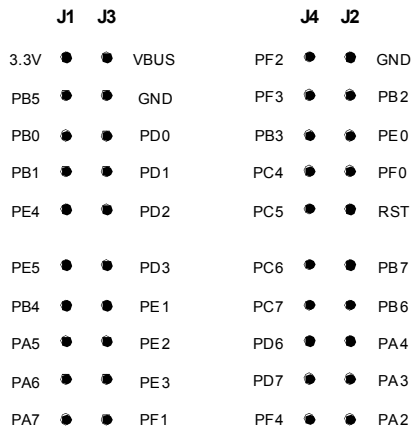
Multiplexing the Seven Segments

To control four 7-segment displays, multiplexing can reduce the number of GPIO pins required. In this setup, the four multiplexed seven-segment displays are turned on one at a time to output the appropriate display. Because of the visual phenomenon known as *persistence of vision*, rapid switching of the seven-segment display can appear as if all four displays are turned on.

Launchpad Interface

Before you build your circuit, you should carefully note the non-sequential pin-out diagram of the LaunchPad shown in Figure 5.4. Although the TM4C123GH6PM MCU has 43 GPIO, only 35 of them are available through the LaunchPad. They are: 6 pins of Port A (PA2-PA7), 8 pins of Port B (PB0-PB7), 4 pins of Port C (PC4-PC7), 6 pins of Port D (PD0-PD3, PD6-PD7), 6 pins of Port E (PE0-PE5) and 5 pins of Port F (PF0-PF4). In addition, there are two ground, one 3.3V, one 5V (VBUS), and one reset pins available on the LaunchPad.

Pins PC0-PC3 are left off as they are used for JTAG debugging. Pins PA0-PA1 are also left off as they are used to create a virtual COM port to connect the LaunchPad to PC. These pins should not be used for regular I/O purpose.

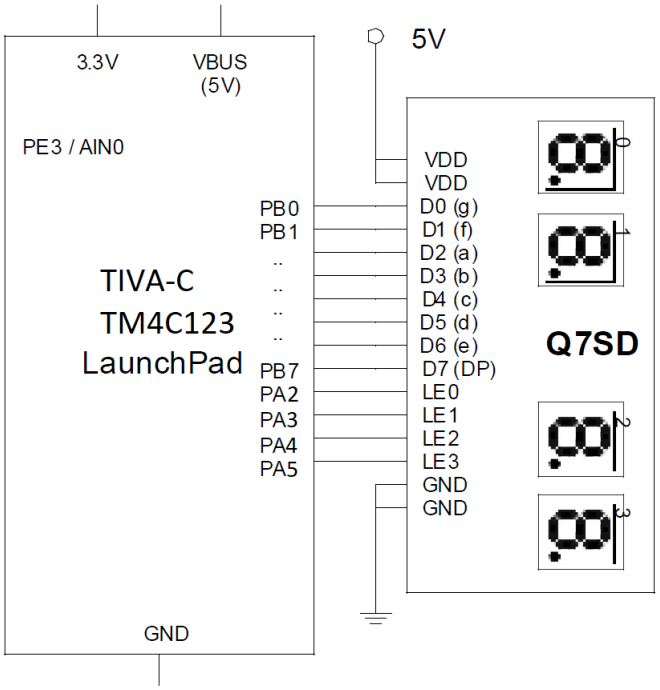


Header Pins on the Launchpad (EK-TM4C123GH6PM)

Display System Using Seven Segments

In this experiment, we will program the LaunchPad and four-digit 7-segment display module to display "UOFS" or "H", "E", "L", "O" depending upon the function selected.

Note carefully the diagram of the display module. Connect the circuit as shown in Figure 5.5. The LaunchPad board is sufficient to supply +5V to Vdd of the 7-segment display module - connect the display power pins directly to VBUS and GND of J3. The experiment uses C language to program the MCU. Write a program to accomplish the task. You will need to open a new project on Keil μVision. Note that display latches are connected to PA2-PA5 and the data pins are connected to PB0-PB7. Using the Latch Enable (LE) pin, your program can update the display digits one at a time.



Connection to Seven Segment Display

Source Code

Source code for displaying UOFS and HELO is given below. Two files are included here. gpio.h includes the register definitions and display_UOFS.c displays UOFS. Create a new project in Keil and add both files in it. Build your code and observe the output.

Example 5.1 (gpio.h).

```
/* ************************************************************
* Macros for Register Addresses and Values
 ************************************************************ */

/* Register for clock */
#define SYSCTL_RCGCGPIO_R    (*((volatile unsigned long *)0x))

/* GPIO Registers for port B */
#define GPIO_PORTB_DATA_R    (*((volatile unsigned long *)0x))
#define GPIO_PORTB_DIR_R     (*((volatile unsigned long *)0x))
#define GPIO_PORTB_DEN_R     (*((volatile unsigned long *)0x))

/* GPIO Registers for port A */
#define GPIO_PORTA_DATA_R    (*((volatile unsigned long *)0x))
#define GPIO_PORTA_DIR_R     (*((volatile unsigned long *)0x))
#define GPIO_PORTA_DEN_R     (*((volatile unsigned long *)0x))

/* Values for enabling seven segments */
#define SEG_1    0x
#define SEG_2    0x
```

```
#define SEG_3   0x
#define SEG_4   0x
#define SEG_OFF 0xFF

/* Function Declarations */
void init_gpio(void);
void display_1(void);
void display_2(void);
void delay(unsigned long value);
```

Example 5.2 (display_UOFS.c).

```
/*****************************************************************
*This program is written for common anode type sevensegment display*
*     Seven segment digits pins:    PA2-PA5*
*     Seven segment data pins:      PB0-PB7*
*     Port B pins:                  76543210*
*                                   pgfedcba*
*****************************************************************/

#include ``gpio.h''

void SystemInit(){}

/*Lookup tables for common anode display */
/*lut for display1*/
const char lut_display1[4]={0x,//U --> 11000001
              0x,//O --> 11000000
              0x,//F --> 10001110
              0x//S --> 10010010
              };

/* lut for display2 */
const char lut_display2[4]={0x,//0x89,H --> 10001001
              0x,//E --> 10000110
              0x,//L --> 11000111
              0x//O --> 11000000
              };

/* lut for segment selection */
const char seg_select[]={0x,//SEG_1
             0x,//SEG_2
             0x,//SEG_3
             0x//SEG_4
            };

/*initialization function for ports */
void init_gpio(void){
```

```
    volatile unsigned long delay_clk;

    /*enable clock for PortA and PortB */
    SYSCTL_RCGCGPIO_R |= 0x;
    // dummy read for delay for clock,must have 3sys clock delay
    delay_clk=SYSCTL_RCGCGPIO_R;

    /* Enable the GPIO pin for PortB pins 0-7 for digital function
    and set the direction as output. */
    GPIO_PORTB_DEN_R |=0x;
    GPIO_PORTB_DIR_R |= 0x;

    /* Enable the GPIO pin for PortA pins 2-5 for digital function.
    and set the direction as output. */
    GPIO_PORTA_DIR_R |=0x;
    GPIO_PORTA_DEN_R |=0x;
}

/* display_1 on seven segments using Macros */
void display_1(void){
    GPIO_PORTA_DATA_R =SEG_OFF;
    GPIO_PORTB_DATA_R =lut_display1[0];
    GPIO_PORTA_DATA_R =SEG_1;
    delay(10000);
    GPIO_PORTA_DATA_R=SEG_OFF;
    GPIO_PORTB_DATA_R=lut_display1[1];
    GPIO_PORTA_DATA_R=SEG_2;
    delay(10000);
    GPIO_PORTA_DATA_R=SEG_OFF;
    GPIO_PORTB_DATA_R=lut_display1[2];
    GPIO_PORTA_DATA_R=SEG_3;
    delay(10000);
    GPIO_PORTA_DATA_R=SEG_OFF;
    GPIO_PORTB_DATA_R=lut_display1[3];
    GPIO_PORTA_DATA_R=SEG_4;
    delay(10000);
}

/* display_2 on seven segments using for loop */
void display_2(void){
    int i;
    for(i=0;i<4;i++)
    {
        GPIO_PORTA_DATA_R=SEG_OFF;
        GPIO_PORTB_DATA_R=lut_display2[i];
        GPIO_PORTA_DATA_R=seg_select[i];
        delay(10000);
    }
}

/* Delay function */
```

```
void delay(unsigned long value){
  unsigned long i ;
  for(i=0;i<value;i++);
}

/* Main function */
int main(void){
  int i;
  init_gpio();
  while(1)
    display_1();
    // display_2();

}
```

