EL419- EMBEDDED SYSTEMS
LAB MANUAL

DEPARTMENT OF ELECTRICAL ENGINEERING,

FAST-NU, LAHORE

Created by: Shazia Ahmed

Date: May 2019

Last Updated by: Tooba Javed

Date: November 2019

Approved by the HoD: Dr. Amjad Hussain

Date: November 2019

Table of Contents

List of Equipment

Sr. No.	Description
1	TM4C123 LaunchPad
2	Booster Pack
3	
4	

EXPERIMENT 1

C FOR EMBEDDED SYSTEMS

OBJECTIVE:

To learn features of C language for memory constraint embedded systems

EQUIPMENT:

Keil uVision 4

BACKGROUND:

Embedded systems are commonly programmed using C, assembly and BASIC. C gives embedded programmers great hardware control without sacrificing the benefits of high-level languages. C used for embedded systems is slightly different as compared to C used for general purpose programming. Programs written 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.

C Data Types

Data type	Size	Range Min	Range Max
char	1 byte	-128	127
unsigned char	1 byte	0	255
short int	2 bytes	-32,768	32,767
unsigned short int	2 bytes	0	65,535
int	4 bytes	-2,147,483,648	2,147,483,647
unsigned int	4 bytes	0	4,294,967,295
long	4 bytes	-2,147,483,648	2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295	
long long	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	8 bytes	0	18,446,744,073,709,551,615

A 32-bit processor such as ARM architecture reads the memory with a minimum of 32 bits on the 4-byte boundary.

Bit wise operations in C

A	B	AND (A & B)	OR (A | B)	EX-OR (A^B)	Invert ~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

Applications of bit wise operations

- Bit masking
- Toggling bits
- Turning off bits
- Turning on bits

Bit-wise shift operation in C

Operation	Symbol	Format of Shift Operation
Shift Right	>>	data >> number of bit-positions to be shifted right
Shift Left	<<	data << number of bit-positions to be shifted left

Using shift operator to generate mask

One way to ease the generation of the mask is to use the left shift operator. To generate a mask with bit n set to 1, use the expression: 1 << n

If more bits are to be set in the mask, they can be "or" together. To generate a mask with bit n and bit m set to 1, use the expression:

```
(1 << n) | (1 << m)
```

Example:

register |= (1 << 6) | (1 << 1);

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
```

Example:

```
int s=9;double m;
m = (double) s/count;
```

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 (n).

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:

I. 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. It is beneficial in increasing the speed of execution and saves a lot of time that is spend by the compiler for invoking/calling functions. It also reduces the length of the program. 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. Its format is: `#define identifier replacement`

  Example: `#define delay 20000`
  Wherever, DELAY is found as a token, it is replaced with 20000.

- **Function-like Macros:** Macros can also be defined which look like a function call. These are called function-like macros. To define a function-like macro, the same `#define' directive is used, but with a pair of parentheses immediately after the macro name.

  Example: `#define sum(a,b,c) a+b+c`

II. Including Files (#include)

It is used to insert the contents of another file into the source code of the current file.
Example: #include “tm4c123gh6pm.h”

Type Qualifiers

Type qualifiers define how variables can be accessed or modified. There are two forms of type qualifier: const and volatile.

- **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. Const is supposed to given a value at the time of initialization but not always. 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. 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

LAB TASKS:

You are provided with Keil project files. For this lab, use those files and do not create a new project. Just edit the .c file as required. For each of the questions below, check status of updated variable in watch window in debug mode.

1. Declare an unsigned char variable **x** and assign a random value to it. Now write C statements for the following operations:
    a. Create another variable with upper four bits al zeros and lower four bits equal to lower nibble of x.
    b. Toggle bit #5
    c. Set bit # 5 and clear 6^{th} bit

2. Write a C statement to declare a 32 bit unsigned integer using C99 data types and set its bits 30-28 to value "6" using compound operators. Try to do this task in two steps only.
3. Write output of following code
    a. 0x2F & 0Xf0
    b. 0x27<<4
    c. x=0x56; x=x&~(3<<2);
    d. x=0x78; x=(x&~(7<<4)) | (5<<4);

POST LAB

1. In 8051 microcontroller, P2 has address of 0A0H. We want to access it by defining a new label for it. So we wrote following statement where p_reg is a pointer to this address.

```
uint8_t *p_reg = (uint8_t *) 0xA0;
```

If P2 is connected with keypad and its value is subject to change externally, what is the bug in the above C statement?

2. There are two files in the Keil project you just used in this lab. One of those is a startup file of TM4C. Open this file and go through it. You will notice usage of “export”. What is this keyword used for?

EXPERIMENT 2

INTRODUCTION TO TIVA C SERIES LAUNCHPAD

OBJECTIVE:

- Familiarization with Tiva C series LaunchPad
- Installing Drivers and Flash programmer
- Installing serial utility and connecting board with it.

EQUIPMENT:

- Tiva C Series TM4C LaunchPad Evaluation Board (EK-TM4C123GXL)
- USB Micro-B plug to USB-A plug cable
- Software package that consist of following softwares for target:
  1. TivaWare (www.ti.com/tool/sw-tm4c-drl)
  2. Drivers for Launchpad (http://www.ti.com/tool/stellaris_icdi_drivers)
  3. LMFlash Programmer (www.ti.com/tool/lmflashprogrammer)
  4. Tera Term utility (https://tera-term.en.lo4d.com/windows)

BACKGROUND:

Overview of the LaunchPad:

Tiva C series LaunchPad Evaluation Kit is developed and manufactured by Texas Instruments. Board and its various components are shown in Figure-1 below. This LaunchPad includes:

1. Tiva C Series TM4C LaunchPad Evaluation Board (EK-TM4C123GXL)
2. On-board In-Circuit Debug Interface (ICDI)
3. USB Micro-B plug to USB-A plug cable

The launchpad has following main features:

- ARM® Cortex™-M4F 64-pin 80MHz TM4C123GH6PM’
- On-board USB ICDI (In-Circuit Debug Interface)
- Micro AB USB port
- Device/ICDI power switch