Exercises

Program the LaunchPad and four-digit 7-segment display module to alternatively display "HELO" and after some delay,"2021".

Experiment 6

Interrupts and ISR Programming

Interrupt Control in Cortex M4

TM4C123GH6PM implements Nested Vectored Interrupt Controller to handle the interrupts. All the exceptions and interrupts are processed in handler mode. The processor returns to the thread mode when it is finished with all exception processing. The processor automatically saves its current state to the stack when it accepts an interrupt to be serviced. The state is automatically restored from the stack upon the exit from the interrupt service routine (ISR). When an interrupt occurs, state saving and vector fetch are performed in parallel reducing interrupt latency and enabling effcient interrupt entry.
Software can set eight priority levels (0 to 7: a higher level corresponds to a lower priority, i.e., level 0 is the highest interrupt priority) on seven exceptions (such as, reset, software interrupt, hardware interrupt, bus fault, etc.) and 65 interrupts.
When an interrupt occurs and it is accepted by processor core, the corresponding interrupt service routine is executed. Starting address of each interrupt is loaded from the vector table which is an array of word-sized data. Each entry in the vector table points to the starting address of one exception or interrupt handler. The vector table is located at address 0x0000.0000 after reset.
Every exception/interrupt handler is located at the address obtained by multiplying the corresponding exception/interrupt number with 4. For example, if the reset is exception type 1, the address of the reset vector is 1 times 4 (each word is 4 bytes), which equals 0x00000004, and NMI vector (type 2) is located in $2 \times 4 =$ 0x00000008. Consult table 2-8 and 2-9 for the entries of the vector table from article **2.5.2 The Cortex-M4F Processor - Exception Types** of the datasheet.

Enabling an Interrupt

Note: *The register map for NVIC is to be consulted from article* ***3.2 - Cortex-M4 Peripherals - Register Map*** *of the controller datasheet.*

To activate an interrupt source, following two steps must be followed:

1. Enable the source from the corresponding NVIC enable register.
2. Set the priority for that interrupt.

For better understanding, we discuss the example of enabling the interrupt for Port F. Follow the following steps to activate the interrupt for port F.

1. Find the interrupt number (i.e., bit position in interrupt register) from Table 2-9 (2nd column) corresponding to GPIO_Port_F (Figure 6.1).

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
45	29	0x0000.00B4	Flash Memory Control and EEPROM Control
46	30	0x0000.00B8	GPIO Port F
47-48	31-32	-	Reserved
49	33	0x0000.00C4	UART2
50	34	0x0000.00C8	SSI1

Interrupt Number

2. Find the interrupt register that is needed to enable IRQ30 from Table 3-8. It is NVIC_EN0_R. So, it tells you that you need to set bit 30 of NVIC_EN0_R to 1 to enable interrupt on Port F (Figure 6.2).

Nested Vectored Interrupt Controller (NVIC) Registers

0x100	EN0	RW	0x0000.0000	Interrupt 0-31 Set Enable
0x104	EN1	RW	0x0000.0000	Interrupt 32-63 Set Enable
0x108	EN2	RW	0x0000.0000	Interrupt 64-95 Set Enable
0x10C	EN3	RW	0x0000.0000	Interrupt 96-127 Set Enable
0x110	EN4	RW	0x0000.0000	Interrupt 128-138 Set Enable

Interrupt Enable

3. From Table 3-8, find the register needed to set the priority of IRQ 30. It is NVIC_PRI7_R (Figure 6.3).

0x418	PRI6	RW	0x0000.0000	Interrupt 24-27 Priority
0x41C	PRI7	RW	0x0000.0000	Interrupt 28-31 Priority
0x420	PRI8	RW	0x0000.0000	Interrupt 32-35 Priority
0x424	PRI9	RW	0x0000.0000	Interrupt 36-39 Priority

Interrupt Priority

To set a priority value, say 5, you may use the following statement in C:

```
NVIC_PRI7_R = ( NVIC_PRI7_R & 0xFF00FFFF ) | 0x00A00000 ;
```

Configuring GPIO as Interrupt

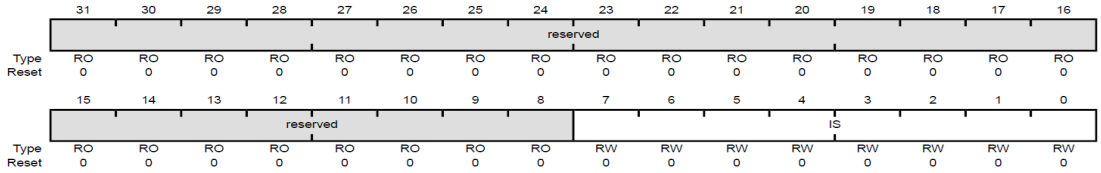
Note: *The register map for GPIO is to be consulted from article* ***10.4 - General-Purpose Input/Outputs (GPIOs) - Register Map*** *of the controller datasheet.*

To configure GPIO pin as interrupt and select the source of the interrupt, its polarity and edge properties following steps must be followed:

1. Disable the interrupts before writing to the control registers.

2. Select whether the source of interrupt is edge-sensitive or level sensitive using GPIO Interrupt Sense register (GPIOIS) Figure 6.4.

GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x404
Type RW, reset 0x0000.0000



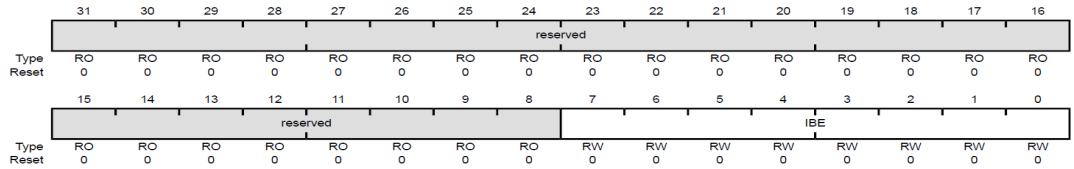
Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IS	RW	0x00	GPIO Interrupt Sense

Value	Description
0	The edge on the corresponding pin is detected (edge-sensitive).
1	The level on the corresponding pin is detected (level-sensitive).

GPIO Interrupt Sense Register

3. To enable interrupts for both edges write the appropriate value in the GPIO Interrupt Both Edges register(GPIOIBE) Figure 6.5.

GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x408
Type RW, reset 0x0000.0000



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IBE	RW	0x00	GPIO Interrupt Both Edges

Value	Description
0	Interrupt generation is controlled by the **GPIO Interrupt Event (GPIOIEV)** register (see page 666).
1	Both edges on the corresponding pin trigger an interrupt.

GPIO Interrupt Sense Register

4. Write the appropriate value in GPIO Interrupt Event register (GPIOIEV) to configure the corresponding pin to detect rising or falling edges depending on the corresponding bit value in the GPIO Interrupt Sense (GPIOIS) register Figure 6.6.

5. Clear the interrupt flag for the corresponding pin by asserting the appropriate bit in the GPIO Interrupt Clear Register (GPIOICR) Figure 6.7.

GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x40C
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								IEV							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IEV	RW	0x00	GPIO Interrupt Event

Value	Description
0	A falling edge or a Low level on the corresponding pin triggers an interrupt.
1	A rising edge or a High level on the corresponding pin triggers an interrupt.

GPIO Interrupt Sense Register

GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x41C
Type W1C, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								IC							
Type	RO	RO	RO	RO	RO	RO	RO	RO	W1C	W1C	W1C	W1C	W1C	W1C	W1C	W1C
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IC	W1C	0x00	GPIO Interrupt Clear

Value	Description
0	The corresponding interrupt is unaffected.
1	The corresponding interrupt is cleared.

GPIO Interrupt Clear Register

6. Enable the interrupts by asserting the corresponding bits in GPIO Interrupt Mask register (GPIOIM) Figure 6.8.

GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x410
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								IME							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IME	RW	0x00	GPIO Interrupt Mask Enable

Value	Description
0	The interrupt from the corresponding pin is masked.
1	The interrupt from the corresponding pin is sent to the interrupt controller.

GPIO Interrupt Mask Register

Source Code

Following code blinks LEDs of different colours on switch press. Understand the code and complete the missing portions. Consult the datasheet for complete understanding of the code.

Example 6.1.

```
//User button connected to PF4
//(turn on different LEDs on falling edge of button press)

#define      SYSCTL_RCGCGPIO_R    (*((volatile unsigned long*)0x400FE608))
//IRQ0to31SetEnableRegister
#define      NVIC_EN0_R           (*((volatile unsigned long*)0xE000E100))
//IRQ28to31PriorityRegister
#define    NVIC_PRI7_R            (*((volatile unsigned long*)0xE000E41C))

#define    GPIO_PORTF_DATA_R      (*((volatile unsigned long*)0x400253FC))
#define    GPIO_PORTF_DIR_R       (*((volatile unsigned long*)0x40025400))
#define    GPIO_PORTF_DEN_R       (*((volatile unsigned long*)0x4002551C))

#define    GPIO_PORTF_PUR_R       (*((volatile unsigned long*)0x40025510))

#define    GPIO_PORTF_IS_R        (*((volatile unsigned long*)0x40025404))
#define    GPIO_PORTF_IBE_R       (*((volatile unsigned long*)0x40025408))
#define    GPIO_PORTF_IEV_R       (*((volatile unsigned long*)0x4002540C))
#define    GPIO_PORTF_IM_R        (*((volatile unsigned long*)0x40025410))
#define    GPIO_PORTF_ICR_R       (*((volatile unsigned long*)0x4002541C))

#define    NVIC_EN0_INT30         ____       //Interrupt30enable
#define    PORTF_CLK_EN           ____       //Clock enable for PortF
#define    LEDs                   ____       //Enable LEDs
#define    SW1                    ____       //Enable user switchSW1
#define    INT_PF4                ____       //Interrupt at PF4


void EnableInterrupts(void);                  //Disable interrupts
void DisableInterrupts(void);                 //Enable interrupts
void Init_INT_GPIO(void);                     //Initialize GPIO and
    Interrupts
void Delay(unsigned long value);              //Implements delay
void WaitForInterrupt(void);

volatile unsigned long i=0;


void Init_INT_GPIO(void){
  volatile unsigned delay_clk;
  SYSCTL_RCGCGPIO_R |= PORTF_CLK_EN;  //EnableclockforPORTF
  delay_clk = SYSCTL_RCGCGPIO_R;     //dummy read to stable the clock
```

```
  //GPIO
  GPIO_PORTF_DEN_R |= (SW1|LEDs); //Enabledigitall/OonPF4,PF3-PF1
  GPIO_PORTF_DIR_R = ____ ;   //Make PF4 input and PF3-PF1 output
  GPIO_PORTF_PUR_R |= SW1 ;   //Enable weak pullup on PF4


  //INTERRUPT
  DisableInterrupts();

  NVIC_EN0_R = NVIC_EN0_INT30;  //Enable interrupt30 in NVIC
  NVIC_PRI7_R = ____;           //Set PF4 priority 5

  GPIO_PORTF_IM_R |= INT_PF4;   //Enable interrupt on PF4
  GPIO_PORTF_IS_R &= ~INT_PF4 ;   //PF4 is edge sensitive
  GPIO_PORTF_IBE_R &= ~INT_PF4 ;  //PF4 is not both edges
  GPIO_PORTF_IEV_R &= ~INT_PF4 ;  //PF4 is falling edge
  GPIO_PORTF_ICR_R |= INT_PF4 ;   //Clear interrupt flag for PF4

  EnableInterrupts();
}


void Delay(unsigned long value){
  unsigned long i=0;
  for(i=0;i<value;i++);
}


void GPIOPortF_Handler(void){
    int j;
    GPIO_PORTF_ICR_R=INT_PF4;
    if(i>=3)
    i=1;
    else
    i++;
    for(j=0;j<2;j++)
    {
        GPIO_PORTF_DATA_R^=1<<i;
        Delay(10000000);
}
}
int main(){
  Init_INT_GPIO();
  while(1)
  {
    WaitForInterrupt();
  }
}
```

Experiment 7

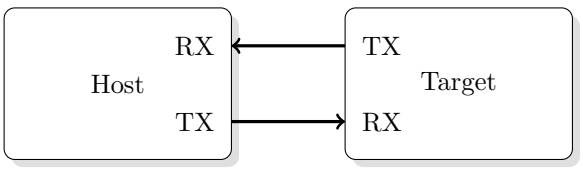
Asynchronous Serial Interfacing (UART)

Objective

The objective of this lab is to utilize the Universal Asynchronous Receiver/Transmitter (UART) to connect the Stellris Launchpad board to the host computer. In the example project, we send characters to the microcontroller unit (MCU) of the board by pressing keys on the keyboard. These characters are sent back (i.e., echoed, looped-back) to the host computer by the MCU and are displayed in a hyperterminal window.

Asynchronous Communication

The most basic method for communication with an embedded processor is asynchronous serial. It is implemented over a symmetric pair of wires connecting two devices (referred as host and target here, though these terms are arbitrary). Whenever the host has data to send to the target, it does so by sending an encoded bit stream over its transmit (TX) wire. This data is received by the target over its receive (RX) wire. The communication is similar in the opposite direction. This simple arrangement is illustrated in Fig. 7.1. This mode of communications is called "asynchronous" because the host and target share no time reference (no clock signal). Instead, temporal properties are encoded in the bit stream by the transmitter and must be decoded by the receiver.