- BoosterPack XL pinout also supports legacy BoosterPack pinout
- 2 user pushbuttons (SW2 is connected to the WAKE pin)
- Reset button
- 3 user LEDs (1 tri-color device)
- Current measurement test points
- 16MHz Main Oscillator crystal
- 32kHz Real Time Clock crystal
- 3.3V regulator
- Support for multiple IDEs including Code Composer Studio, ARM Keil, IAR systems etc.

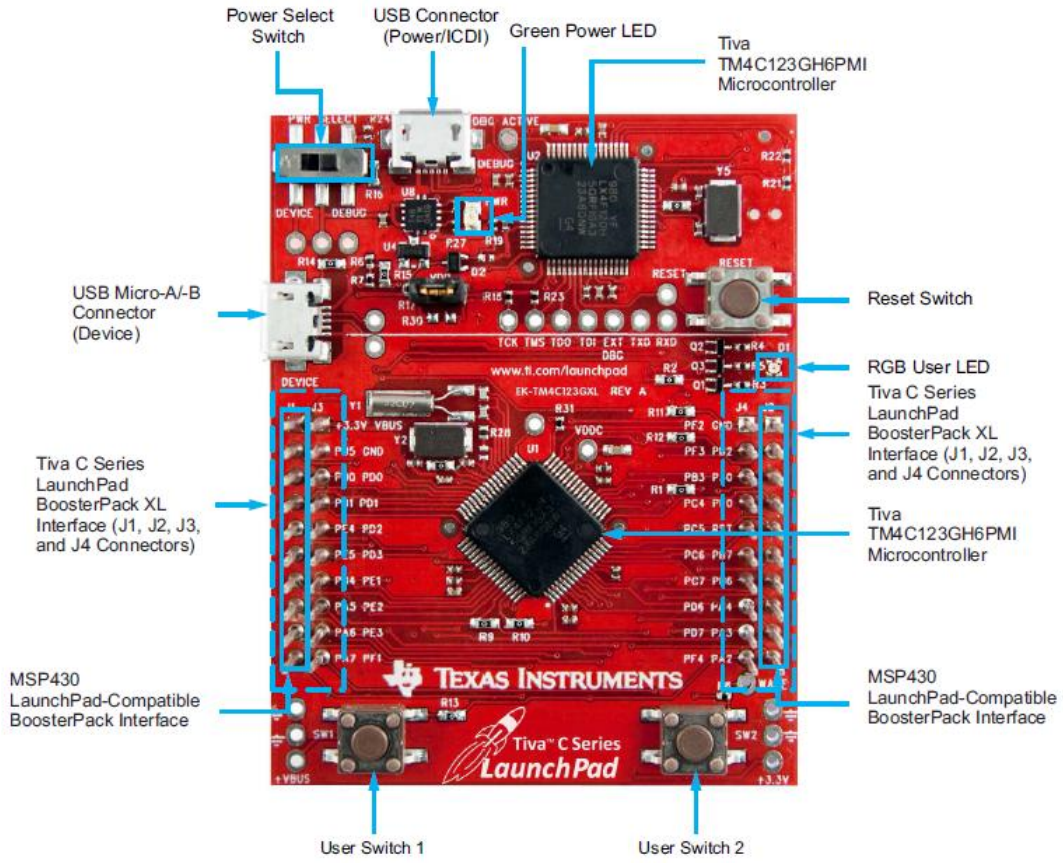


Figure 2-1 TM4C123G LaunchPad Evaluation Board
(Reference: TI User Guide for Tiva C series LaunchPad)

Overview of the microcontroller

The launchpad uses the TM4C123GH6PM microcontroller which has following main features:

1. 32-bit ARM Cortex-M4 CPU with floating point,
2. 256 kBytes of 100,000 write-erase cycles of flash memory,
3. 32KB RAM
4. 2-KB EEPROM
5. On-chip ROM with drivers and boot loaders
6. 2x 12ch 12-bit ADCs (1 MSPS)
7. 16x Motion PWM channels
8. 24x Timer/Capture/Compare/PWMs
9. 3x Analog comparators
10. 4x SPI/SSI, 4x I2C, 8x UART
11. USB Host/Device/OTG
12. 2x CAN
13. Low-power hibernation mode
14. 43x GPIO pins

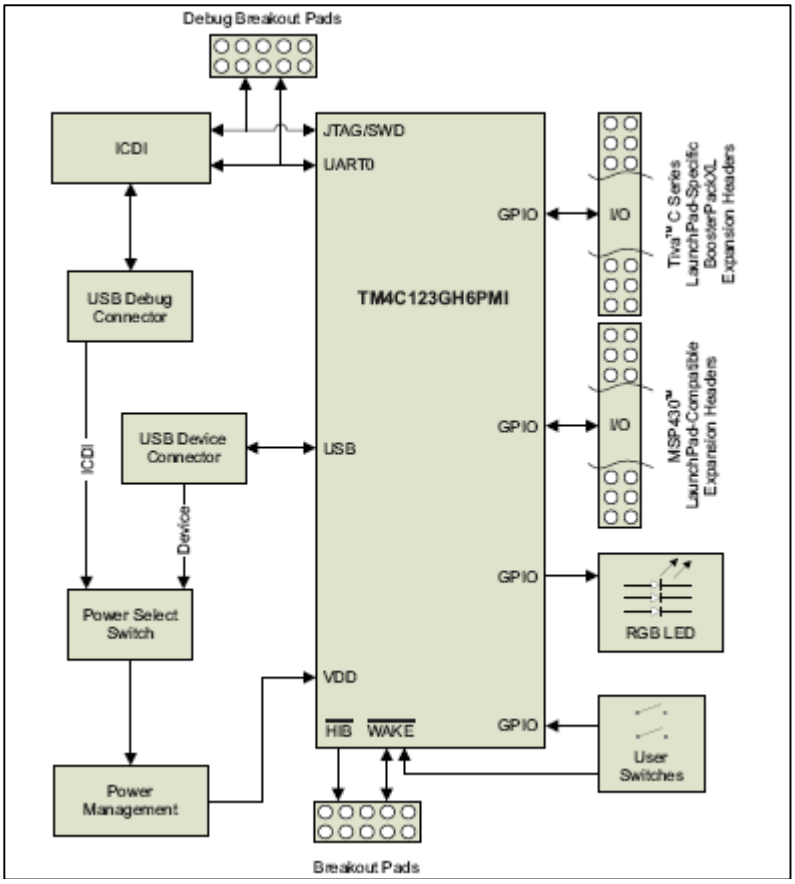


Figure 2-2 Tiva C Series Launchpad Evaluation Board Block Diagram
Reference: TI User Guide for Tiva C series LaunchPad, Pg. 7

EXPERIMENT:

1. Download the latest version of **TivaWare**. Double click exe file to install. Install in the directory where you would install the IDE
2. Switch the Power Select (top left corner) to right for debug mode on launchpad.
3. Connect the USB cable from Windows enabled PC to Debug USB port (top right corner highlighted in green in the following figure) on Tiva C series LaunchPad. This USB port provides debug and Virtual COM Port connectivity via the Tiva C Series In-Circuit Debug interface (ICDI). This should turn on the Gree Power LED.

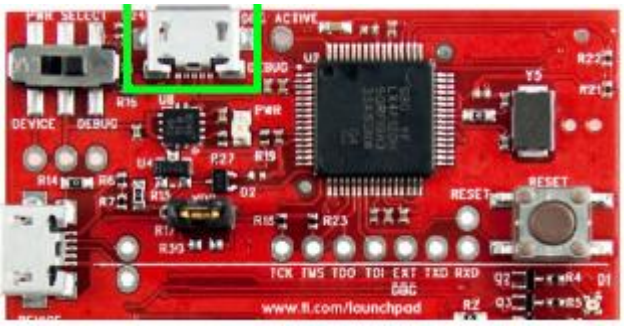


Figure 2-3 Debug USB Port

Installing Drivers:

4. Download LaunchPad drivers

5. Connect the LaunchPad to the Window system as described in above section via Debug port.
6. Go to Device manager you will see three unknown devices connected to USB port under other devices list as “In-Circuit Debug Interface”
7. Open these devices and click update driver one by one.
8. Give path of LaunchPad driver directory.

Using LM Flash Programmer

9. Run the LM Flash Programmer which is used for downloading code in the controller
10. In the Configuration tab, use the Quick Set control to select the EK-TM4C123GXL evaluation board.

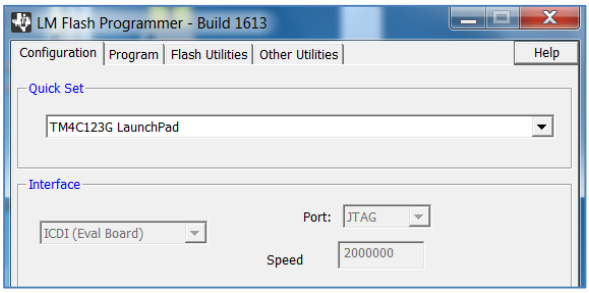


Figure 2-4 Configuration Tab

11. Move to the Program tab and click the Browse button. Navigate to the example applications directory. The default location is C:\ti\TivaWare_C_Series_<version>\examples\boards\ek-tm4c123gxl).
12. Each example application has its own directory. Navigate to the example directory that you want to load and then into the directory which contains the binary (*.bin) files. Select the binary file and click Open.
13. Set the “Erase Method” to “Erase Necessary Pages”, check the “Verify After Program” box, and check “Reset MCU After Program”. Program execution starts once the verify process is complete.

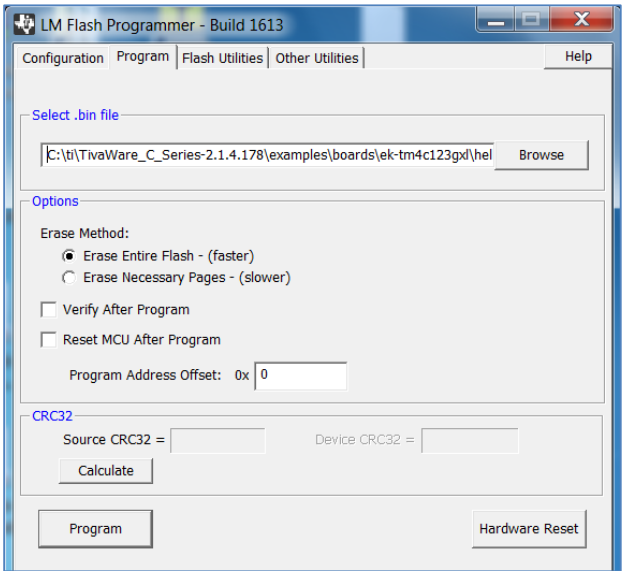


Figure 2-5 Selecting .bin file

LAB TASKS:

1. Run blinky project from C:\ti\TivaWare_C_Series-2.1.4.178\examples\boards\ek-tm4c123gxl\blinky project to your board. Open any of its .bin file and upload it to your controller. It is a very simple example that blinks the on-board LED using direct register access.

2. Run hello project from C:\ti\TivaWare_C_Series-2.1.4.178\examples\boards\ek-tm4c123gxl\hello. Open any of its .bin file and upload it to your controller. It is a very simple ``hello world" example. It simply displays ``Hello World!" on the UART and is a starting point for more complicated applications. UART0, connected to the Virtual Serial Port (running at 115,200 baudrate), is used to display messages from this application. For observing virtual port output use TeraTerm.

   Using TeraTerm

   a. Install Tera Term utility by clicking teraterm.exe
   b. Select I agreed and click next.
   c. Keep the default path for application to install and click next
   d. Install the rest of application keeping the default settings
   e. Launch Tera Term and select serial connection
   f. Open setup for serial
   g. Select the desired baud rate
   h. Connect LaunchPad and press reset button

POST LAB:

Run UART Echo program from C:\ti\TivaWare_C_Series-2.1.4.178\examples\boards\ek-tm4c123gxl\uart_echo. Open any of its .bin file and program it to your controller.This example application utilizes the UART to echo text. The first UART (connected to the USB debug virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

EXPERIMENT 3

CONFIGURING KEIL uVision4 TO CREATE AND RUN FIRST PROJECT

OBJECTIVE:

1. Familiarization with Keil uVision4 project setup and debugging options
2. Mechanism to load program and execute it on LaunchPad

EQUIPMENT:

- Tiva C Series TM4C LaunchPad Evaluation Board (EK-TM4C123GXL)
- On-board In-Circuit Debug Interface (ICDI)
- USB Micro-B plug to USB-A plug cable
- ***Software:***
    - Example Code: code.c
    - Extra dll for simulation: TM4C_extra.dll

EXPERIMENT:

Creating New Project:

i. Open Keil µVision v4. Click on Project >> New µVision Project and enter a name for the project

ii. You'll be asked to select a target microcontroller. For our board, select Texas Instruments and the device number TM4C123GH6PM

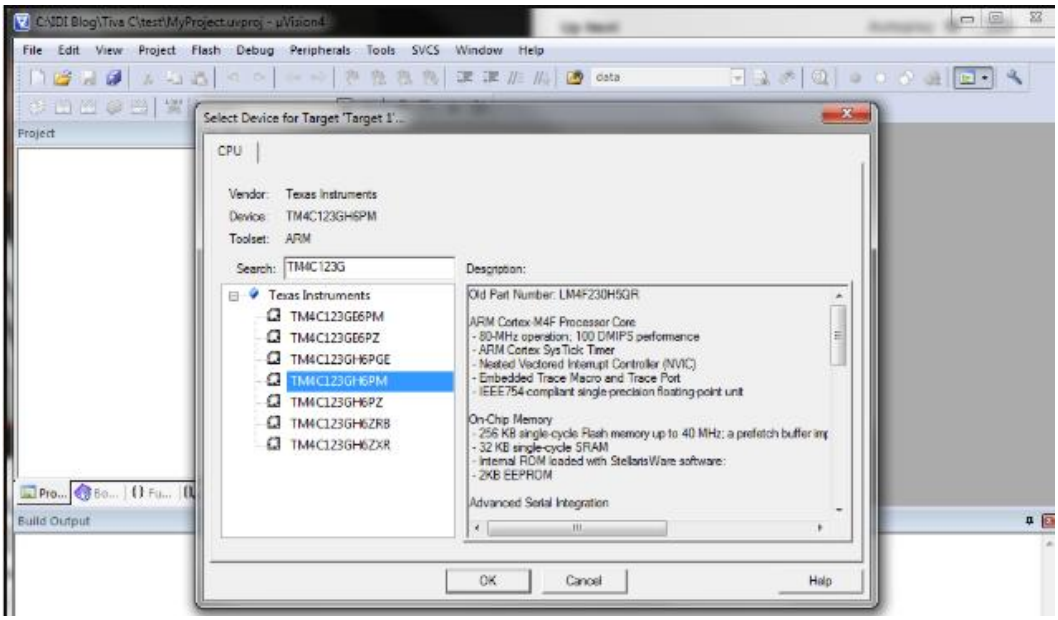


Figure 3-1 Selecting target device

iii. Accept copying the Startup project file when prompted

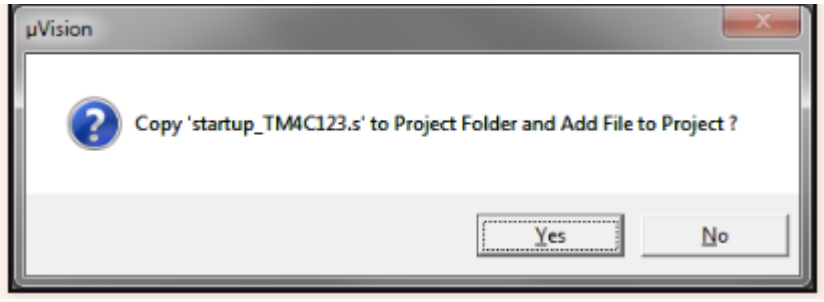


Figure 3-2 System prompt to install project startup file

iv. Your Project window should now show the startup file under Target1 >> Source Group

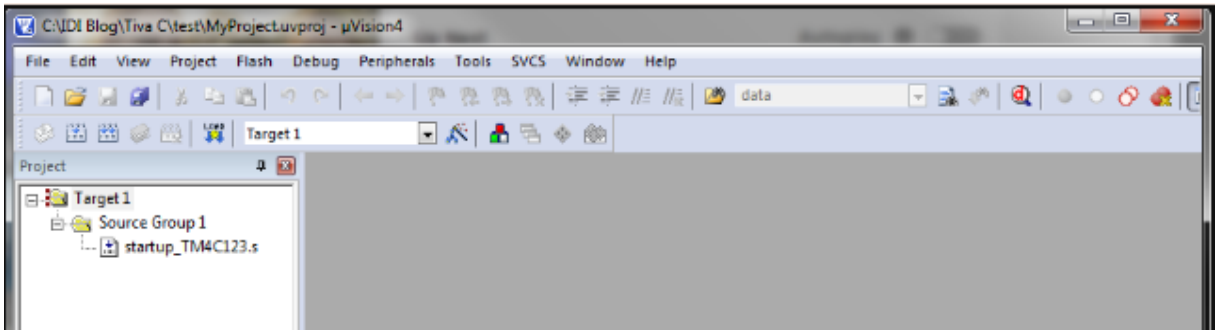


Figure 3-3 Screenshot of Keil with startup file

v. Open the startup file and replace its code with the code (startup.s) provided to you in lab

vi. Finally, right-click on Source Group 1 again and click Add New Item to Group 'Source Group 1'. Then select C File(.c), enter file name at the bottom and click Add. This will be your main program file.

vii. The IDE should show the startup and main file that you created under Source Group 1

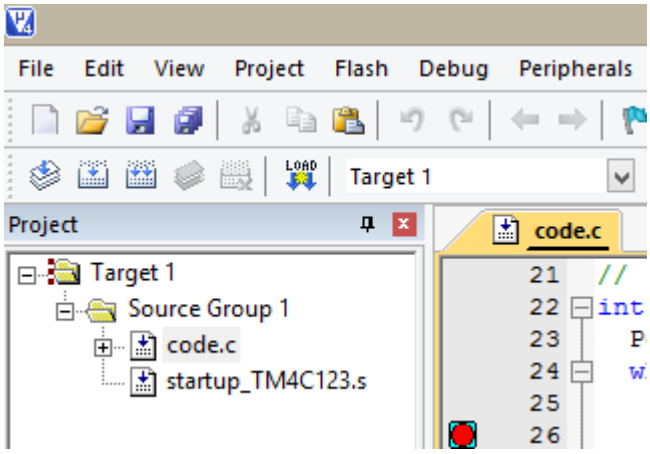


Figure 3-4 Screenshot of Keil after project files were added

viii. Open the .c file you created and copy and paste the following code:

```
#include "tm4c123gh6pm.h"  // Header file accesses registers in
//TMC123 microcontroller define easy to read names for registers
#define GPIO_PORTF_DATA_R       (*((volatile unsigned long *)0x400253FC))
#define GPIO_PORTF_DIR_R        (*((volatile unsigned long *)0x40025400))
#define GPIO_PORTF_AFSEL_R      (*((volatile unsigned long *)0x40025420))
#define GPIO_PORTF_PUR_R        (*((volatile unsigned long *)0x40025510))
#define GPIO_PORTF_DEN_R        (*((volatile unsigned long *)0x4002551C))
#define GPIO_PORTF_LOCK_R       (*((volatile unsigned long *)0x40025520))
```

```
#define GPIO_PORTF_CR_R         (*((volatile unsigned long *)0x40025524))
#define GPIO_PORTF_AMSEL_R      (*((volatile unsigned long *)0x40025528))
#define GPIO_PORTF_PCTL_R       (*((volatile unsigned long *)0x4002552C))
#define SYSCTL_RCGC2_R          (*((volatile unsigned long *)0x400FE108))

//   Global Variables
#define SW1 (GPIO_PORTF_DATA_R&0x10)
#define SW2 (GPIO_PORTF_DATA_R&0x01)
//   Function Prototypes
void PortF_Init(void);
void Delay(void);

// Subroutines Section
int main(void){
  PortF_Init();                         // Call initialization of port PF0-PF4
  while(1){

    if(!SW1){
      GPIO_PORTF_DATA_R ^= 0x08;        // LED is green
    Delay();                      // wait 0.1 sec
              }
    else if(!SW2){
      GPIO_PORTF_DATA_R ^= 0x02;        // LED is red
    Delay();                      // wait 0.1 sec
             }
             else
             {GPIO_PORTF_DATA_R ^= 0x04;       // LED is blue
    Delay();                      // wait 0.1 sec
             }


  }
}

// Subroutine to initialize port F pins for input and output
// PF4 and PF0 are input SW1 and SW2 respectively
// PF3,PF2,PF1 are outputs to the LED
void PortF_Init(void){
  volatile unsigned long delay;
  SYSCTL_RCGC2_R |= 0x00000020;     // 1) F clock
  delay = SYSCTL_RCGC2_R;           // reading register adds a delay
  GPIO_PORTF_LOCK_R = 0x4C4F434B;   // 2) unlock PortF PF0
  GPIO_PORTF_CR_R = 0x1F;           // allow changes to PF4-0
  GPIO_PORTF_AMSEL_R = 0x00;        // 3) disable analog function
  GPIO_PORTF_PCTL_R = 0x00000000;   // 4) GPIO clear bit PCTL
  GPIO_PORTF_DIR_R = 0x0E;          // 5) PF4,PF0 input, PF3,PF2,PF1 output
  GPIO_PORTF_AFSEL_R = 0x00;        // 6) no alternate function
  GPIO_PORTF_PUR_R = 0x11;          // enable pullup resistors on PF4,PF0
  GPIO_PORTF_DEN_R = 0x1F;          // 7) enable digital pins PF4-PF0
}