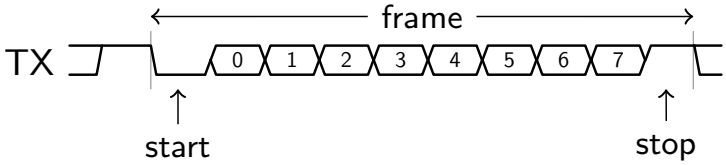


Basic Serial Communication

A commonly used device for encoding and decoding such asynchronous bit streams is a Universal Asynchronous Receiver/Transmitter (UART). UART is a circuit that sends parallel data through a serial line. UARTs are frequently used in conjunction with the RS-232 standard (or specification), which specifies the electrical, mechanical, functional, and procedural characteristics of two data communication equipment.

A UART includes a transmitter and a receiver. The transmitter is essentially a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate. The

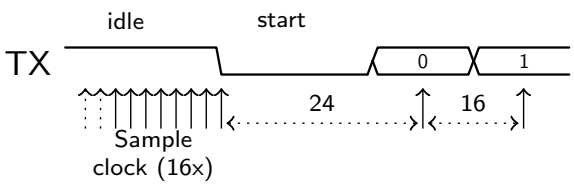
receiver, on the other hand, shifts in data bit by bit and reassembles the data. One of the basic encodings used for asynchronous serial communications is illustrated in Fig. 7.2. Every character is transmitted in a "frame" which begins with a (low) start bit followed by eight data bits and ends with a (high) stop bit. The data bits are encoded as high or low signals for (1) and (0), respectively. Between frames, an idle condition is signaled by transmitting a continuous high signal. Thus, every frame is guaranteed to begin with a high-low transition and to contain at least one low-high transition. Alternatives to this basic frame structure include different numbers of data bits (e.g. 9), a parity bit following the last data bit to enable error detection, and longer stop conditions. Fig. 7.2



Transmission of a byte

There is no clock directly encoded in the signal (in contrast with signaling protocols such as Manchester encoding) – the start transition provides the only temporal information in the data stream. The transmitter and receiver each independently maintain clocks running at (a multiple of) an agreed frequency – commonly called the baud rate. These two clocks are not synchronized and are not guaranteed to be exactly the same frequency, but they must be close enough in frequency (better than 2%) to recover the data. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud-rate (i.e., number of bits per second), the number of data bits and stop bits, and use of parity bit.

To understand how the UART's receiver extracts encoded data, assume it has a clock running at a multiple of the baud rate (e.g., 16x). Starting in the idle state (as shown in Fig. 7.3), the receiver "samples" its RX signal until it detects a high-low transition. Then, it waits 1.5 bit periods (24 clock periods) to sample its RX signal at what it estimates to be the center of data bit 0. The receiver then samples RX at bit-period intervals (16 clock periods) until it has read the remaining 7 data bits and the stop bit. From that point this process is repeated. Successful extraction of the data from a frame requires that, over 10.5 bit periods, the drift of the receiver clock relative to the transmitter clock be less than 0.5 periods in order to correctly detect the stop bit.



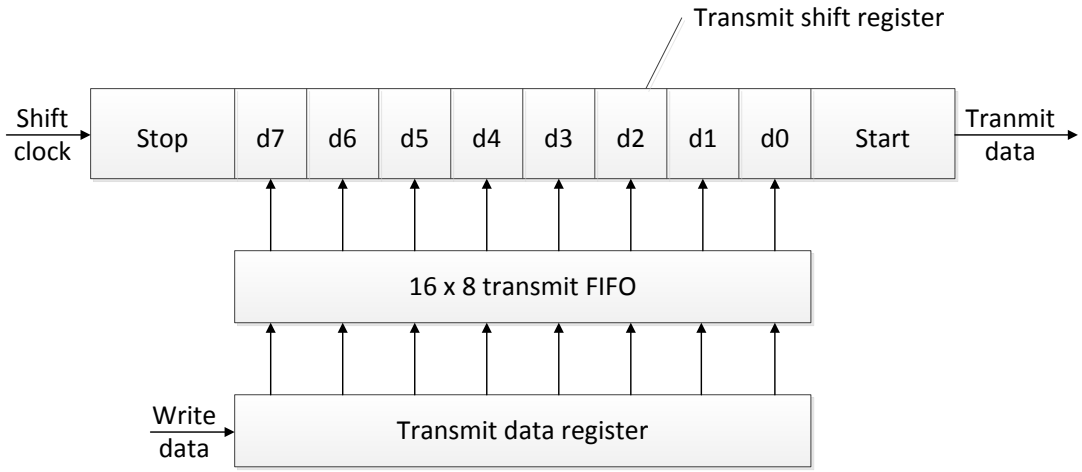
UART Signal Decoding

UART in TM4C123

The simplest form of UART communication is based upon polling the state of the UART device. The UART module in TM4C123 microntroller has 16-element FIFO and 10-bit shift register, which cannot be directly accessed by the programmer. The FIFO and shift register in the transmitter are separate from the FIFO and shift register associated with the receiver. While the data register occupies a single memory word, it is really two separate locations; when the data register is written, the written character is transmitted by the UART. When the data register is read, the character most recently received by the UART is returned. The UART Flag Register(UARTFR) contains a number of flags to determine the current UART state. The important flags are:

TXFE – Transmit FIFO Empty
TXFF – Transmit FIFO Full
RXFE – Receive FIFO Empty
RXFF – Receive FIFO Full

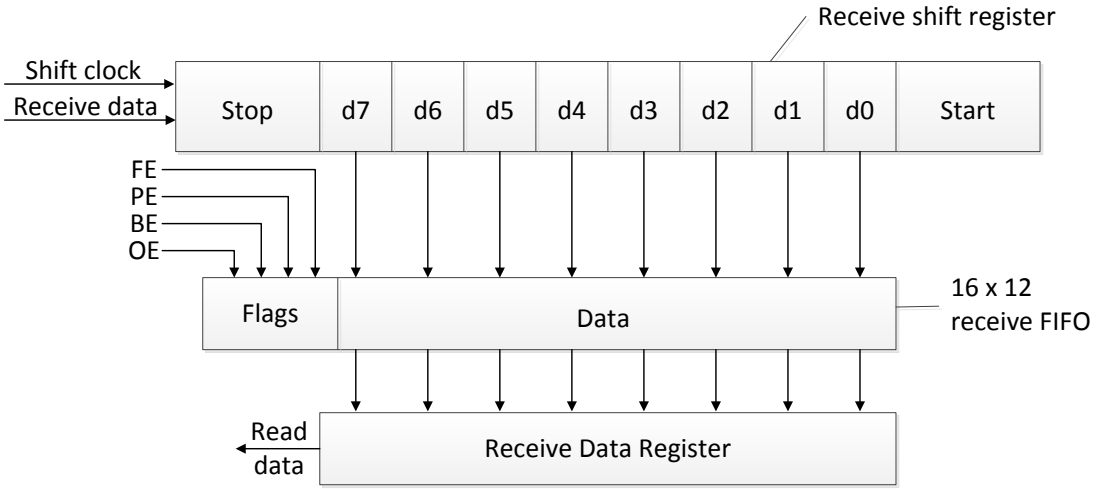
To transmit data using the UART, the application software must make sure that the transmit FIFO is not full (it will wait if TXFF is 1) and then write to the transmit data register(e.g., UART0_DR_R). When a new byte is written to UART0_DR_R, it is put into the transmit FIFO. Byte by byte, the UART gets data from the FIFO and loads into 10-bit shift register which transmits the frame one bit at a time at a specified baud rate. The FIFO guarentees that the data are transmitted in the order they were written.