void Delay(void){unsigned long volatile time;
  time = 727240*200/91;  // 0.1sec
  while(time){ time--;} }
```

ix. Click on Project and Build Target. This command will compile your code and report any errors or warnings in the Build Output window.

x. Click on Project and Options for Target ‘Target 1’. Find the Utilities tab and under Configure Flash Menu Command, uncheck the Use Debug Driver box. Then, under Use

Target Driver for Flash Programming select the Stellaris ICDI driver and hit OK. This will allow us to program the board through USB.

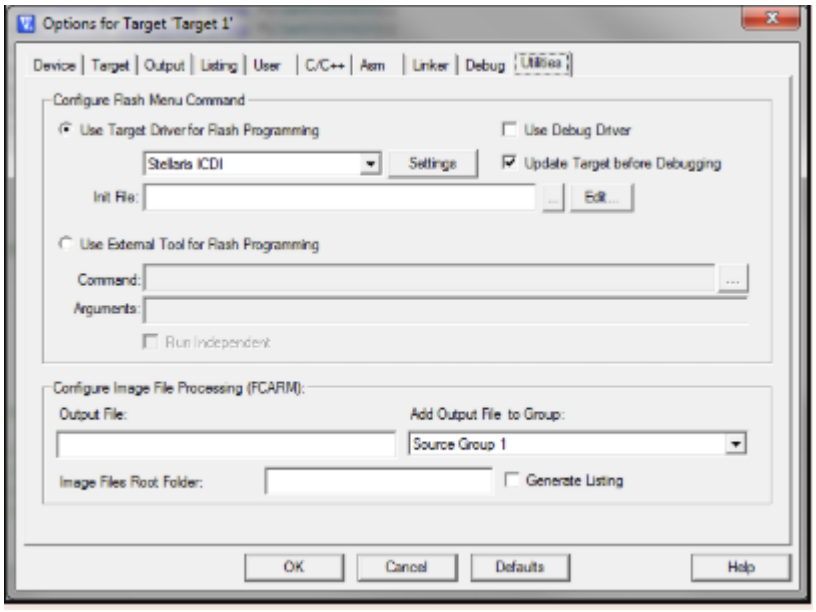


Figure 3-5 Configuring drivers to load code into LaunchPad

i. Click on Flash and then Download. This will load the compiled code to the microcontroller. The Build Output window will report the status of the process or any errors.

xi. Once the code is loaded, press the reset button in the LaunchPad to start the program.

About the code:
The blue LED should be blinking every 0.1 seconds. On pressing SW1, Green LED should also start blinking and on pressing SW2 Red LED should start blinking with the same rate.

xii. Paste the given dll in C:\Keil\ARM\BIN. Now, open project settings by go to Project->options for target. Goto debug tab and enter "-dTM4C_extra.dll" in the marked section in the figure below:

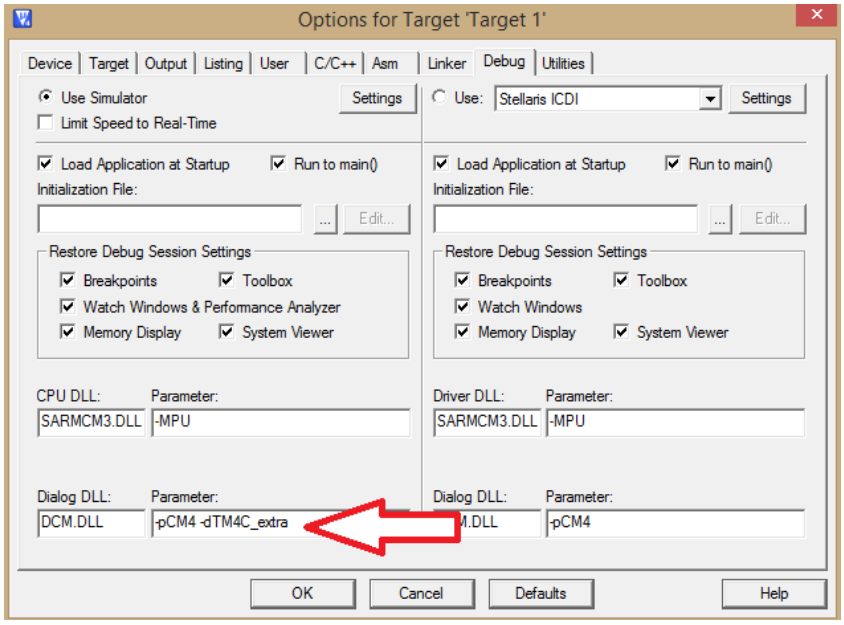


Figure 3-6 Adding dll to the project

This option will enable more features in simulations. Press Ok and rebuild the target. Now start debugging session. Goto Peripherals menu->TexasPortF. It will open following window

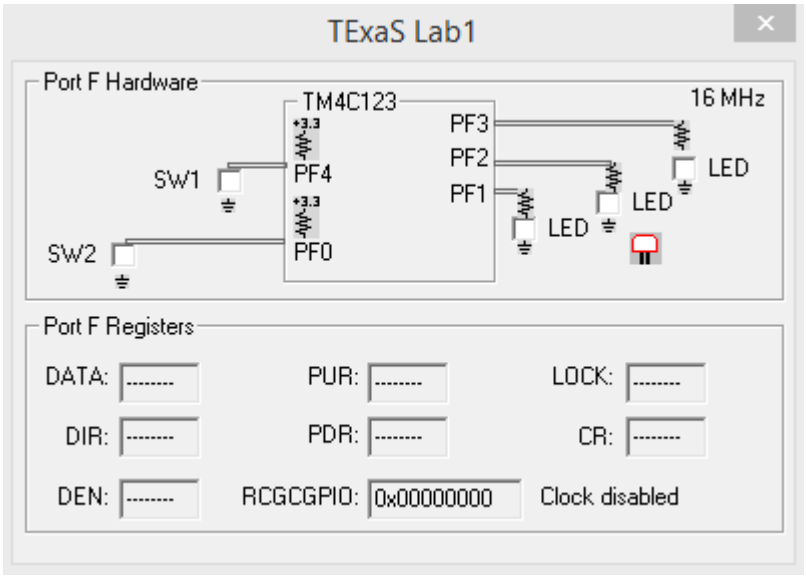


Figure 3-7 TM4C simulator option enabled by TM4C_extra.dll

xiii. Now press F5 and observe the simulation.

xiv. Goto View->Analysis window->Logic Analyzer. Press setup and enter a signal "PORTF.1" Set "display type" to "bit". Run the simulation and press SW2, this will show PF1 on logic analyzer.

POST LAB

i. Take screenshots of your simulations you performed in lab and include in your report

ii. Test in-circuit debugging option by selecting "Stellaris ICDI" in "Options for Target" menu. You will have to add a breakpoint at first if condition in the code to test your code step wise.

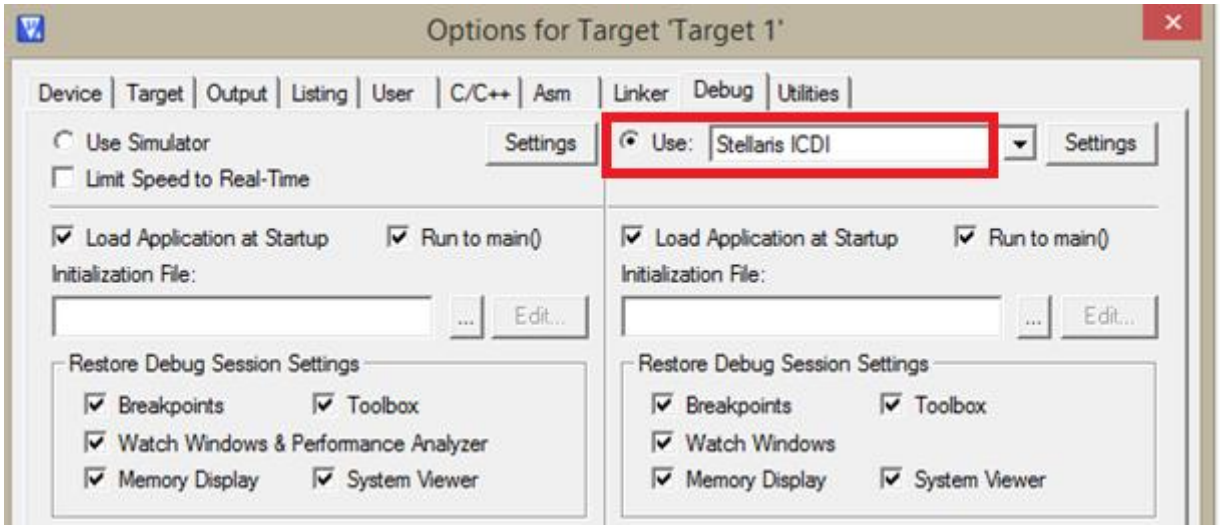


Figure 3-8 selecting debug option

Also, while debugging you may observe change in PORTF Data register in Keil by selecting GPIOF from System viewer as shown below:

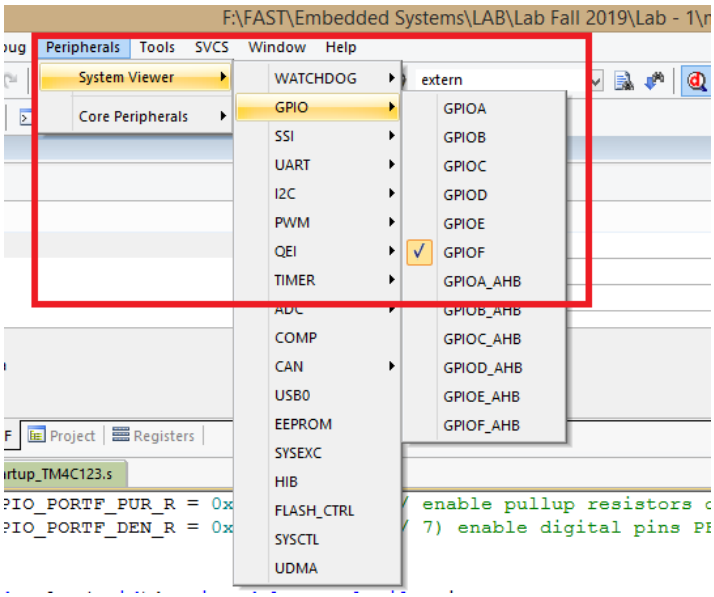


Figure 3-9 Viewing GPIO status from System Viewer

EXPERIMENT 4

CONFIGURING GPIO PORTS OF TM4C123GH6PM FOR DIGITAL I/O

OBJECTIVE:

1. Learn how to configuring GPIO for writing and reading data
2. Learn different ways to access SFRs of GPIO

EQUIPMENT

Tiva C Launchpad

BACKGROUND

A microcontroller communicates with the any device connected to it 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). These pins as general purpose input output (GPIO) pins. Any GPIO pin can be configured through software to be either a digital or analog input/ output.

There are six I/O Ports in TM4C: PORT A, B, C, D, E and F. There are 8 pins in Ports A-D, six in PORTE and five in PORTF. Each port also has an alternate function associated with it. For example, PORTA can be used in UART module.

A block diagram of ports with CPU buses is given below:

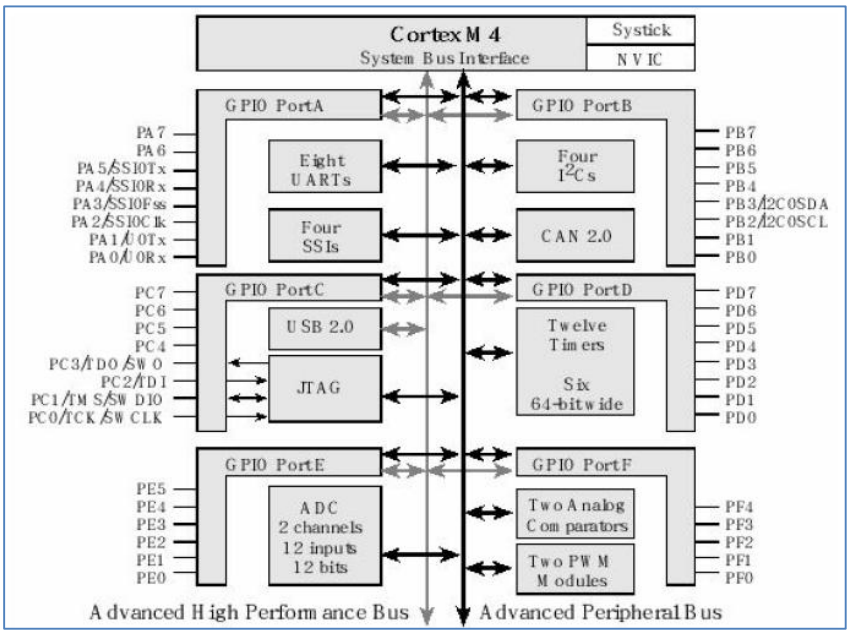


Figure 4-1 PORTs of TM4C

I/O ports are mapped into 4GB memory of processor from locations 0x40004000 to 0x40025FFF. The following figure shows the address range assigned to each GPIO port. 4K bytes of memory is assigned to each of the port. The reason being is each GPIO Port has a large number of special function registers associated with it. The ARM chips have two buses: APB (Advanced Peripheral

Bus) and AHB (Advanced High Performance Bus). The AHB bus takes 1 clock cycle to access peripherals and is much faster than APB that takes 2 clock cycles.

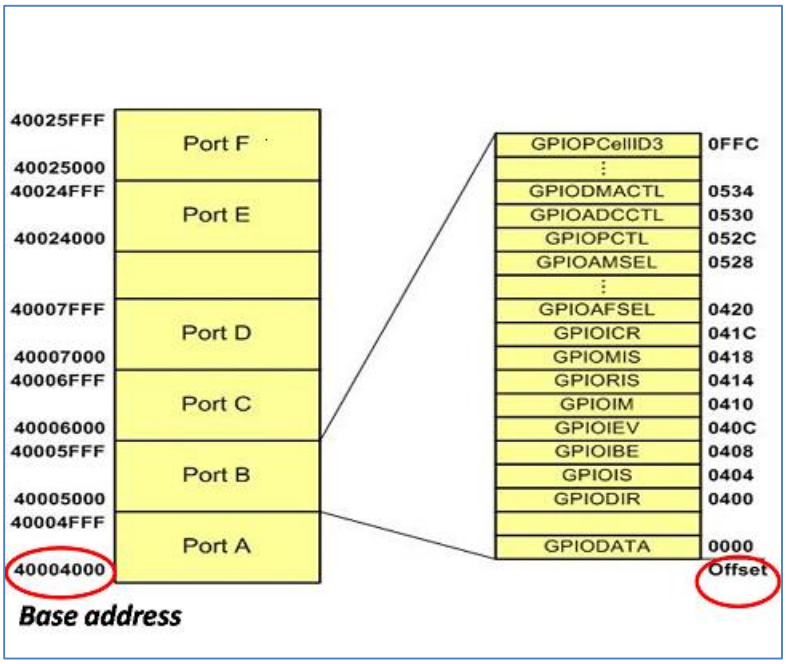


Figure 4-2 Memory Map of I/O Ports

The base addresses for the GPIOs Data registers are

Port	Base address
PortA	0x40004000
PortB	0x40005000
PortC	0x40006000
PortD	0x40007000
PortE	0x40024000
PortF	0x40025000

There are many registers associated with each of the port and they have designated in the memory map. Some of the registers are:

- **Clock Enable Register (RCGCGPIO):** is used to enable the clock source for the I/O port circuitry. If a port is not used then the clock source to it can be disabled in order to save power.
- **Direction Register (GPIODIR):** is used to make pin either an input or output
- **Data Register (GPIODATA):** is used to actually read or write the data. The data register is located at offset 0x0000 from the address of the port. This register supports bit-binding it occupies 256 words (offset 0x000 to 0x3FC). In order to access whole register at once, offset 0x3FC is used to compute the physical address. In order to modify single bit of a port, we must not modify other pins. Two methods are used for this purpose:
    - **Read-modify-write method:** Using masking concept, AND / OR operations. In order to make bit 5 of PORTF 1 without affecting other bits, PORTF_DATA_R= PORTF_DATA_R | 0x20

- **Bit Specific Addressing:** Define a macro for the respective physical address of those bits using bits binding concept. Use the following table of constants associated with each bit number.

If we wish to access bit	*Constant*
7	0x0200
6	0x0100
5	0x0080
4	0x0040
3	0x0020
2	0x0010
1	0x0008
0	0x0004

For example, assume we are interested in Port A bits 1, 2, and 3. The base address for Port A is 0x4000.4000, and the constants are 0x0008, 0x0010, and 0x0020. The sum of 0x4000.4000+0x0008+0x0010 +0x0020 is the address 0x4000.4038. If we read from 0x4000.4038 only bits 1, 2, and 3 will be returned. If we write to this address only bits 1, 2, and 3 will be modified.

```
#define PA123   *((volatile uint32_t *)0x40004038)
..
PA123=0x0A; //only bits 1,2,3 will be updated, rest will be same
```

- **Digital Enable Register (GPIODEN):** is used to enable a pin to work as a digital I/O instead of analog function

Detailed Steps to initialize GPIO:

1. ***Enable Clock:***

- Clock is enabled for each port using **SYSTCL_RCGCGPIO_R** port
- Clock is disabled to save power
- Register is 32 bit wide
- For enabling the clock we deal only with the last 8 bits.
  After enabling, wait for stabilizing of clock by waiting for its respective status bit **SYSCTL_PRGPIO_R** to be true

Note: R0 to R5 are used to enable the clock for ports A to F.
1: Enabled
0: Disabled

2. ***Unlock port (needed only for PD7 and PF0***)

Write 0x4C4F434B in GPIO_PORTX_LOCK_R register to unlock, other locks (reads 1 if locked, 0 if unlocked)

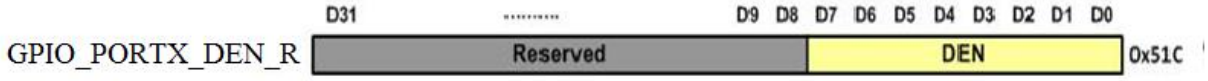
3. *Set direction of pin:*

Note: D0 to D7 are used to set the direction for pins 0 to 7 of the port.
1: Output
0: Input

4. *Enable changes to digital port:*

By writing 1 to the respective bit field of **GPIO_PORTX_CR_R** register allows us to make changes to the respective pins of the port.

5. *Enable digital function of the pins*



Note: D0 to D7 are used to Enable the digital circuit for pins 0 to 7 of the port.
1: The digital functions for the corresponding pin are enabled.
0: The digital functions for the corresponding pin are disabled.

6. *Enable/disable Pull-Up resistor (depends on scenario)*

- This microcontroller has in built pull-up resistor which are used when required
- Pull up resistors are enabled by setting respective pin in **GPIO_PORTx_PUR_R**
- Instead of using pull-up we are also allowed for using pull-down register by writing 1 to the **PDR** register instead of **PUR**. It is used if there is no pull up resistor connected in hardware.

7. *Write a logic to Data Register depending upon which LED to turn on or off*

Keil and TIVA C header files

There are two different header files available for TM4C123GH6PM.

1. **Keil uVision header file:** It is found in C:\Keil\ARM\INC\TI\TM4C123 folder. Each port is defined as a pointer to struct with the registers as the members of the struct. For example, PORTF direction register is referred as GPIOF->DIR

2. **TIVA ware header file:** It is found in C:\ti\TivaWare_C_Series 2.1.4.178\inc\tm4c123gh6pm.h. In this file each register is defined with its own name. For example, the direction register of the PORTF is referred to as GPIO_PORTF_DIR_R.

Launchpad connections:

- PC0 – PC3 are used for JTAG connections to the debugger on the Launchpad.
- PF1-PF3 are connected to RGB LEDs and PF0 & PF4 are connected to push buttons on TI Tiva Launcpad.

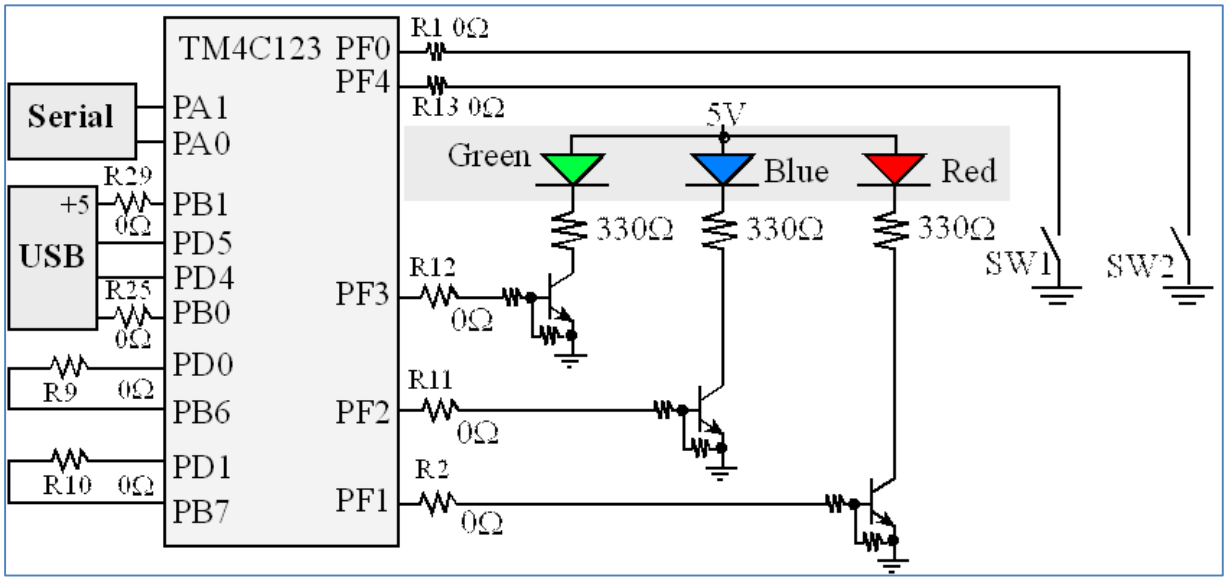


Figure 4-3 PORTF Connections with Swithces and LEDs on Launchpad

Sample code:

Code# 1: Toggling LEDs in TIVA Launchpad after 0.5 seconds

```
/*PF1 - red LED
  PF2 - blue LED
  PF3 - green LED
  They are high active (a '1' turns ON the LED).*/