UART Data Transmission

Receiving frame is a little bit trickier than transmission as we have to synchronize the receive shift register with the data. The receive FIFO empty flag, RXFE, is clear when new input data are in the receive FIFO. When the software reads from UART0_DR_R, data are removed from the FIFO. The other flags associated with the receiver are RXFF (Receive FIFO Full). Four status bits are also associated with each byte of data. These status bits are Overrun Error(OE),

Break Error(BE), Parity Error(PE) and Framing Error(FE). The status of these bits can be checked using UART Receive Status/Error Clear Register(UARTRSR/UARTECR).



UART Data Reception

Initialization and Configuration

TivaC Launchpad has eight UART modules connected to different ports of the microcontroller. In this lab, we will be using UART module 0 connected to PA0 (U0Rx) and PA1 (U0Tx) of GPIO port A. As with all other peripherals, UART must be initialized before it can be used. This initialization includes pin configuration, clock distribution, and device initialization. The steps required to initialize the UART are stated below:

1. The first initialization step is to enable the clock signal to the respective UART module in Run Mode Clock Gating Control UART register (RCGCUART). To enable UART0 module bit 0 of this register should be asserted.

2. Enable the clock for the appropriate GPIO port to which UART is connected. The clock can be enabled using the RCGCGPIO register.

3. To enable the alternate functionality set the appropriate bits of GPIO Alternate Function Select (GPIOAFSEL) register. Now, write the value in GPIO Port Control (GPIOPCTL) register to enable UART signals for the appropriate pins.

Now, we will discuss the steps required to configure the UART module.

1. The first step is calculate the baud-rate divisor (BRD) for setting the required baudrate. The baud-rate divisor is a 22-bit number consisting of a 16-bit integer and a 6-bit fractional part. UART Integer Baud-Rate Divisor (UARTIBRD) register specifies the integer part and UART Fractional Baud-Rate Divisor (UARTFBRD) register specifies the fractional part of baud-rate. Following expression gives the relation of baud-rate divisor and system clock.

$$\boxed{BRD = BRDI + BRDF = UARTSysClk/(ClkDiv * BaudRate)}$$

As the system clock (SysClk) is 16MHz and the desired baud-rate is 115200 bits/sec, then the baud-rate divisor can be calculated as:

$$\boxed{BRD = 16,000,000/(16 * 115200)}$$

So, the value 8 should be written in DIVINT field of UARTIBRD register. Fractional part of baud-rate divisor is calculated in the following equation and the result should be written to the DIVFRAC field of UARTFBRD register.

$$\boxed{UARTFBRD[DIVFRAC] = integer(0.6805 * 64 + 0.5) = 44}$$

2. After calculating the baud rate divisor we must disable the UART by asserting UARTEN bit in UART Control (UARTCTL) register.
3. Integer and fractional values of baud rate divisor, calculated previously, should be written to the appropriate bits of UARTIBRD and UARTFRD registers respectively.
4. Write the desired parameters for the serial communication you want to configure in UART Line Control (LCRH) register. In this experiment, we will be using a word length of 8, one stop bit and enable the FIFOs.
5. To configure the clock source for UART configure the appropriate bit of UART Clock Configuration (UARTCC) register. We will be using system clock for our experiment.
6. After configuring all the parameters, now enable the UART by asserting the UARTEN bit in UART Control (UARTCTL) register.

Note1: *The register map for System Control module (for RCGCUART) is to be consulted from article* ***5.4 - System Control - Register Map*** *of the controller datasheet.*

Note2: *The register map for UART is to be consulted from article* ***14.5 - Universal Asynchronous Receivers/Transmitters (UARTs) - Register Map*** *of the controller datasheet.*

Note5: *The register map for GPIO is to be consulted from article* ***10.4 - General-Purpose Input/Outputs (GPIOs) - Register Map*** *of the controller datasheet.*

Source Code

Template for source code is provided on piazza. It configures UART0 to communicate with the computer and display the echoed characters on hyperterminal (putty).
UART2 can also be configured. It requires to unlock PD7 and extension board for RS232 connector.

Experiment 8

Timers and Time Base Generation

Timers/Counters

Almost every microcontroller comes with one or more (sometimes many more) built-in timer/-counters, and these are extremely useful to the embedded programmer. The term timer/counter itself reflects the fact that the underlying counter hardware can usually be configured to count either regular clock pulses (making it a timer) or irregular event pulses (making it a counter). Sometimes timers are also called "hardware timers" to distinguish them from software timers which are bits of software that perform some timing function.

What is a Timer?

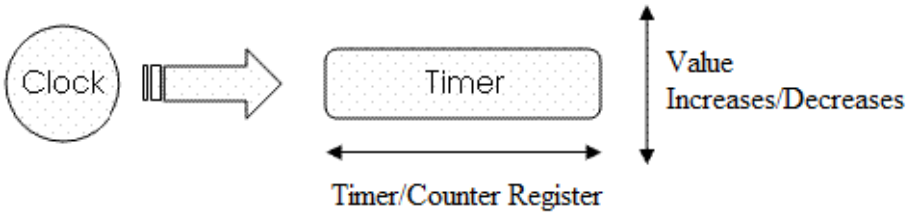
We use timers every day - the simplest one can be found on your wrist. A simple clock will time the seconds, minutes and hours elapsed in a given day - or in the case of a twelve hour clock, since the last half-day. ARM timers do a similar job, measuring a given time interval.

Micro-controllers, such as the TM4C123 utilize hardware timers to generate signals of various frequencies, generate pulse-width-modulated (PWM) outputs, measure input pulses, and trigger events at known frequencies or delays. The TM4C123 parts have several different types of timer peripherals which vary in their configurability. The simplest timers are primarily limited to generating signals of a known frequency or pulses of fixed width. While more sophisticated timers add additional hardware to utilize such a generated frequency to independently generate signals with specific pulse widths or measure such signals.

How a Timer Works?

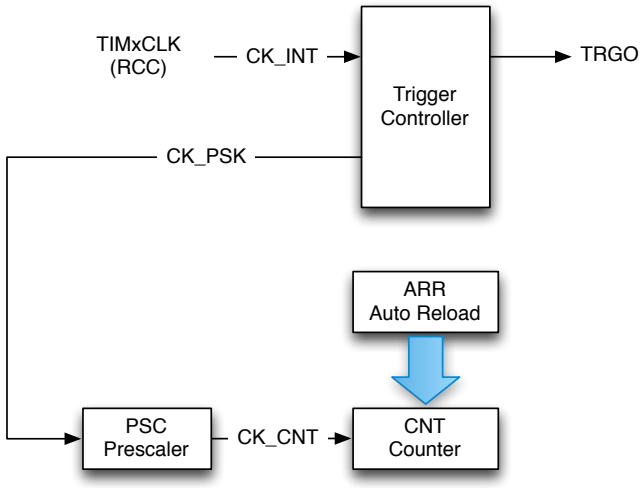
An ARM timer in simplest term is a register. Timers generally have a resolution of 16/32 or 32/64 bits. So a 16 bit timer is 16 bits wide, and is capable of holding value within 0-65536. But this register has a magical property - its value increases/decreases automatically at a predefined rate (supplied by user). This is the timer clock and this operation does not need CPU's intervention.