/* PORTF data register */
#define PORTFDAT (*((volatile unsigned int*)0x400253FC))
/* PORTF data direction register */
#define PORTFDIR (*((volatile unsigned int*)0x40025400))
/* PORTF digital enable register */
#define PORTFDEN (*((volatile unsigned int*)0x4002551C))
/* run mode clock gating register */
#define RCGCGPIO (*((volatile unsigned int*)0x400FE608))
/* coprocessor access control register */
#define SCB_CPAC (*((volatile unsigned int*)0xE000ED88))

void delayMs(int n);     /* function prototype for delay */

int main(void)
{
    /* enable clock to GPIOF at clock gating register */
    RCGCGPIO |= 0x20;
    /* set PORTF pin3-1 as output pins */
    PORTFDIR = 0x0E;
    /* set PORTF pin3-1 as digital pins */
    PORTFDEN = 0x0E;
```

```
    while(1)
    {
        /* write PORTF to turn on all LEDs */
        PORTFDAT = 0x0E;
        delayMs(500);
        /* write PORTF to turn off all LEDs */
        PORTFDAT = 0;
        delayMs(500);
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {}  /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system
specific initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access*/
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB_CPAC |= 0x00F00000;
}
```

Code# 2: Toggle Blue LED with SW1

```
#include <stdio.h>
#include <stdint.h>
#include "C:\ti\TivaWare_C_Series-2.1.4.178\inc\tm4c123gh6pm.h"

#define SW1 (GPIO_PORTF_DATA_R&0x10)
void PortF_Init(void){
SYSCTL_RCGCGPIO_R |= 0x00000020; // activate clock for Port F
while((SYSCTL_PRGPIO_R&0x00000020) == 0){};// ready?
GPIO_PORTF_LOCK_R = 0x4C4F434B; // unlock GPIO Port F
GPIO_PORTF_CR_R = 0x14; // allow changes to PF2 & PF4
GPIO_PORTF_DIR_R = 0x04; // PF4 in, PF2 out
GPIO_PORTF_PUR_R = 0x10; // enable pull-up on PF4
GPIO_PORTF_DEN_R = 0x14; //  enable digital I/O on PF2 & PF4}
void main(void)
{PortF_Init();
while(1){
if (!SW1)
   GPIO_PORTF_DATA_R=GPIO_PORTF_DATA_R^0x04;
}
}
```

LAB TASKS:

1. Modify the sample code # 2 by bit-specific addressing method for accessing PF4
2. Implement a 3 bit counter (000 to 111) using three LEDs of launchpad. Counter value should be updated every second.

For all the problems you may use this delay routine:

Use this delay routine which approximately adds delay of 1 second

```
void delay()
{
uint32_t counter=0;
for (counter=0; counter<1500000;counter++);
}
```

POST LAB:

Implement a 2-bit AND gate using switches and any LED of Launchpad. Use bit-specific addressing method.

EXPERIMENT 5

GETTING STARTED WITH BOOSTERPACK

OBJECTIVE:

- Getting familiar with BOOSTXL-EDUMKII BoosterPack.
- Explore the usage of BoosterPack peripherals

BACKGROUND:

The BOOSTXL-EDUMKII BoosterPack plug-in module is an easy-to-use plug-in module that offers a high level of integration for developers to quickly add to LaunchPad development kit designs. Various analog and digital I/O are present in this board which include analog joystick, environmental and motion sensors, RGB LED, microphone, buzzer, color LCD display, and more. Some other features of boosterpack are:

- TI OPT3001 light sensor
- TI TMP006 temperature sensor
- Servo motor connector
- 3-axis accelerometer
- RGB multicolor LED
- Piezo buzzer
- Color 128x128 TFT LCD display
- Microphone
- 2-axis joystick with pushbutton
- User push buttons

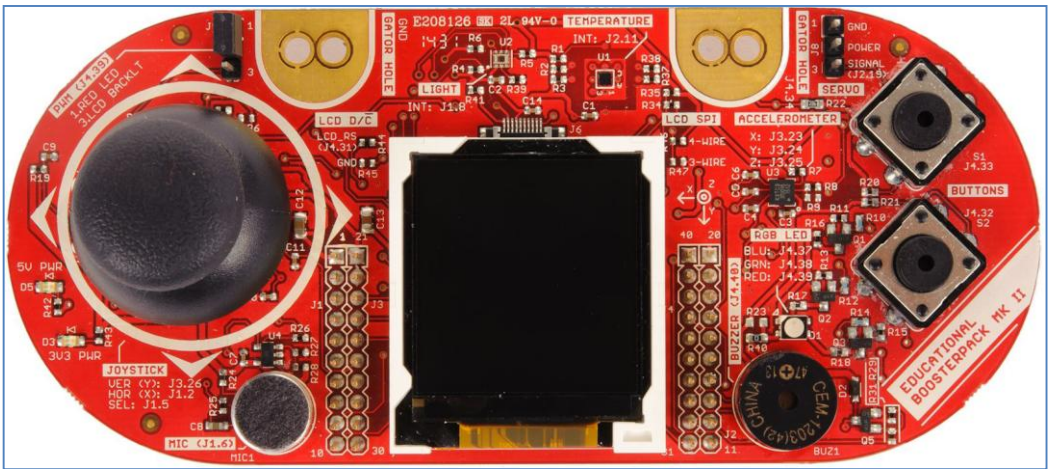


Figure 5-1 BOOSTXL-EDUMKII BoosterPack Plug-in Module

The 40-pin standard is compatible with the 20-pin standard that is used by other LaunchPad development kits. We can easily connect our Launchpad with Boosterpack via header. Internal connections of each module on boosterpack is given in the following pinout diagram. For example, Blue, Green and Red LEDs of "RGB Multicolor LED Module" are connected with J4.37, J4.38 and J4.39 respectively.

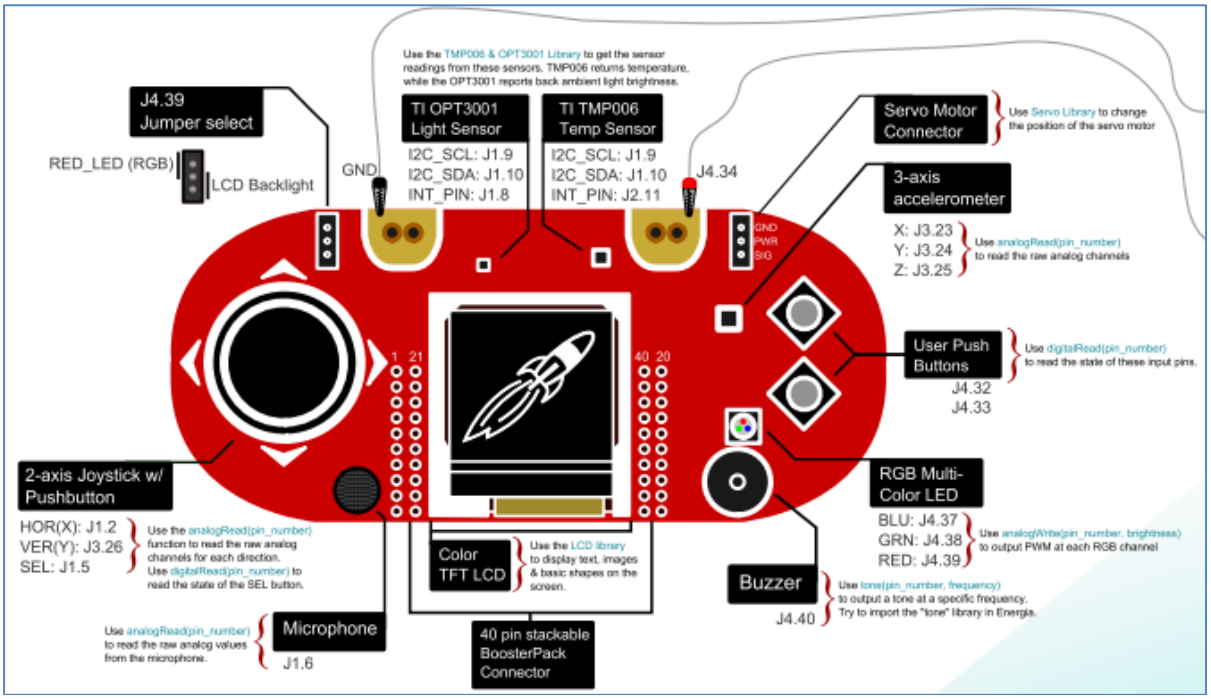


Figure 5-2 A closer look at the booster pack

For this lab you must understand the connections of the following modules on Booster pack:

RGB Multicolor LED

The Cree CLV1A-FKB RGB multicolor LED light output can make any color by mixing red, green, and blue. Each color channel can individually be modified by pulse width modulation (PWM) to achieve the desired color. The reference designator for the RGB LED is D1

BoosterPack Plug-in Module Header Connection	Pin Function
J4.37	Blue channel
J4.38	Green channel
J4.39	Red channel

Piezo Buzzer

The CUI CEM-1203(42) piezo buzzer can play various frequencies based on the user-provided PWM signal. You can even play different tones back to back to create a song. The reference designator for the piezo buzzer is BUZ1.

BoosterPack Plug-in Module Header Connection	Pin Function
J4.40	Buzzer input

User Pushbuttons

The user pushbuttons on the BOOSTXL-EDUMKII are connected to pull up resistors that drive the BoosterPack plug-in module pin high until the button is pressed and the pin is driven low. The reference designators for the user pushbuttons are S1 and S2.

BoosterPack Plug-in Module Header Connection	Pin Function
J4.32	S2 button
J4.33	S1 button

Power

The board was designed to be powered by the attached LaunchPad development kit, and requires both 3.3-V and 5-V power rails. Some 20-pin LaunchPad development kits like MSP-EXP430FR4133 may not provide the necessary 5-V power, which will limit the functionality.

Note:

- You may connect boosterpack directly with Launchpad. But LCD doesnot work this way as its connections overlap with some sensors. In order to use LCD use jumper wires.
- Boosterpack examples come with energia IDE. You may download it from https://energia.nu/download/

LAB TASKS:

Connect Launchpad with boosterpack and turn on/off the buzzer using any of the push buttons on boosterpack

POST LAB:

Install Energia IDE and try running some booskterpack examples

EXPERIMENT 6

UART MODULE IN TM4C123GH6PM

OBJECTIVE:

Configuring UART module to communicate with PC

EQUIPMENT:

- TI Launcpad
- PC
- Tera Term desktop application

BACKGROUND:

Universal Asynchronous Receiver Transmitter (UART)

The serial port allows the microcontroller to communicate with devices such as other computers, printers, input sensors and LCDs. Serial transmission involves sending one bit at a time.
The total number of bits transmitted per second is called the **baud rate**.
A **frame** is the smallest complete unit of serial transmission which includes a **start bit** (which is 0), 8 bits of data (least significant bit first), and a **stop bit** (which is 1).

UART Module on TM4C

The TM4C microcontrollers have upto 8 UART Ports (UART0-UART7). General block diagram of UART module is given below. In the TI Launchpad UART0 of TM4C is connected to the ICDI (In Circuit Debug Interface) which is connected to a USB connector. This ICDI USB connection can be used as virtual COM port. When the Launchpad is connected with PC, the device driver at the host PC establishes virtual connection between PC and the Launchpad as UART0. So, for all experiments we will be using UART0 module of TM4C.

UART0 uses PA0 and PA1 pins as alternate functions for TX0 and RX0 respectively.

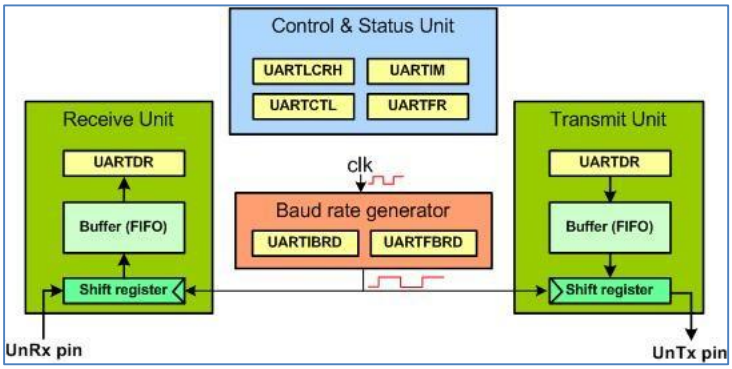
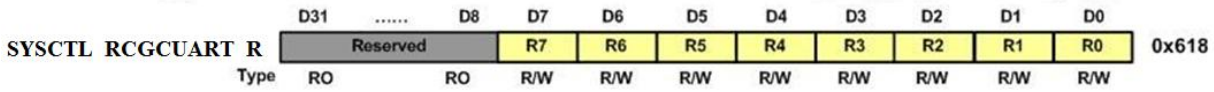


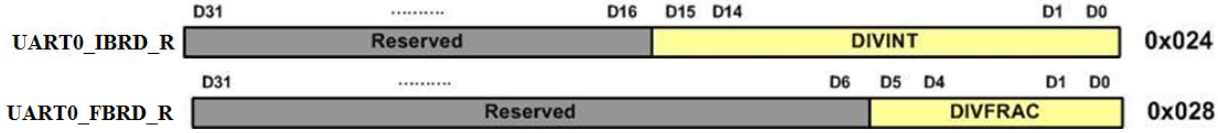
Figure 6-1 UART Module of TM4C

Steps involved in programming UART

- To activate a UART you will need to turn on the UART clock in the **SYSCTL_RCGCUART_R** register. You should also turn on the clock for the digital port in the **SYSCTL_RCGCGPIO_R** register.



- **Baud Rate settings:** there are two registers to set the baud rate. UART integer baud rate divisor (UARTn_IBRD) and UART Fractional Baud rate divisor (UARTn_FBRD)



  Only 16 bits are used from IBRD register and 6 bits from the FBRD register making total of 22 bits (16 bit integer + 6 bit fractional part). To reduce the error both divisor and fractional parts are used. Baud rate can be calculated as

  Desired baud rate = SysClk / (16 x ClkDiv)

  Where the SysClk is the working system clock connected to the UART. The default clock is 16MHz so above formula is
  reduced to

  Desired Baud rate = 1MHz/ClkDiv

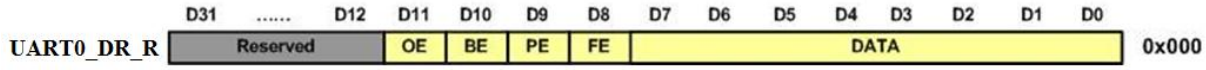
  The ClkDiv value gives us both the integer and the fractional values.
  UARTn_IBRD_R = Decimal Part of ClkDiv
  UARTn_FBRD_R= Fractional Part of ClkDiv x 64 + 0.5

- Now set the control register value. It is a 32 bit register but few of them are useful to us including RXE, TXE, HSE and UARTEN.

  TXE is the Transmitter Enable bit, and **RXE** is the Receiver Enable bit. We set **TXE**, **RXE**, and **UARTEN** equal to 1 in order to activate the UART device. However, we should clear **UARTEN** during the initialization sequence. HSE is high-speed enable bit that allows higher baud rate settings. By default the system clock is divided by 16, if we want to divide by 8 then set HSE to 1.

- UART line control register is used to set the frame size: number of bits per character in a frame and number of stop bits among other things. Usually 8 bit length is selected and FIFO is enabled.

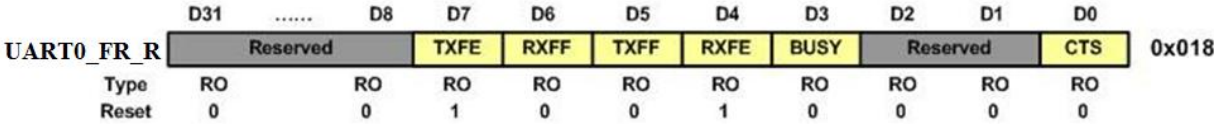
- UART Data register is used for transmitting and receiving data. A write to this register initiates the transmission. The received byte is placed in the register and must be retrieved by reading it before it is lost.



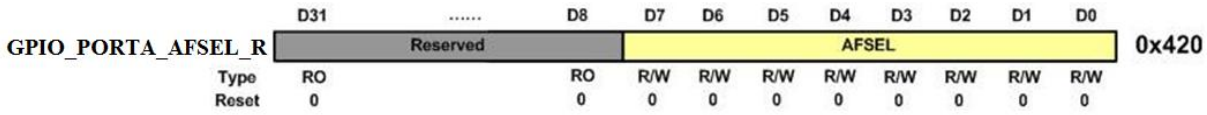
The OE, BE, PE, and FE are error flags associated with the receiver. You can see these flags in two places: associated with each data byte in **UART0_DR_R** or as a separate error register in **UART0_RSR_R** . The overrun error (**OE**) is set if data has been lost because the input driver latency is too long. **BE** is a break error, meaning the other device has sent a break. **PE** is a parity error (however, we will not be using parity). The framing error (**FE**) will get set if the baud rates do not match. The software can clear these four error flags by writing any value to **UART0_RSR_R** .

- The status of the two FIFOs can be seen in the **UART0_FR_R** register.
  - The **BUSY** flag is set while the transmitter still has unsent bits, even if the transmitter is disabled. It will become zero when the transmit FIFO is empty and the last stop bit has been sent. If you implement busy-wait output by first outputting then waiting for **BUSY** to become 0 then the routine will write new data and return after that particular data has been completely transmitted.

  - TXFE: it is raised when FIFO is empty
  - RXFF: it is raised when FIFO is full
  - TXFF: low when FIFO is not full
  - RXFE: low when FIFO is empty



- **I/O Port settings:** We must configure I/O pins associated with UART for their alternate functions. First enable alternate function and then chose alternate function of those pins



Make bit for respective pin = 1 for which alternate function has to be enabled

Chose alternate function for particular pin by setting a value in PCTL register according to the table below

Pin	Digital Function (GPIOPCTL PMCx)															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
17	PA0	U0Rx							CAN1Rx							
18	PA1	U0Tx							CAN1Tx							
45	PB0	U1Rx						T2CCP0								
46	PB1	U1Tx						T2CCP1								
16	PC4	U4Rx	U1Rx		M0PWM6		IDX1	WT0CCP0	U1RTS							
15	PC5	U4Tx	U1Tx		M0PWM7		PhA1	WT0CCP1	U1CTS							
14	PC6	U3Rx					PhB1	WT1CCP0	USB0EPEN							
13	PC7	U3Tx						WT1CCP1	USB0PFLT							
43	PD4	U6Rx						WT4CCP0								
44	PD5	U6Tx						WT4CCP1								
53	PD6	U2Rx			M0FAULT0		phA0	WT5CCP0								
10	PD7	U2Tx					phB0	WT5CCP1	NMI							
9	PE0	U7Rx							USB0EPEN							
8	PE1	U7Tx							USB0PFLT							
59	PE4	U5Rx		I2C2SCL	M0PWM4	M1PWM2			CAN0Rx							
60	PE5	U5Tx		I2C2SDA	M0PWM5	M1PWM3			CAN0Tx							
28	PF0	U1RTS	SSI1Rx	CAN0Rx		M1PWM4	PhA0	T0CCP0	NMI							
29	PF1	U1CTS	SSI1Tx			M1PWM5	PhB0	T0CCP1								

Figure 6-2 I/O Pins alternate functions

LAB TASK:

Following are the functions that transmit and receive a single character to and from serial port.

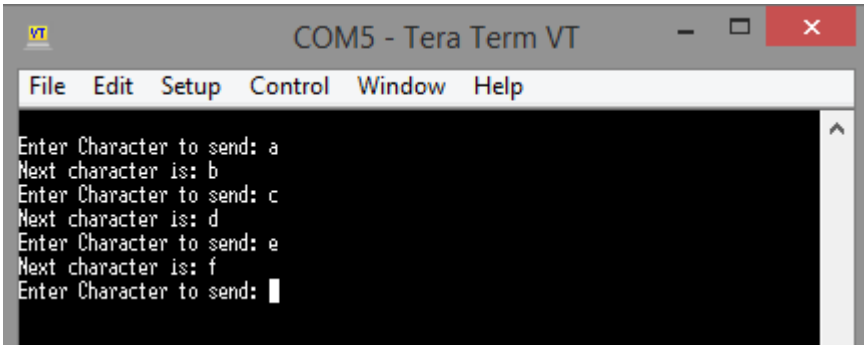
```
void UART0Tx(char c)
{
while((UART0_FR_R & 0x0020) !=0);
   UART0_DR_R = c;
}

char UART0Rx(void){
while ((UART0_FR_R&0x0010)!=0);
return ((char)(UART0_DR_R&0xFF));
}
```

Using **UART0Tx(char)** function, define another function **void sendString(char*ptr)** that sends a string to serial port.

Using all these functions write a code that receives a character from PC and sends next character to PC. Baud rate should be set to 115200.

The output on TeraTerm should be as given in screenshot.



Note: \n is used for new line character and \r is used for moving cursor to start of line

POST LAB:

There are two push buttons on Launchpad. Write a program that detects button press and print on Tera Term screen which button is pressed.

EXPERIMENT 7
SYNCHRONOUS SERIAL INTERFACE

OBJECTIVE:

- To understand concept of synchronous communication
- To learn how to configure SSI module of TM4C

EQUIPMENT:

- Two TI Launchpads

BACKGROUND:

SSI allows microcontrollers to communicate synchronously with peripheral devices and other microcontrollers. Another name for this protocol is Serial Peripheral Interface or SPI. The fundamental difference between a UART, which implements an asynchronous protocol, and a SSI, which implements asynchronous protocol, is the manner in which the clock is implemented. Two devices communicating with asynchronous serial interfaces (UART) operate at the same frequency (baud rate) but have two separate clocks. With a UART protocol, the clock signal is not included in the interface cable between devices. Two devices communicating with synchronous serial interfaces (SSI) operate from the same clock (synchronized). With a SSI protocol, the clock signal is included in the interface cable between devices. Typically, the master device creates the clock, and the slave device(s) uses the clock to latch the data (in or out.)

The SSI protocol includes four I/O lines: Clock, Rx, Tx and Slave Select (SS). Following figure shows the general architecture of SSI

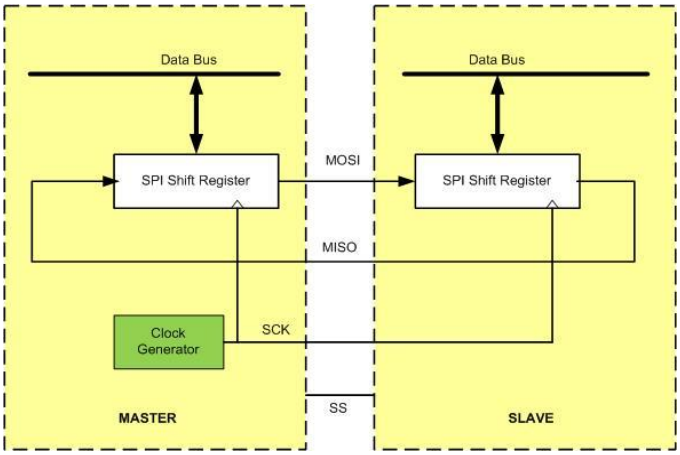


Figure 8-1 Block Diagram of SSI Module of TM4C

SSI Programming in TM4C

Tiva C series microcontroller has 4 Synchronous Serial Interface or SSI modules. The SSI on the TM4C employs two hardware FIFOs. Both FIFOs are 8 elements deep and 4 to 16 bits wide, depending on the selected data width.

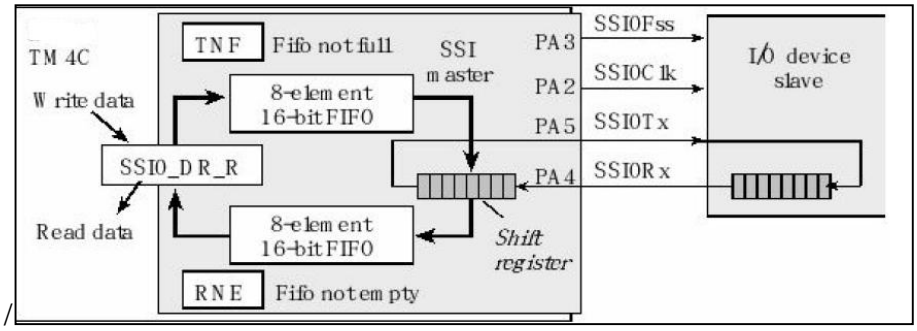


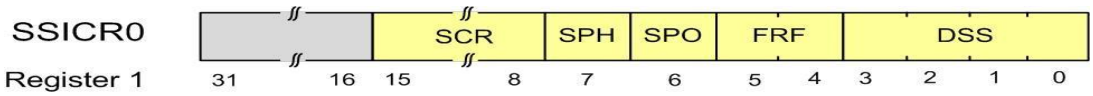
Figure 8-2 SSI Module of TM4C

Following are the registers involved in programming

- Enable clock to SSI module by setting values in RCSCSSI register
- Do all the configurations for the I/O port associated with the chosen SSI Module

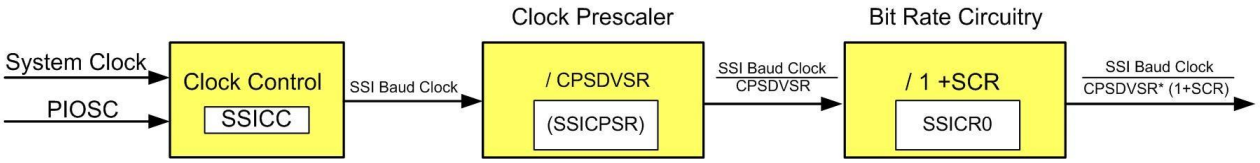
SSI Module Pin	GPIO Pin	SSI Module Pin	GPIO Pin
SSI0Clk	PA2	SSI2Clk	PB4
SSI0Fss	PA3	SSI2Fss	PB5
SSI0Rx	PA4	SSI2Rx	PB6
SSI0Tx	PA5	SSI2TX	PB7
SSI1Clk	PD0 or PF2	SSI3Clk	PD0
SSI1Fss	PD1 or PF3	SSI3Fss	PD1
SSI1Rx	PD2 or PF0	SSI3Rx	PD2
SSI1TX	PD3 or PF1	SSI3TX	PD3

- Set the control register of SSI SSCCRO



Bits	Name	Function	Description
0-3	DSS	SSI Data Size Select	0x03 to 0x0F for 4-bit to 16-bit data size 0x07 means 8-bit data size
4-5	FRF	SSI Frame Format Select	0 for SPI, 1 for TI, and 2 for MICROWIRE frame format
6	SPO	SSI Serial Clock Polarity	Clock polarity
7	SPH	SSI Serial Clock Phase	Clock phase
8-15	SCR	SSI Serial Clock Rate	BR=SysClk/(CPSDVSR * (1 + SCR))

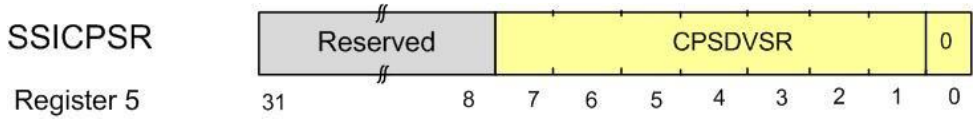
- **Setting bit rate:** SSI Module clock source can be either system clock or internal oscillator. The selected frequency is fed to prescaler before it is used by the bit rate circuity.



The SSI clock frequency is established by the 8-bit field **SCR**field in the **SSI0_CR0_R** register and the 8-bit field **CPSDVSR**field in the **SSI0_CPSR_R** register. **SCR** can be any 8-bit value from 0 to 255. **CPSDVSR** must be an even number from 2 to 254. Let f_{BUS} be the frequency of the bus clock. The frequency of the SSI is

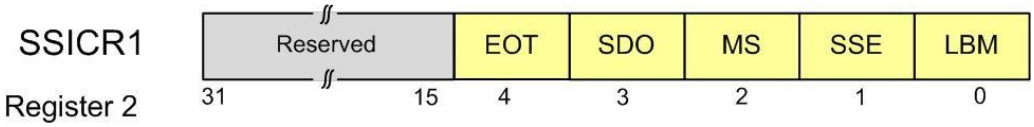
$f_{SSI} = f_{BUS} / (\mathbf{CPSDVSR} * (1 + \mathbf{SCR}))$

The frequency of the SSI is
Rate = SysClk / (CPSDVSR * (1 + SCR))

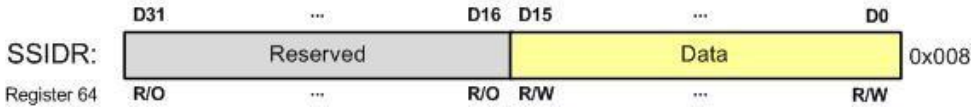


SCR bit is set in SSICR0 register.

- Chose Master or Slave from SSICR1 register bit MS.SSE is used for enable/disable SSI



- When performing I/O the software puts into the transmit FIFO by writing to the **SSI0_DR_R** register and gets from the receive FIFO by reading from the **SSIDR** register.



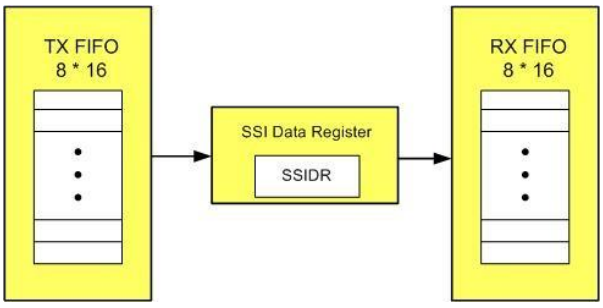
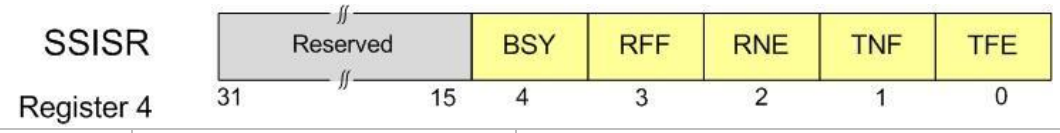


Figure 8-3 TX and RF FIFO

If there is data in the transmit FIFO, the SSI module will transmit it. With SSI it transmits and receives bits at the same time. When a data transfer operation is performed, this distributed 8- to 32- bit register is serially shifted 4 to 16 bit positions by the SCK clock from the master so the data is effectively exchanged between the master and the slave. Data in the master shift register are transmitted to the slave. Data in the slave shift register are transmitted to the master. Typically, the microcontroller is master and the I/O module is the slave, but one can operate the microcontroller in slave mode. When designing with SSI, you will need to consult the data sheets for your specific microcontroller.

- Status of SSI communication can be monitored by reading flags in SSISR register



Bits	Name	Function	Description
0	TFE	SSI Transmit FIFO Empty	The bit is 1 when the transmit FIFO is empty
1	TNF	SSI Transmit FIFO Not Full	The bit is 1 when the SSI transmit FIFO not full
2	RNE	SSI Receive FIFO Not Empty	The bit is 1 when the receive FIFO is not empty
3	RFF	SSI Receive FIFO Full	The bit is 1 when the receive FIFO is full
4	BSY	SSI Busy Bit	The bit is 1 when the SSI is currently transmitting or receiving

Common status bits for the SPI module include:

- BSY, SSI is currently transmitting and/or receiving a frame, or the transmit FIFO is not empty
- RFF, SSI receive FIFO is full
- RNE, SSI receive FIFO is not empty
- TNF, SSI transmit FIFO is not full
- TFE, SSI transmit FIFO is empty

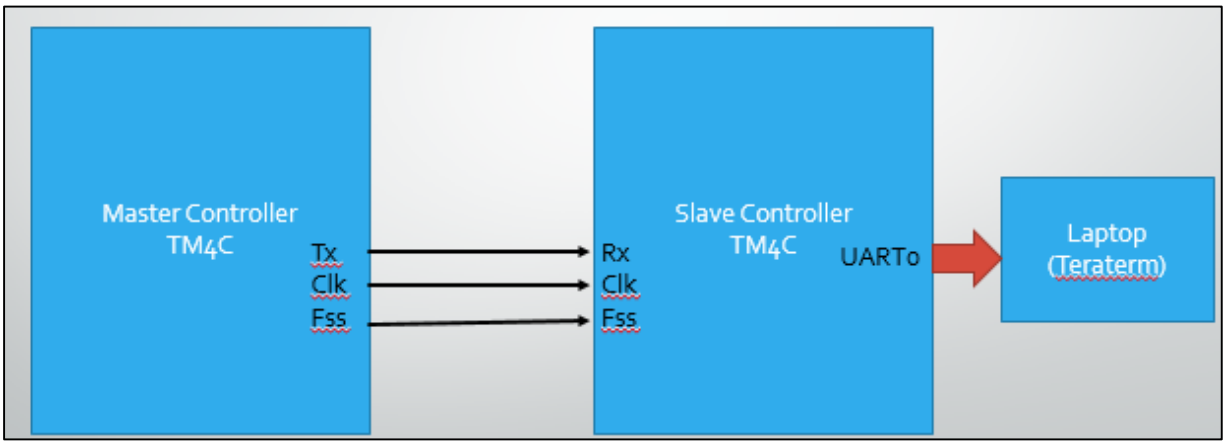
Wait on busy flag for reading/writing for single byte transfer at a time

LAB TASK:
For the problems given below, consider the following scenario:

Two TM4C controllers are connected with each other via SPI. Assume both are using SSI2 module. One is configured as a master and the other is slave. Master sends upper case letter to slave and slave returns its lower case

POST LAB:
Master wants the slave to turn on a specific LED (RGB) connected on its launchpad. Master sends a character R, G, or B to the slave to turn on its red, green or blue led respectively. One led should

be on at a time. Slave should also forward the message received by master to the PC connected with it via UART0.



EXPERIMENT 8
INTERRUPTS

OBJECTIVE:

- To learn about interrupts of TM4C
- To understand concept of NVIC in TM4C

EQUIPMENT:
TI LaunchPad

BACKGROUND:
An **interrupt** is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a **trigger**.
On the ARM ® Cortex™-M processor, exceptions include resets, software interrupts and hardware interrupts. Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of memory.
Following table lists the interrupt sources available on the TM4C family of microcontrollers.

Interrupt #	**Interrupt**	**Memory Location (Hex)**
	Stack Pointer initial value	0x00000000
1	Reset	0x00000004
2	NMI	0x00000008
3	Hard Fault	0x0000000C
4	Memory Management Fault	0x00000010
5	Bus Fault	0x00000014
6	Usage Fault (undefined instructions, divide by zero, unaligned memory access,...)	0x00000018
7	Reserved	0x0000001C
8	Reserved	0x00000020
9	Reserved	0x00000024
10	Reserved	0x00000028
11	SVCall	0x0000002C
12	Debug Monitor	0x00000030
13	Reserved	0x00000034
14	PendSV	0x00000038
15	SysTick	0x0000003C
16	IRQ 0 for peripherals	0x00000040
17	IRQ 1 for peripherals	0x00000044
…	…	…
255	IRQ 239 for peripherals	0x000003FC

Interrupt numbers 0 to 15 contain the faults, software interrupt and SysTick. Rest of the interrupts are hardware IRQ interrupts. There are up to 240 possible interrupt sources and their 32-bit vectors are listed in order starting with location 0x0000.0008. From a programming perspective, we can attach ISRs to interrupts by writing the ISRs as regular assembly subroutines or C functions with no input or output parameters and editing the **Startup.s** file to specify those functions for the appropriate interrupt. For example, if we wrote a Port C interrupt service routine named **PortCISR** , then we would replace **GPIOPortC_Handler** with **PortCISR** . In this book, we will write our ISRs using standard function names so that the **Startup.s** file need not be edited. I.e., we will simply name the ISR for edge-triggered interrupts on Port C as **GPIOPortC_Handler** . The ISR for this interrupt is a 32-bit pointer located at ROM address 0x0000.0048. Because the vectors are in ROM, this linkage is defined at compile time and not at run time.

NVIC in ARM Cortex - M

Interrupts on the Cortex-M are controlled by the Nested Vectored Interrupt Controller (NVIC). To activate an interrupt source we need to set its priority and enable that source in the NVIC. In order to configure and enable an interrupt following steps are performed:

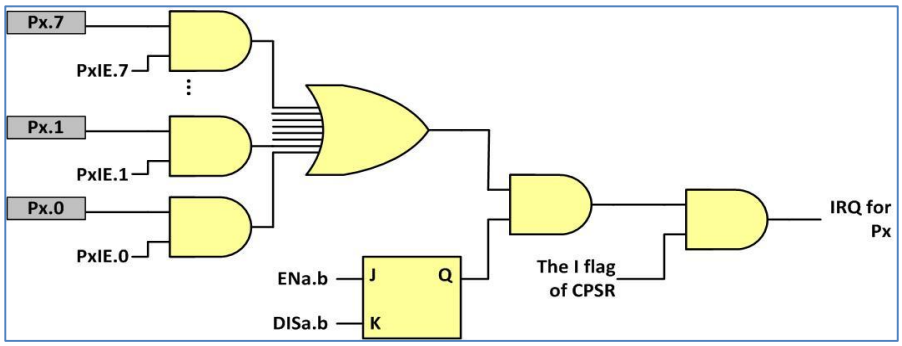


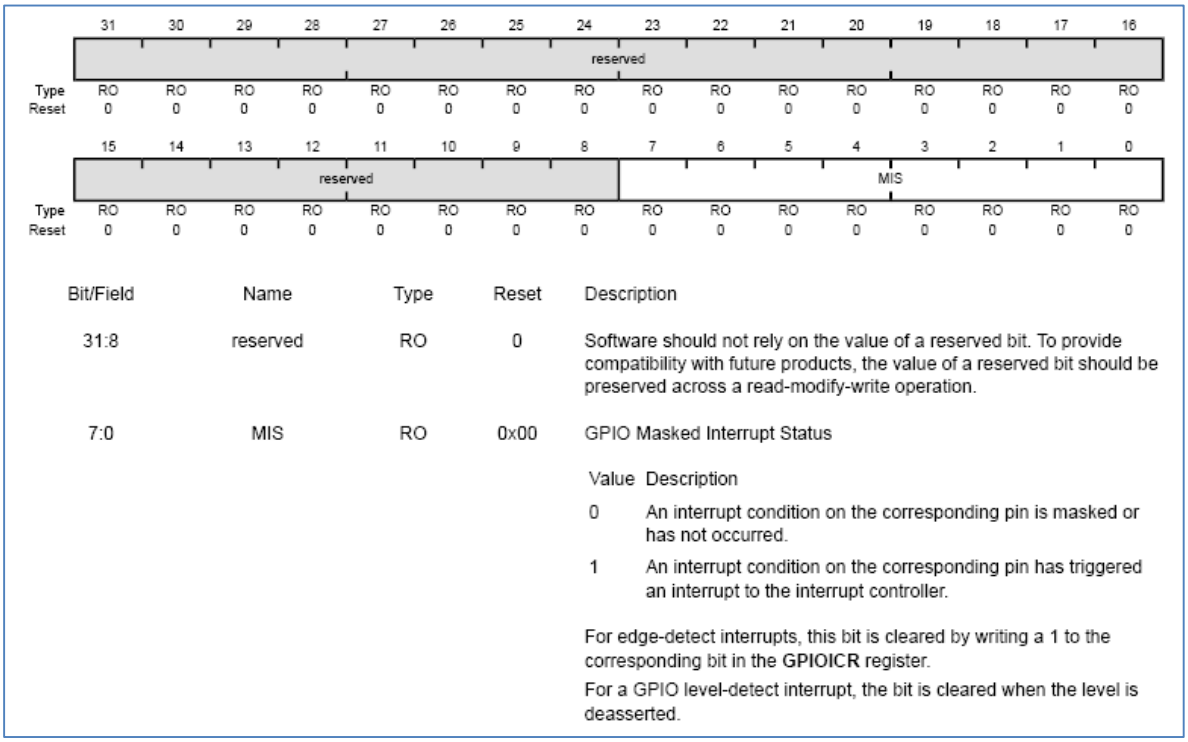
Figure 9-1 Interrupt enable with all three levels

- Enable interrupt for a specific peripheral module using interrupt mask (IM) register. In case of GPIO, GPIOIM is used. The lower 8 bits of 32 bit register are used to enable/disable for specific pin of port.

	D31 ..... D8	D7	D6	D5	D4	D3	D2	D1	D0	
GPIOIM:	Reserved	IME Px.7	IME Px.6	IME Px.5	IME Px.4	IME Px.3	IME Px.2	IME Px.1	IME Px.0	0x410

Note: D0 to D7 are used to enable/disable the interrupt for pins 0 to 7 of the port.
1: Enable interrupt
0: Disable interrupt (mask the interrupt)

Status of interrupt can be checked from GPIOMIS register.



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								MIS							
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

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	MIS	RO	0x00	GPIO Masked Interrupt Status