An example of a basic timer is illustrated in Figure 8.2. This timer has four components – a controller, a prescaler (PSC), an "auto-reload" register (ARR) and a counter (CNT). The function of the prescaler is to divide a reference clock to lower frequency. The TM4C123 timers have 8-bit prescaler registers for 16-bit timers which can divide the reference clock by any value 1..255 and 16-bit prescalar for 32-bit timer which can divide the reference clock by any value



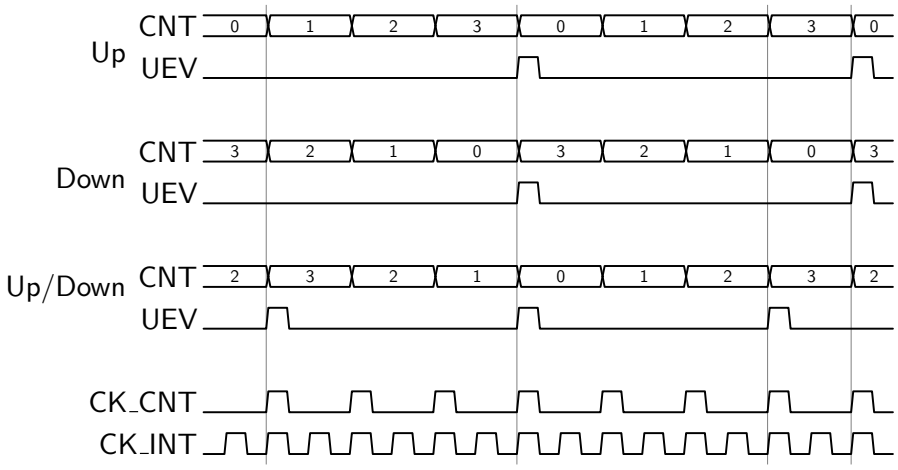
Timer Register

1..65535. The counter register can be configured to count up, down, or up/down and to be reloaded from the auto reload register whenever it wraps around (an "update event") or to stop when it wraps around. The basic timer generates an output event (TGRO) which can be configured to occur on an update event or when the counter is enabled (for example on a GPIO input).



Basic Timer

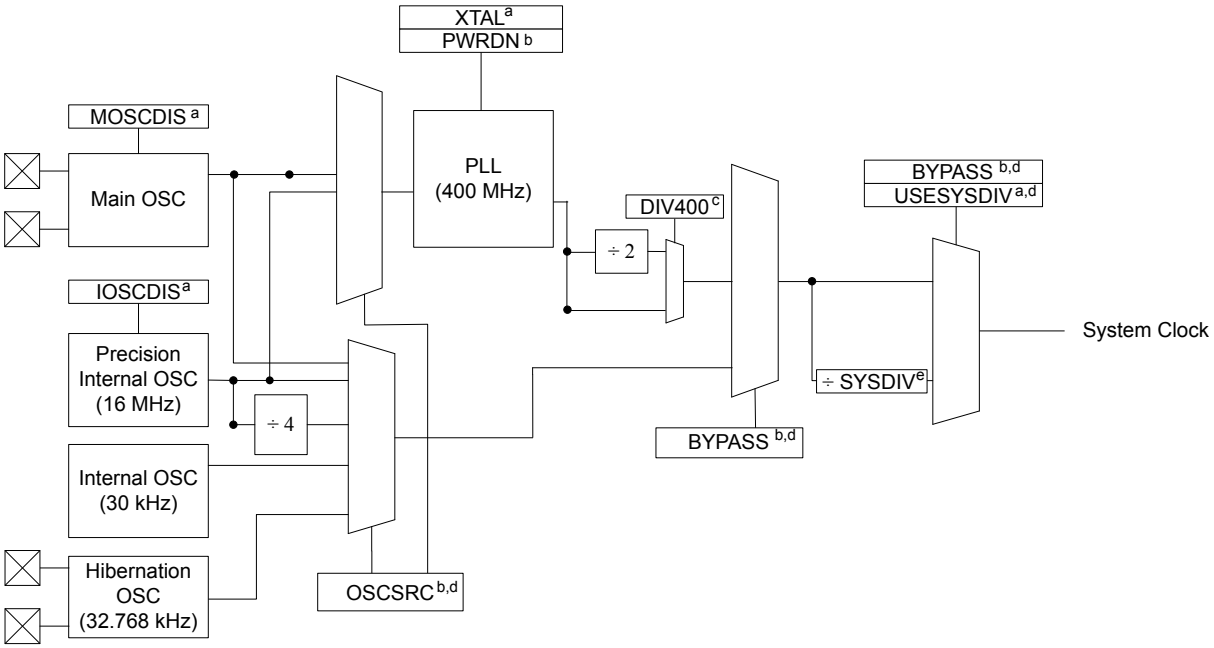
To understand the three counter modes consider Figure 8.3. In these examples, we assume a prescaler of 1 (counter clock is half the internal clock), and a auto reload value of 3. Notice that in "Up" mode, the counter increments from 0 to 3 (ARR) and then is reset to 0. When the reset occurs, an "update event" is generated. This update event may be tied to TRGO, or in more complex timers with capture/compare channels it may have additional effects. Similarly, in "Down" mode, the counter decrements from 3 to 0 and then is reset to 3 (ARR). In Down mode, an update "event" (UEV) is generated when the counter is reset to ARR. Finally, in Up/Down mode, the counter increments to ARR, then decrements to 0, and repeats. A UEV is generated before each reversal with the effect that the period in Up/Down mode is one shorter than in either Up or Down mode.



Counter Modes (ARR=3, PSC=1)

Clock Configuration

The clock system on the Tiva-C Launchpad is extremely flexible. The clock tree of system clock configuration is shown in Figure 8.4



System Clock Tree

The internal system clock (SysClk), is derived from any of the four reference sources plus two others: the output of the main internal PLL and the precision internal oscillator divided by four (4 MHz ± 1%). The frequency of the PLL clock reference must be in the range of 5 MHz to 25 MHz (inclusive). Following table shows how the various clock sources can be used in a system.

Clock Source	Drive PLL?		Used as SysClk?	
Precision Internal Oscillator	Yes	BYPASS = 0, OSCSRC = 0x1	Yes	BYPASS = 1, OSCSRC = 0x1
Precision Internal Oscillator divide by 4 (4 MHz ± 1%)	No	-	Yes	BYPASS = 1, OSCSRC = 0x2
Main Oscillator	Yes	BYPASS = 0, OSCSRC = 0x0	Yes	BYPASS = 1, OSCSRC = 0x0
Low-Frequency Internal Oscillator (LFIOSC)	No	-	Yes	BYPASS = 1, OSCSRC = 0x3
Hibernation Module 32.768-kHz Oscillator	No	-	Yes	BYPASS = 1, OSCSRC2 = 0x7

Clock Source Options

The Run-Mode Clock Configuration (RCC) and Run-Mode Clock Configuration 2 (RCC2) registers provide control for the system clock. The RCC2 register is provided to extend fields that offer additional encodings over the RCC register. When used, the RCC2 register field values are used by the logic over the corresponding field in the RCC register. In particular, RCC2 provides for a larger assortment of clock configuration options. These registers control the following clock functionality:

- Source of clocks in sleep and deep-sleep modes
- System clock derived from PLL or other clock source
- Enabling/disabling of oscillators and PLL
- Clock divisors
- Crystal input selection

To configure RCC and RCC2 clock configuration registers consult the data sheet of TM4C123.