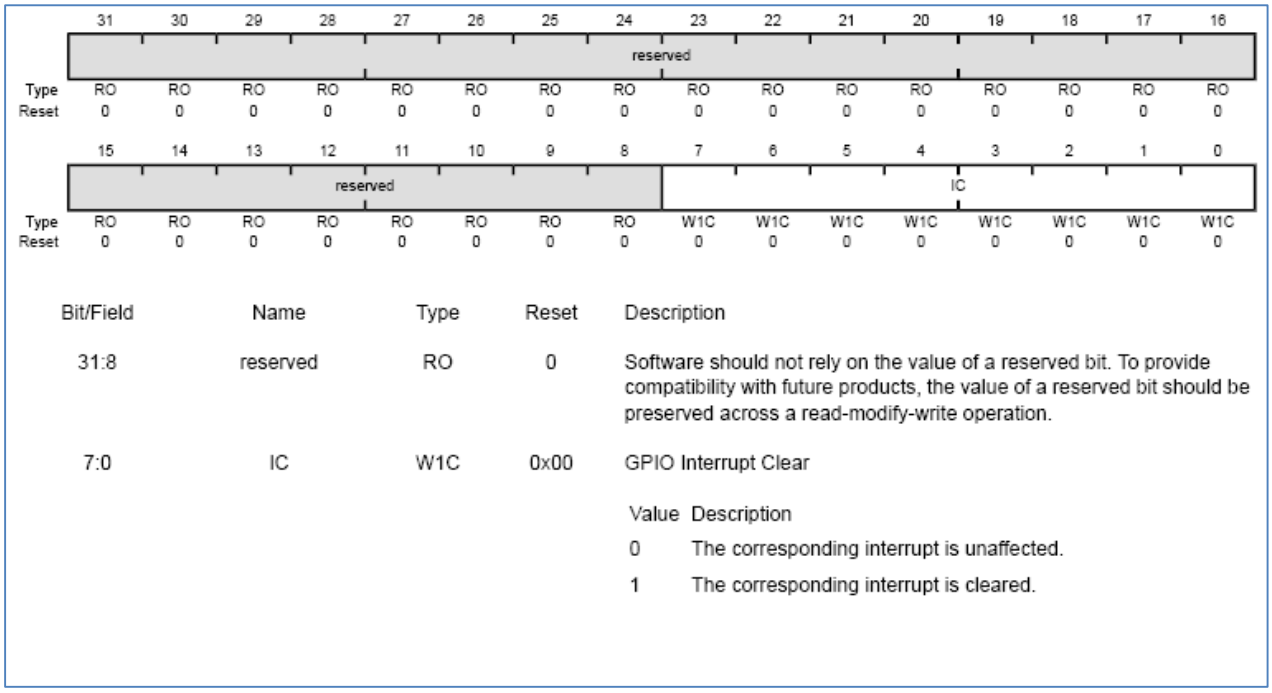
Value	Description
0	An interrupt condition on the corresponding pin is masked or has not occurred.
1	An interrupt condition on the corresponding pin has triggered an interrupt to the interrupt controller.

For edge-detect interrupts, this bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register.

For a GPIO level-detect interrupt, the bit is cleared when the level is deasserted.

- Enable interrupts at NVIC module. There is an interrupt enable for each entry in the interrupt vector table. These enable bits are in the registers EN0 to EN3 in NVIC. Each register convers 32 IRQ interrupts.
- Enable interrupt globally by assembly language instruction CPSID I and CPSIE I
- Priorities can be set to each interrupt before enabling them. For each IRQ number there is one byte corresponding to that IRQ to assign its priority. The allowed priority levels are ranging from 0 to 7 and they are defined by three bits left justified in that byte. PRIn register is used for this purpose.
- Interrupt trigger point for an I/O pin can be chosen from
    - Lowe level
    - High level
    - Rising edge
    - Falling edge
    - Both edges

  It can be chosen from GPIOIS and GPIOIEV registers. Edge/level sensitive can be chosen from IS register and falling/rising can be chosen form IEV register.
- ISR should clear respective bits in GPIOICR before exit, otherwise interrupt will be called again and again



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.

Interrupt Handler Routine:

- Open startup file
- Locate handler routine and rename it with your desired name
- Using “extern” mention that this routine is written in another code file (i.e., your main .c file)

LAB TASKS:

1. Implement a traffic control system. There is a single Traffic light whose lights turn on after every second. When traffic flow is less say after midnight, SW1 is pressed and yellow lights starts to blink with 1KHz rate. Another scenario is when traffic police takes over and turn off the traffic light. This scenario is triggered by pressing SW2. Write interrupt driven code for this problem. In order to go back to normal mode from any scenario, same switch is pressed again. Both switches are at the same priority level.
2. Modify code such that if both switches are pressed at the same time, interrupt of SW2 is handled first.

POST LAB:

Read about UART Interrupts. Modify post lab # 6 using interrupts.

EXPERIMENT 9
ANALOG TO DIGITAL CONVERTER

OBJECTIVE:
- Learn to configure ADC module of TM4C
- To learn interfacing sensors with TM4C

EQUIPMENT:
- Tiva C Launchpad
- Boosterpack for temperature sensor

BACKGROUND:
An analog to digital converter (ADC) converts an analog signal into digital form. An embedded system uses the ADC to collect information about the external world (data acquisition system.) The input signal is usually an analog voltage, and the output is a binary number. The ADC precision is the number of distinguishable ADC inputs (e.g., 4096 alternatives, 12 bits). The ADC range is the maximum and minimum ADC input (e.g., 0 to +3.3V). The ADC resolution is the smallest distinguishable change in input (e.g., 0.8 mV). The resolution is the change in input that causes the digital output to change by 1.

Formula to calculate output value of ADC with given input voltage (vin) and step size:
Dout= vin/stepsize
Stepsize=vref/2^n

ADC Module of TM4C:
TM4C has two ADC modules ADC0 and ADC1 with 12 channels each. Each ADC module has 12 bit resolution.

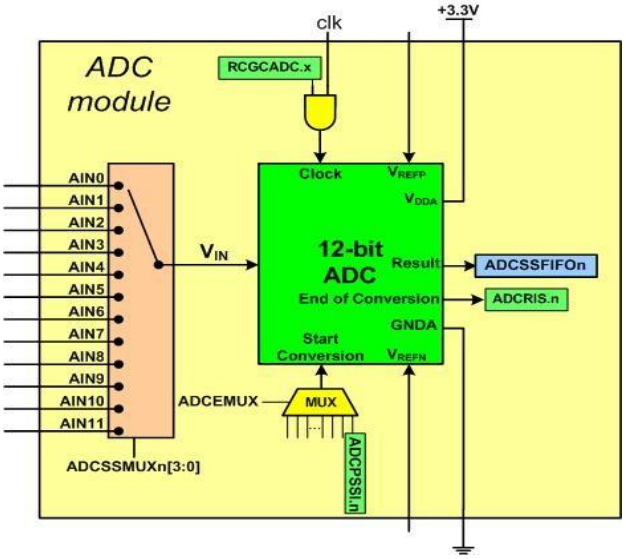


Figure 10-1 Block Diagram of ADC Module of TM4C

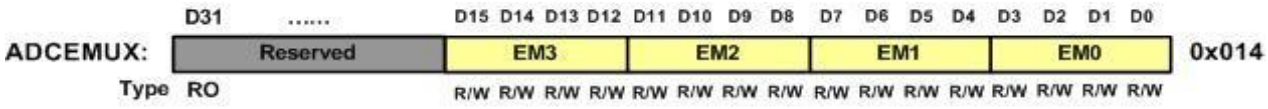
The ADC has four sample sequencers (SS3, SS2, SS1, SS0) that move the conversion result of the ADC to one of the FIFOs.
We perform the following steps to software start the ADC and sample one channel It will sample one channel using software start and busy-wait synchronization.

1. We enable the port clock for the pin that we will be using for the ADC input.
2. Make that pin an input by writing zero to the **DIR** register.
3. Enable the alternative function on that pin by writing one to the **AFSEL** register.
4. Disable the digital function on that pin by writing zero to the **DEN** register.
5. Enable the analog function on that pin by writing one to the **AMSEL** register.
6. We enable the ADC clock by setting bit 0 of the **SYSCTL_RCGCADC_R** register.

bit	Name	Description
0	R0	0: ADC module 0 is disabled, 1: Enable and provide a clock to ADC module 0
1	R1	0: ADC module 1 is disabled, 1: Enable and provide a clock to ADC module 1

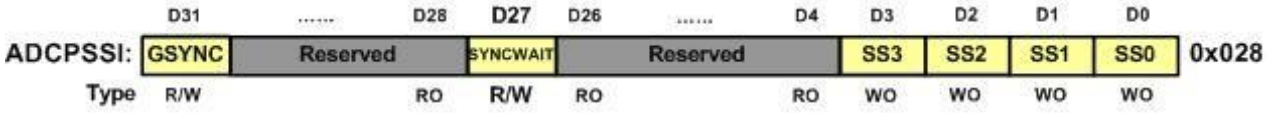
7. Before configuring the sequencer, we need to disable it. To disable sequencer 3, we write a 0 to bit 3 (**ASEN3**) in the **ADC0_ACTSS_R** register. Disabling the sequencer during programming prevents erroneous execution if a trigger event were to occur during the configuration process.

8. We configure the trigger event for the sample sequencer in the **ADC0_EMUX_R** register. For this example, we write a 0000 to bits 15–12 (**EM3**) specifying software start mode for sequencer 3.



EMx bits select the trigger source for Sample Sequencer x. By default the field is 0x0 which means the ADC conversion begins when the SSn bit of the ADCPSSI register is set by software. The following table shows the available choices for trigger. For more information see the datasheet.

EMx value	Trigger source
0x0	Processor (default)
0x1	Analog Comparator 0
0x2	Analog Comparator 1
0x3	reserved
0x4	External (GPIO Pins)
0x5	Timer
0x6	PWM generator 0
0x7	PWM generator 1
0x8	PWM generator 2
0x9	PWM generator 3
0xF	Always (continuously sample)



SSx value	Trigger source
0	No effect
1	Begin sampling on sample sequence x

9. For each sample in the sample sequence, configure the corresponding input source in the **ADCSSMUXn** register. In this example, we write the channel number to bits 3–0 in the **ADC0_SSMUX3_R** register.

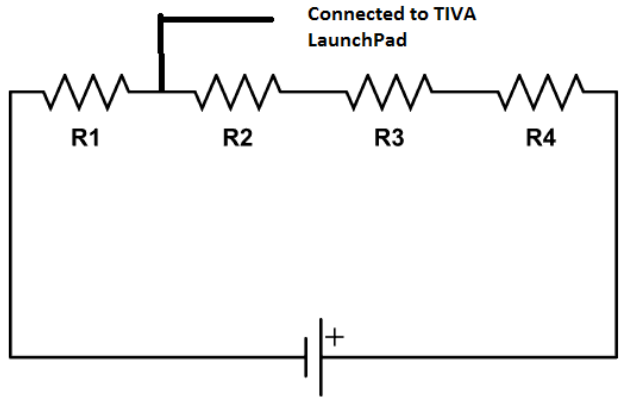
bit	Name	Description
0-3	MUX0	Sample Input Select

Pin Name	Description	Pin	Pin Number
AIN0	ADC input 0	PE3	6
AIN1	ADC input 1	PE2	7
AIN2	ADC input 2	PE1	8
AIN3	ADC input 3	PE0	9
AIN4	ADC input 4	PD3	64
AIN5	ADC input 5	PD2	63
AIN6	ADC input 6	PD1	62

AIN7	ADC input 7	PD0	61
AIN8	ADC input 8	PE5	60
AIN9	ADC input 9	PE4	59
AIN10	ADC input 10	PB4	58
AIN11	ADC input 11	PB5	57

LAB TASKS:

Implement a voltmeter using TM4C using its ADC module. Read input voltage from any of its channels and send its value (upto 2 decimal places) to PC via UART0. In order to test your code, implement a simple series resistive circuit on breadboard and read voltages of each node. Compare your results of your code with theoretical calculations.



POST LAB: Write a subroutine to configure ADC interrupt.

EXPERIMENT 10
PULSE WIDTH MODULATION

OBJECTIVE:

- To Learn to configure PWM module of TM4C
- To learn generating tones of different frequencies using PWM

EQUIPMENT:

- Tiva C Launchpad
- Boosterpack for buzzer

BACKGROUND:

Pulse Width Modulation allows to generate a wave of fixed frequency but of varying duty cycle. The waveforms that are high for **H** cycles and low for **L** cycles. The system is designed in such a way that **H+L** is constant (meaning the frequency is fixed). The duty cycle is defined as the fraction of time the signal is high: Hence, duty cycle varies from 0 to 1. We interface this digital output wave to an external actuator (like a DC motor), such that power is applied to the motor when the signal is high, and no power is applied when the signal is low. We purposely select a frequency high enough so the DC motor does not start/stop with each individual pulse, but rather responds to the overall average value of the wave. The average value of a PWM signal is linearly related to its duty cycle and is independent of its frequency.

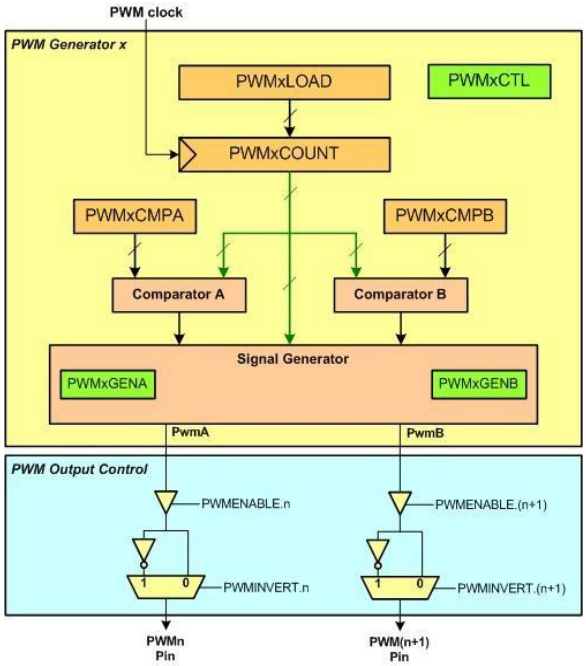


Figure 11-1 PWM Generator of TM4C

Programming PWM Module of TM4C

There are two on chip PWM modules on TM4C: PWM0 and PWM1. Each module has four generators. Each generator has a counter (timer). Each generator has two compare registers CMPA and CMPB. Each generator has two output pins, which means there are total of 8 PWM pins per module. The counter of generator can be programmed as up or down counter. A simplified block diagram of generator is given in figure 11-1.

Steps involved in programming PWM Module

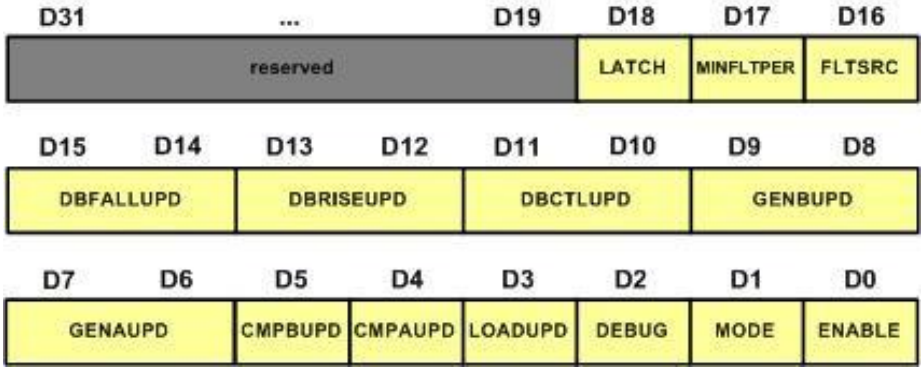
1. Configure PWM Clock. Enable clock to PWM module by setting respective pin in PCGCPWM register. R1 is for Module 1 and R0 is for module 0.

2. The next step is to set the clock frequency that is fed to the PWM module. System clock can be provided as it is or can be divided by 2, 4, 8, 16,32 or 64(default). RCC register is used for this purpose.

bit	Name	Description
19-17	PWMDIV	PWM Unit Clock Divisor: The system clock is divided by $2^{PWMDIV+1}$ when the USEPWMDIV bit is set to one.
20	USEPWMDIV	Enable PWM Clock Divisor (Use PWM clock Divisor) 0: The PWM clock divider is by passed, 1: The PWM clock divider is used as the PWM clock source.

PWMDIV value	0	1	2	3	4	5	6	7
Division	Clk/2	Clk/4	Clk/8	Clk/16	Clk/32	Clk/64	Clk/64	Clk/64

3. PWM generator can be enabled by PWMxCTL register which is available for *each* generator.



D0 bit is the enable bit. Mode bit is the mode selection. There are two modes: count down, count up/down. In both modes, the load value is in a register LOAD (PWMxLOAD). It contains the maximum value of the count. Current counter value can be read from PWMxCOUNT register. There are two more registers: CompareA and CompareB. The relation between Load, count and compare registers can be seen from the following figure:

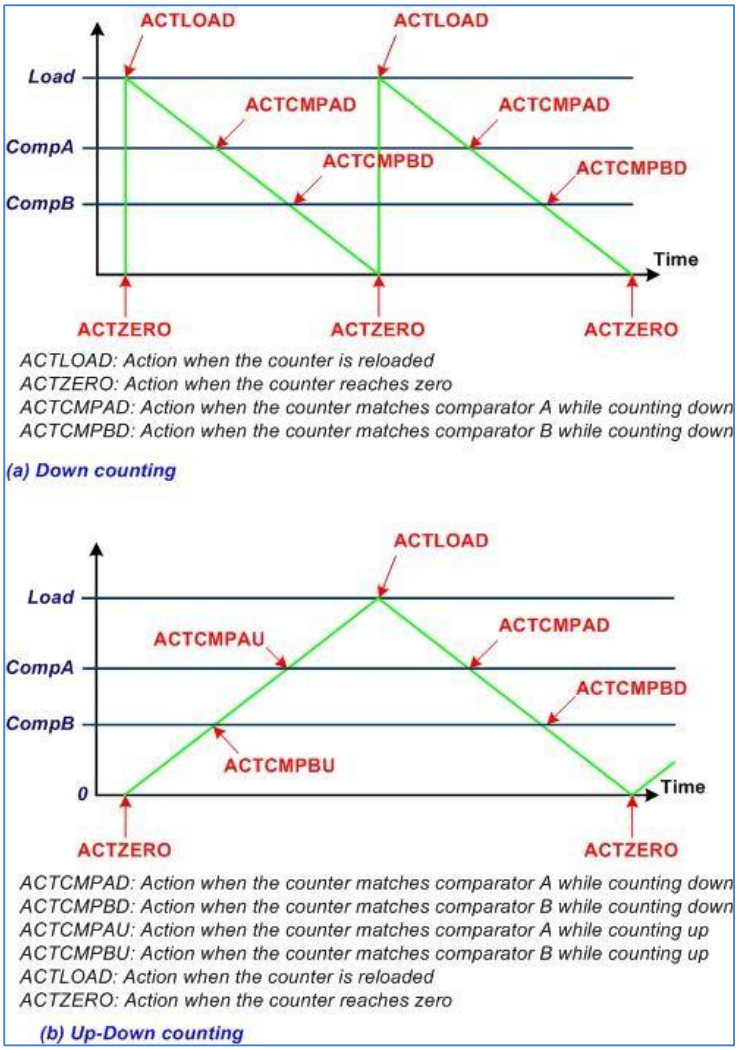


Figure 11-2 PWM events

LAB TASKS:

Generate "Happy Birthday" Tone using PWM generator of TM4C. There are 25 notes (frequencies with 50% DC) in this tone. kth note is to be played for interval [k] * 100 milli seconds. Use buzzer of Booster pack which connects with PF2 of TM4C.