Timers in TM4C123

The Tiva-C General-Purpose Timer Module (GPTM) in TM4C123 contains six 16/32-bit GPTM blocks and six 32/64-bit Wide GPTM blocks. Each 16/32-bit GPTM block provides two 16-bit timers/counters(referred to as Timer A and Timer B) that can be configured to operate independently as timers or event counters. The register map of the GPTM is given in the data sheet. Some key registers are described below:

SYSCTL_RCGCTIMER_R - enable clock for GPTM (enable clock for Timer0)

TIMER0_CTL_R - control Timer0 module

TIMER0_CFG_R - control global operation of Timer0 module (use 32-bit mode)

TIMER0_TAMR_R - control the mode for Timer0

TIMER0_TAPR_R - set the prescalar for Timer0

TIMER0_TAILR_R - load the starting count value into the timer when counting down and set the upper bound for the timeout event when counting up

How to Configure a Timer for Periodic Interrupts

The GPTM is configured for Periodic mode by the following sequence:

1. Enable the General Purpose Timer Module(GPTM) by asserting the appropriate TIMERn bit in the RCGCTIMER register.
2. Ensure the timer is disabled (the TnEN bit in the GPTMCTL register is cleared) before making any changes.
3. Write the GPTM Configuration Register (GPTMCFG) with a value of 0x04 to configure in 16-bit mode.
4. Write a value of 0x2 in the TnMR field of the GPTM Timer n Mode Register (GPTMTnMR) to configure it in periodic mode
5. Load the period value into the GPTM Timer n Interval Load Register (GPTMTnILR)
6. Load the prescale value into the GPTM Timer n Prescale Register (GPTMTnPR).
7. Assert Timer A Time-Out Clear Interrupt (TATOCINT) bit of GPTM Interrupt Clear Register(GPTMICR) to clear the time-out flag.
8. Set Timer A Time-Out Interrupt Mask (TATOIM) bit of GPTM Interrupt Mask Register(GPTMIMR) to enable the time-out interrupt.
9. Set the priority in the correct NVIC Priority Register.
10. Enable the correct interrupt in the correct NVIC Priority Register
11. Set the TnEN bit in the GPTMCTL register to enable the timer and start counting.

Note1: *The register map for System Control module (for RCGCTIMER) is to be consulted from article* ***5.4 - System Control - Register Map*** *of the controller datasheet.*

Note2: *The register map for Timers is to be consulted from article* ***11.5 - General-Purpose Timers - Register Map*** *of the controller datasheet.*

Note3: *Consult table 2-8 and 2-9 for the entries of the vector table from article* ***2.5.2 The Cortex-M4F Processor - Exception Types*** *of the datasheet.*

Note4: *The register map for NVIC is to be consulted from article* ***3.2 - Cortex-M4 Peripherals - Register Map*** *of the controller datasheet.*

Note5: *The register map for GPIO is to be consulted from article* ***10.4 - General-Purpose Input/Outputs (GPIOs) - Register Map*** *of the controller datasheet.*

Source Code

Complete source code for generating the periodic interrupts for Timer 0A (configured in 16-bits). This program toggles the state of PF1 with 1 Hz frequency.

Example 8.1 (**timer.h**).

```
//GPIO registers
#define   SYSCTL_RCGCGPIO_R      (*((volatile unsigned long*)0x400FE608))
#define   GPIO_PORTF_DATA_R      (*((volatile unsigned long*)0x400253FC))
#define   GPIO_PORTF_DIR_R       (*((volatile unsigned long*)0x40025400))
#define   GPIO_PORTF_DEN_R       (*((volatile unsigned long*)0x4002551C))

//Timer registers
#define   SYSCTL_RCGCTIMER_R     (*((volatile unsigned long*)0x400FE604))
#define   TIMER0_CTL_R           (*((volatile unsigned long*)0x4003000C))
#define   TIMER0_CFG_R           (*((volatile unsigned long*)0x40030000))
#define   TIMER0_TAMR_R          (*((volatile unsigned long*)0x40030004))
#define   TIMER0_TAILR_R         (*((volatile unsigned long*)0x40030028))
#define   TIMER0_TAPR_R          (*((volatile unsigned long*)0x40030038))
#define   TIMER0_ICR_R           (*((volatile unsigned long*)0x40030024))
#define   TIMER0_IMR_R           (*((volatile unsigned long*)0x40030018))

//NVIC registers
#define   NVIC_EN0_R             (*((volatile unsigned long*)0xE000E100))
#define   NVIC_PRI4_R            (*((volatile unsigned long*)0xE000E410))

//constant values
#define   TIM0_CLK_EN            0x____      //enableclockforTimer0
#define   TIM0_EN                0x____    //disableTimer0beforesetup
#define   TIM_16_BIT_EN          0x____      //configure16-bittimermode
#define   TIM_TAMR_PERIODIC_EN   0x____      //configureperiodicmode
#define   TIM_FREQ_10usec        ____        //selectprescalaerfor desired
    frequency 100kHz
#define   TIM0_INT_CLR           0x____      //cleartimeoutinterrupt
#define   EN0_INT19              0x____    //enableinterrupt19
#define   PORTF_CLK_EN           0x____      //enableclockforportF
#define   TOGGLE_PF1             0x____      //toggleredled(PF1)
#define   LED_RED                0x____      //configureredled(PF1)

//function headers
void GPIO_Init(void);
void Timer_Init(unsigned long period);
void DisableInterrupts(void);
void EnableInterrupts(void);
void WaitForInterrupt(void);
```

Example 8.2 (Timer0A Periodic Interrupt.c).

```
#include ''timer.h''

void Timer_Init(unsigned long period)
{
  SYSCTL_RCGCTIMER_R |= TIM0_CLK_EN;              //enable clock for
      Timer0
  TIMER0_CTL_R &= ~(TIM0_EN);                     //disable Timer0 before
      setup
  TIMER0_CFG_R |= TIM_16_BIT_EN;                  //configure 16-bit timer
       mode
  TIMER0_TAMR_R |= TIM_TAMR_PERIODIC_EN;          //configure periodic
      mode
  TIMER0_TAILR_R = period;                        //set initial load value
  TIMER0_TAPR_R = TIM_FREQ_10usec;                //set prescalaer for
      desired frequency 100kHz
  TIMER0_ICR_R = TIM0_INT_CLR;                    //clear timeout
      interrupt
  TIMER0_IMR_R |= TIM0_EN;                        //enable interrupt mask
      for Timer_0A
  DisableInterrupts();

  //Set priority for interrupt
  NVIC_PRI4_R = (NVIC_PRI4_R & 0x00FFFFFF)|0x40000000;
  NVIC_EN0_R |= EN0_INT19;                        //enable interrupt 19
  TIMER0_CTL_R |= TIM0_EN;                        //enableTimer_0A
  EnableInterrupts();
}

void GPIO_Init()
{
  SYSCTL_RCGCGPIO_R |= PORTF_CLK_EN;
  GPIO_PORTF_DIR_R |= LED_RED;
  GPIO_PORTF_DEN_R |= LED_RED;
}

void Timer0A_Handler(void)
{
  TIMER0_ICR_R = TIM0_INT_CLR;
  GPIO_PORTF_DATA_R ^= TOGGLE_PF1;
}

int main(void)
{

  Timer_Init(50000);          //generate asquare wave for 2Hz
  GPIO_Init();                //initialize PF1 as digital output
  while(1);
}
```

Experiment 9

PWM Generation Using Timers

The Tiva-C General-Purpose Timer Module (GPTM) in TM4C123 contains six 16/32-bit GPTM blocks and six 32/64-bit Wide GPTM blocks. Each 16/32-bit GPTM block provides two 16-bit timers/counters(referred to as Timer A and Timer B) that can be configured to operate independently as timers or event counters. When a timer is used as an output device, its main purpose is to generate signals with desired attributes. For instance, we can use timer to generate periodic signal (square wave shape) of certain frequency as well as to generate pulses of varying width. It can also be used to generate time delays.

Pulse Width Modulation

Pulse width modulation (PWM) is a technique to generate variable width pulses, while maintaining a constant signal frequency. PWM signals are widely used in many control applications. Below are some common uses of PWM signals.

1. Motor speed control
2. Power converters
3. Brightness control for LEDs
4. Encoded data transmission

The time period, t_p and the pulse width t_w are the two parameters that must be speci

ed to generate the desired PWM signal, such that tw ¡ tp. An important attribute of PWM signal is its duty cycle, d, which is defined as a percentage value using the following expression

How to Configure a Timer for PWM

A timer is configured to PWM mode using the following sequence:

1. Configure the respective GPIO pin for alternate functionality.
2. Ensure the timer is disabled (the TnEN bit is cleared) before making any changes.
3. Write the GPTM Configuration (GPTMCFG) register with a value of 0x0000.0004.
4. In the GPTM Timer Mode (GPTMTnMR) register, set the TnAMS bit to 0x1, the TnCMR bit to 0x0, and the TnMR field to 0x2.
5. Configure the output state of the PWM signal (whether or not it is inverted) in the TnPWML field of the GPTM Control (GPTMCTL) register.
6. If a prescaler is to be used, write the prescale value to the GPTM Timer n Prescale Register (GPTMTnPR).

7. If PWM interrupts are used, configure the interrupt condition in the TnEVENT field in the GPTMCTL register and enable the interrupts by setting the TnPWMIE bit in the GPTMTnMR register. Note that edge detect interrupt behavior is reversed when the PWM output is inverted
8. Load the timer start value into the GPTM Timer n Interval Load (GPTMTnILR) register.
9. Load the GPTM Timer n Match (GPTMTnMATCHR) register with the match value.
10. Set the TnEN bit in the GPTM Control (GPTMCTL) register to enable the timer and begin generation of the output PWM signal.

In PWM Time mode, the timer continues running after the PWM signal has been generated. The PWM period can be adjusted at any time by writing the GPTMTnILR register, and the change takes effect at the next cycle after the write. ***Note1:*** *The register map for System Control module (for RCGCTIMER) is to be consulted from article* ***5.4 - System Control - Register Map*** *of the controller datasheet.*

Note2: *The register map for Timers is to be consulted from article* ***11.5 - General-Purpose Timers - Register Map*** *of the controller datasheet.*

Note3: *Consult table 2-8 and 2-9 for the entries of the vector table from article* ***2.5.2 The Cortex-M4F Processor - Exception Types*** *of the datasheet.*

Note4: *The register map for NVIC is to be consulted from article* ***3.2 - Cortex-M4 Peripherals - Register Map*** *of the controller datasheet.*

Note5: *The register map for GPIO is to be consulted from article* ***10.4 - General-Purpose Input/Outputs (GPIOs) - Register Map*** *of the controller datasheet.*

Source Code

Template for source code is provided on piazza. It configures Timer 1A for 1kHz PWM signal.

Exercise

Configure PWM for PF1 or PF3.

Experiment 10

Analog Interfacing

Lab Objective

In this lab, we will learn to configure analog-to-digital converter (ADC) and sample sequencers available in TM4C123 based TIVA-C Launchpad. We will measure the temperature using on-chip temperature sensor and display the result on a 7-segment display.

ADC in TM4C123

TM4C123G MCU contains two identical 12-bit ADC modules with a capability of 12 shared input channels and hardware averaging upto 64 samples. Each module is controlled by a number of registers and offers a variety of options. Both the modules have four sequencers. In this lab, we will use sequencer 3 since it captures only one sample per trigger and stores the sample into the corresponding FIFO. The sampling rate can be varied from 125 KSPS to 1 MSPS.

ADC Initialization

In this experiment, we will be using ADC0 and sample sequencer 3 to sample the data of on-chip temperature sensor and display it on seven segment module. A brief description of initialization and configuration steps for the ADC module is given below.

- As with all peripherals, first initialization step is to enble the clock signal for the appropriate GPIO pin and ADC module using RCGCGPIO and RCGCADC registers respectively. In this experiment, we are using on-chip temperature sensor so, we don't need to enable the clock signal for any analog input (AIN) pin.
- Enable the alternate functionality by asserting the appropriate bits in GPIOAFSEL register for the analog signal. Also configure the pin as an input by clearing the respective bits in GPIODEN register.
- Disable the analog isolation circuit by asserting the appropriate bits for the respective analog input (AIN) pins in GPIOAMSEL register.

In this experiment we can skip the above mentioned steps to configure a GPIO pin as an analog input because we will be using on-chip temperature sensor as analog input. Now, we discuss the steps to configure sample sequencer.

- Set the sampling rate for the ADC by writing appropriate value to the ADCPC register.

- Disable the sample sequencer by asserting the corresponding bit in ADCACTSS register. We disable all the peripherals before configuration to avoid any erroneous execution which may result in unintended results.
- Select the trigger event from ADCEMUX register. ADC in TM4C123G MCU provides four different options of software, analog comparator, timer or GPIO which are completely programmable for each sample.
- Configure the corresponding input source in the ADCSSMUXn register for each sample in sample sequence. For example, to configure AIN3 as an input source for the first sample in sample sequencer 3 we should write 0x03 to the lower nibble of ADCSSMUX3 register.
- Write the corresponding nibble in ADCSSCTL register for con

  guring sample control bits for each sample in sample sequencer. In the last sample, END bit must be asserted to mark the end of analog sample otherwise ADC may exhibit unpredictable behaviour. To use internal temperature sensor ensure that TS bit is asserted in the corresponding nibble.
- If interrupts are to be used, corresponding MASK bit should be asserted in ADCIM register. In this experiment, we will not use interrupts so 0 should be written to this bit.
- After configuring the sample sequencer with the required parameters now activate the sample sequencer by writing 1 to corresponding bit of the sample sequencer.

After configuration, ADC is triggered by writing 0x0008 to ADC0PSSI register. When the conversion is complete, bit 3 of ADC0RIS register is set to 1 automatically by the hardware. So, by polling this bit, we will know when the conversion is done and the result is ready. Then, the 12-bit ADC result is read from ADC0 SSFIFO3 register.

Note1: *The register map for System Control module (for RCGCADC, RCGCGPIO) is to be consulted from article* ***5.4 - System Control - Register Map*** *of the controller datasheet.*

Note2: *The register map for ADC is to be consulted from article* ***13.5 - Analog-to-Digital Converter (ADC) - Register Map*** *of the controller datasheet.*

Note5: *The register map for GPIO is to be consulted from article* ***10.4 - General-Purpose Input/Outputs (GPIOs) - Register Map*** *of the controller datasheet.*

Source Code

Template for source code is provided on piazza. ADC0 and sample sequencer 3 to sample the data of on-chip temperature sensor.
Example 12.4 of the textbook configures ADC0 to read temperature from temperature sensor (LM35) on the extension board.