```
// Happy birthday notes
/*                  Hap py  Birth Day  to  you,  Hap py  birth day  to
                   C4   C4   D4   C4   F4   E4   C4   C4   D4   C4   G4 */
unsigned int notes[] = {262, 262, 294, 262, 349, 330, 262, 262, 294, 262, 392,

/*                  you, Hap py  Birth Day  dear  xxxx     Hap  py   birth
                   F4   C4   C4   C5   A4   F4   E4   D4   B4b  B4b  A4 */
                  349, 262, 262, 523, 440, 349, 330, 294, 466, 466, 440,

/*                  day  to  you
                   F4   G4   F4   */
                  349, 392, 349
                  };
```

```
unsigned short interval[] = {4, 4, 8, 8, 8, 10, 4, 4, 8, 8, 8, 10, 4, 4, 8, 8, 8,
                        8, 8, 4, 4, 8, 8, 8, 12};
```

POST LAB

Generate a PWM wave whose duty cycle increases by 10% on pressing SW1. Chose any PWM channel which is connected with LED of Launchpad.

EXPERIMENT 11

SYSTICK TIMER and PLL ON TIVA TM4C123GH6PM MICROCONTROLLER

OBJECTIVE:

- To learn configuring SysTick Timer
- To learn tuning the frequency of oscillator using PLL

BACKGROUND:

Phase Locked Loop

Normally, an external crystal determines the execution speed of a microcontroller. Most microcontrollers have a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the bus clock will require less power to operate and generate less heat. Speeding up the bus clock obviously allows for more calculations per second. The default bus speed of TM4C microcontrollers is that of the internal oscillator, also meaning that the PLL is not initially active. For example, the default bus speed for the TM4C internal oscillator is 16 MHz ±1%. The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal. This means for most applications we will activate the main oscillator and the PLL so we can have a stable bus clock.

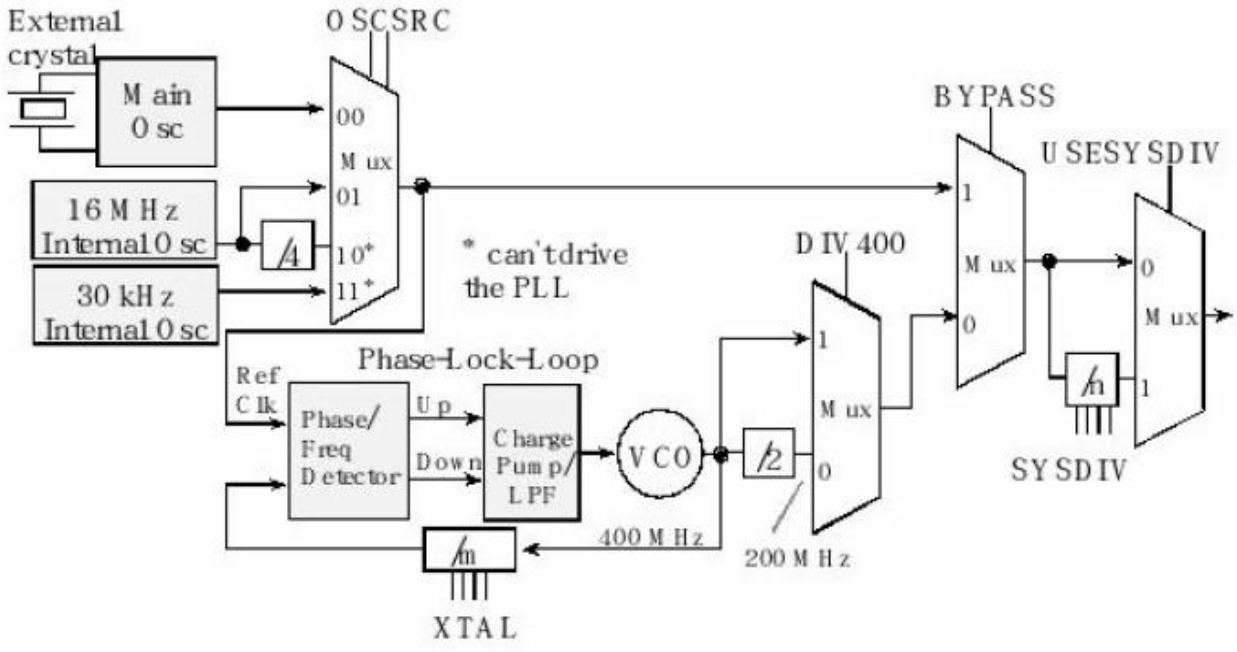


Figure 10.1 Block Diagram of the main clock tree on TM4C including the PLL

How PLL works:

An external crystal is attached to the TM4C microcontroller, as shown in Figure 10.1. The PLLs on the other Tiva microcontrollers operate in the same basic manner. Figure 10.2 shows the clock registers used to define what speed the processor operates. The output of the main oscillator (Main Osc) is a clock at the same frequency as the crystal. By setting the OSCSRC bits to 0, the

multiplexer control will select the main oscillator as the clock source. Use RCC2 because it provides for more options.

1. The first step is set BYPASS2 (bit11). At this point, the PLL is bypassed and there is no system clock divider.
2. The second step is to specify the crystal frequency in the four XTAL bits using the code in Figure 10.2.
3. The OSCSRC2 bits are cleared to select the main oscillator as the oscillator clock source.
4. Clear PWRDN2 (bit 13) to activate the PLL.
5. Configure and enable the clock divider using the 7-bit SYSDIV2 field. If the 7-bit SYSDIV2 is n, then the clock will be divided by n+1. E.g., To get the desired 80 MHz from the 400 MHz PLL, we need to divide by 5. So, we place a 4 into the SYSDIV2 field.
6. Wait for the PLL to stabilize by waiting for PLLRIS (bit 6) in the SYSCTL_RIS_R to become high.
7. Connect the PLL by clearing the BYPASS2 bit.

XTAL	Crystal Freq (MHz)	XTAL	Crystal Freq (MHz)
0x0	Reserved	0x10	10.0 MHz
0x1	Reserved	0x11	12.0 MHz
0x2	Reserved	0x12	12.288 MHz
0x3	Reserved	0x13	13.56 MHz
0x4	3.579545 MHz	0x14	14.31818 MHz
0x5	3.6864 MHz	0x15	16.0 MHz
0x6	4 MHz	0x16	16.384 MHz
0x7	4.096 MHz	0x17	18.0 MHz
0x8	4.9152 MHz	0x18	20.0 MHz
0x9	5 MHz	0x19	24.0 MHz
0xA	5.12 MHz	0x1A	25.0 MHz
0xB	6 MHz (reset value)	0x1B	Reserved
0xC	6.144 MHz	0x1C	Reserved
0xD	7.3728 MHz	0x1D	Reserved
0xE	8 MHz	0x1E	Reserved
0xF	8.192 MHz	0x1F	Reserved

Figure 10.2 values of XTAL for different crystal frequencies

Registers used for setting PLL:

Address	26-23	22	13	11	10-6	5-4	Name
$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
$400FE050					PLLRIS		SYSCTL_RIS_R
	31	30	28-22	13	11	6-4	
$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

Example code is given below that activates a microcontroller with a 16MHz frequency to run at 80MHz. To modify this program to operate on other microcontrollers, you will need to change the crystal frequency and the system clock divider.

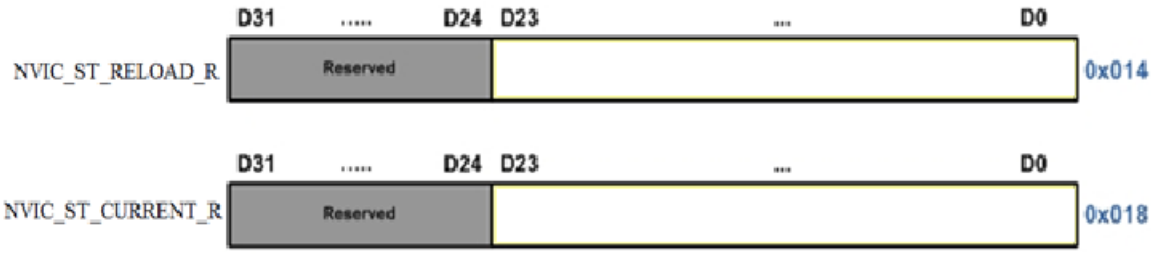
```
#define SYSDIV2 4
void PLL_Init(void){
// 0) Use RCC2 c
SYSCTL_RCC2_R |= 0x80000000; // USERCC2
// 1) bypass PLL while initializing
SYSCTL_RCC2_R |= 0x00000800; // BYPASS2, PLL bypass
// 2) select the crystal value and oscillator source
SYSCTL_RCC_R = (SYSCTL_RCC_R & ~0x000007C0) // clear bits 10-6
+ 0x00000540; // 10101, configure for 16 MHz crystal
SYSCTL_RCC2_R &= ~0x00000070; // configure for main oscillator
source
// 3) activate PLL by clearing PWRDN
SYSCTL_RCC2_R &= ~0x00002000;
// 4) set the desired system divider
SYSCTL_RCC2_R |= 0x40000000; // use 400 MHz PLL
SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~0x1FC00000)+(SYSDIV2<<22); // 80
MHz
// 5) wait for the PLL to lock by polling PLLLRIS
while((SYSCTL_RIS_R&0x00000040)==0){}; // wait for PLLRIS bit
// 6) enable use of PLL by clearing BYPASS
SYSCTL_RCC2_R &= ~0x00000800;
}
```

SysTick Timer

SysTick timer is a simple counter that we can use to create time delays and generate periodic interrupts. The basis of SysTick is a 24-bit down counter that runs on a fixed rate without external signal (either by the system clock or internal oscillator). It counts down from an initial value to 0. When it reaches to 0 in the next clock, it underflows and raises a flag called COUNT and reloads the initial value and starts all over. Initial value can be between 0x000000 to 0xFFFFFF.

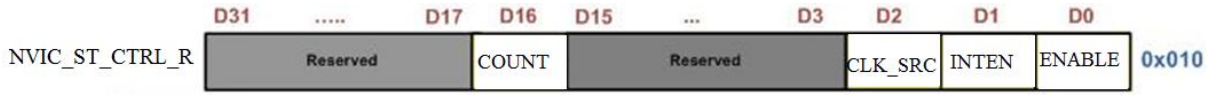
There are four steps to initialize the SysTick timer.

1. First, we clear the **ENABLE** bit (NVIC_ST_CTRL_R**)** to turn off SysTick during initialization.
2. Second, we set the **RELOAD** register with the start value of counter. STRELOAD should contain N-1 for the COUNT to fire every N clock cycles because the counter counts down to 0.



3. Third, we write to the **NVIC_ST_CURRENT_R** value to clear the counter.

4. Lastly, we write the desired mode to the control register, **NVIC_ST_CTRL_R** . We set the **CLK_SRC** bit specifying the core clock will be used. We must set **CLK_SRC**=1, because **CLK_SRC**=0 external clock mode is not implemented on the TM4C family.



5. We will set **INTEN** to enable interrupts, but in this first example we clear **INTEN** so interrupts will not be requested. We need to set the **ENABLE** bit so the counter will run. When the **CURRENT** value counts down from 1 to 0, the **COUNT** flag is set. On the next clock, the **CURRENT** is loaded with the **RELOAD** value. In this way, the SysTick counter (**CURRENT**) is continuously decrementing. If the **RELOAD** value is n, then the SysTick counter operates at modulo n+1 (…n, n-1, n-2 … 1, 0, n, n- 1, …). In other words, it rolls over every n+1 count. The **COUNT** flag could be configured to trigger an interrupt.

If we activate the PLL to run the microcontroller at 80 MHz, then the SysTick counter decrements every 12.5 ns otherwise it will decrements every **62.5 ns** with 16MHz core clock. In general, if the period of the core bus clock is t, then the **COUNT** flag will be set every **(n+1)×t**. Reading the **NVIC_ST_CTRL_R** control register will return the **COUNT** flag in bit 16 and then clear the flag. Also, writing any value to the **NVIC_ST_CURRENT_R** register will reset the counter to zero and clear the **COUNT** flag.

Formula to load a value for required delay:

Required delay = (N+1)/ ClkFrequency

LAB TASKS:

1. Use systick timer to write a routine to add a delay of 10ms using main oscillator. Toggle PF2 every 10ms.
2. Using PLL, obtain 50MHz from main oscillator. And run the same systick routine you wrote in part 1. How much is the delay generated by systick timer now? Use logic analyzer to view the new waveform and calculate the delay of each half cycle.

POST LAB:

Use PLL to set frequency to 25MHz. Can you tell which modules you have studied so far are affected by enabling PLL?

EXPERIMENT 12

INTRODUCTION TO REAL TIME OPERATING SYSTEM

OBJECTIVE:

- To learn how to use FreeRTOS with TM4C

BACKGROUND:

FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller – although its use is not limited to microcontroller applications. FreeRTOS provides the core real time scheduling functionality, inter-task communication, timing and synchronization primitives only. This means it is more accurately described as a real time kernel, or real time executive. FreeRTOS can easily be used with any microcontroller such as TM4C. API can be referenced from https://www.freertos.org/a00106.html

FreeRTOS supports all general kernel objects including task, semaphore, queues, messages etc. The demo project we are going to use in this lab follows a structure where each and every task or object has its own header file. Different attributes are defined in their own header file. Main code has following format:

```
int main( void )
{
    /* Perform any hardware setup necessary. */
    prvSetupHardware();

    /* --- APPLICATION TASKS CAN BE CREATED HERE --- */

    /* Start the created tasks running. */
    vTaskStartScheduler();

    /* Execution will only reach here if there was insufficient heap to
    start the scheduler. */
    for( ;; );
    return 0;
}
```

Relevant kernel routines:

- **Creating Task:** Create a new task and add it to the list of tasks that are ready to run.

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           configSTACK_DEPTH_TYPE usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

Parameters:

pvTaskCode	Pointer to the task entry function (just the name of the function that implements the task, see the example below). Tasks are normally implemented as an infinite loop, and must never attempt to return or exit from their implementing function. Tasks can however delete themselves.
pcName	A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used to obtain a task handle. The maximum length of a task's name is set using the configMAX_TASK_NAME_LEN parameter in FreeRTOSConfig.h.
usStackDepth	The number of words (not bytes!) to allocate for use as the task's stack. For example, if the stack is 16-bits wide and usStackDepth is 100, then 200 bytes will be allocated for use as the task's stack. As another example, if the stack is 32-bits wide and usStackDepth is 400 then 1600 bytes will be allocated for use as the task's stack. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.
pvParameters	A value that will passed into the created task as the task's parameter. If pvParameters is set to the address of a variable then the variable must still exist when the created task executes – so it is not valid to pass the address of a stack variable.
uxPriority	The priority at which the created task will execute. Systems that include MPU support can optionally create a task in a privileged (system) mode by setting bit portPRIVILEGE_BIT in uxPrriority. For example, to create a privileged task at priority 2 set uxPriority to (2 | portPRIVILEGE_BIT).
pxCreatedTask	Used to pass a handle to the created task out of the xTaskCreate() function. pxCreatedTask is optional and can be set to NULL.

Returns:

If the task was created successfully then pdPASS is returned. Otherwise errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY is returned.

- **Delaying a Task:** Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate – with the resolution of one tick period

```
void vTaskDelay( const TickType_t xTicksToDelay );
```

Parameters:

xTicksToDelay	The amount of time, in tick periods, that the calling task should block.

Example usage:

```
void vTaskFunction( void * pvParameters )
{
/* Block for 500ms. */
const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    for( ;; )
    {
        /* Simply toggle the LED every 500ms, blocking between each toggle. */
        vToggleLED();
        vTaskDelay( xDelay );
    }
}
```

EXERCISE – 1:

We can create a new project from scratch but for now, just modify the demo project given in TIVA ware located at
C:\ti\TivaWare_C_Series-2.1.4.178\examples\boards\ek-tm4c123gxl\freertos_demo

Run this project and download it to TM4C controller. Observe the output.

Details of demo project:

The project contains header files of all kernel objects as well has header files of application tasks. For the sake of understanding, you may run the project as it is or it can be edited according to your need.

The application blinks the user-selected LED at a user-selected frequency. To select the LED press the left button and to select the frequency press the right button. The UART outputs the application status at 115,200 rate. This application utilizes FreeRTOS to perform the tasks in a concurrent fashion. Following tasks are created:
An LED task, which blinks the user-selected on-board LED at a user-selected rate (changed via the buttons).

A Switch task, which monitors the buttons pressed and passes the information to LED task.

In addition to the tasks, this application also uses the following FreeRTOS resources:

- A Queue to enable information transfer between tasks.
- A Semaphore to guard the resource, UART, from access by multiple tasks at the same time.
- A non-blocking FreeRTOS Delay to put the tasks in blocked state when they have nothing to do.

EXERCISE – 2:

Replace the main code with the code given to you and remove led_task.c and switch_task.c from project. In the given file there is one task defined to blink Blue LED every 500ms. Define another similar task which blinks Red LED every 100 ms. Keep its priority and other parameters same as that of the first task.

POST LAB:

How many priorities can be assigned to tasks in freeRTOS and what is the value of the lowest and highest priority?

EXPERIMENT 13

USING SEMAPHORES IN REAL TIME OPERATING SYSTEMS

OBJECTIVE:

- To learn about signaling between tasks in FreeRTOS

BACKGROUND:

Define semaphore in FreeRTOS
Share a resource between two tasks using mutex semaphore
Sample Code:
In following code, a counter (Task1) runs every 500 msec and gives a signal to semaphore associated with Task2. Task2 waits for the signal from Task1 and on receivingsignal, it performs its required function. Semaphore is created in main.

```
#include <p18f452.h>
#pragma config OSC = XT
#pragma config WDT = OFF
#pragma config LVP = OFF
#include <salvo.h>

#define BINSEM OSECBP(1)

//Define labels for context switches
_OSLabel(Task1_Label)
_OSLabel(Task2_Label)

unsigned char count;
/*Timer0 Interrupt*/
#pragma interrupt chk_isr

void chk_isr (void)
{
if (INTCONbits.TMR0IF==1)
{
INTCONbits.T0IF=0;
TMR0H=0xFD;
TMR0L=0x96;
OSTimer();
}
}

#pragma code myISR=0x0008
void myISR(void)
{
_asm
GOTO chk_isr
_endasm
}
#pragma code

void Init(void)
{
T0CON=0x83;
INTCONbits.TMR0IE=1;
INTCONbits.PEIE=1;
INTCONbits.GIE=1;
TMR0H=0xFD;
TMR0L=0x96;
count=0;
```

```
}
/*************************************************************************
Task Definitions (configured as functions)
*************************************************************************/
void Task1( void )
{
for (;;) //infinite loop
{
count=count+1;
OS_Delay (50,Task1_Label); //Task switch, delay for 50x10ms, (500ms)
OSSignalBinSem(BINSEM);
}
}
void Task2( void )
{
for (;;) //infinite loop
{
OS_WaitBinSem(BINSEM,OSNO_TIMEOUT , Task2_Label);
PORTB=count;
OS_Yield(Task2_Label);

}
}
/*************************************************************************
Main
*************************************************************************/
void main( void )
{
//Initialise
TRISB =0x00;

//Initialise the RTOS
Init();
OSInit();
//Create Tasks
OSCreateTask(Task1, OSTCBP(1), 10);
OSCreateTask(Task2, OSTCBP(2), 10);
//Set up continuous loop, within which scheduling will take place.
OSCreateBinSem(BINSEM, 0); //Create the Binary Semaphore
while(1)
OSSched();

}
```

LAB TASK:

A voltage source is connected to AN0 of PIC18. Read this analog value and perform following operations:

If (voltage < 2.5) Led connected to RB0 is ON else OFF
If (voltage ≥ 2.5) Led connected to RB1 is ON else OFF

Write two tasks:

- TASK-1: for reading analog voltage from AN0 (LOW PRIORITY)
- TASK-2: turn on respective Led depending on the voltage value (HIGH PRIORITY)

Appendix A: Lab Evaluation Criteria

Labs with projects

1. Experiments and their report 50%
    a. Experiment 60%
    b. Lab report 40%
2. Quizzes (3-4) 15%
3. Final evaluation 35%
    a. Project Implementation 60%
    b. Project report and quiz 40%

Labs without projects

1. Experiments and their report 50%
    a. Experiment 60%
    b. Lab report 40%
2. Quizzes (3-4) 20%
3. Final Evaluation 30%
    i. Experiment 60%
    ii. Lab report, pre and post experiment quiz 40%

Notice:

Copying and plagiarism of lab reports is a serious academic misconduct. First instance of copying may entail ZERO in that experiment. Second instance of copying may be reported to DC. This may result in awarding FAIL in the lab course.

Appendix B: Safety around Electricity

In all the Electrical Engineering (EE) labs, with an aim to prevent any unforeseen accidents during conduct of lab experiments, following preventive measures and safe practices shall be adopted:

- Remember that the voltage of the electricity and the available electrical current in EE labs has enough power to cause death/injury by electrocution. It is around 50V/10 mA that the "cannot let go" level is reached. "The key to survival is to decrease our exposure to energized circuits."
- If a person touches an energized bare wire or faulty equipment while grounded, electricity will instantly pass through the body to the ground, causing a harmful, potentially fatal, shock.
- Each circuit must be protected by a fuse or circuit breaker that will blow or "trip" when its safe carrying capacity is surpassed. If a fuse blows or circuit breaker trips repeatedly while in normal use (not overloaded), check for shorts and other faults in the line or devices. Do not resume use until the trouble is fixed.
- It is hazardous to overload electrical circuits by using extension cords and multi-plug outlets. Use extension cords only when necessary and make sure they are heavy enough for the job. Avoid creating an "octopus" by inserting several plugs into a multi-plug outlet connected to a single wall outlet. Extension cords should ONLY be used on a temporary basis in situations where fixed wiring is not feasible.
- Dimmed lights, reduced output from heaters and poor monitor pictures are all symptoms of an overloaded circuit. Keep the total load at any one time safely below maximum capacity.
- If wires are exposed, they may cause a shock to a person who comes into contact with them. Cords should not be hung on nails, run over or wrapped around objects, knotted or twisted. This may break the wire or insulation. Short circuits are usually caused by bare wires touching due to breakdown of insulation. Electrical tape or any other kind of tape is not adequate for insulation!
- Electrical cords should be examined visually before use for external defects such as: Fraying (worn out) and exposed wiring, loose parts, deformed or missing parts, damage to outer jacket or insulation, evidence of internal damage such as pinched or crushed outer jacket. If any defects are found the electric cords should be removed from service immediately.
- Pull the plug not the cord. Pulling the cord could break a wire, causing a short circuit.
- Plug your heavy current consuming or any other large appliances into an outlet that is not shared with other appliances. Do not tamper with fuses as this is a potential fire hazard. Do not overload circuits as this may cause the wires to heat and ignite insulation or other combustibles.
- Keep lab equipment properly cleaned and maintained.
- Ensure lamps are free from contact with flammable material. Always use lights bulbs with the recommended wattage for your lamp and equipment.
- Be aware of the odor of burning plastic or wire.
- ALWAYS follow the manufacturer recommendations when using or installing new lab equipment. Wiring installations should always be made by a licensed electrician or other qualified person. All electrical lab equipment should have the label of a testing laboratory.
- Be aware of missing ground prong and outlet cover, pinched wires, damaged casings on electrical outlets.
- Inform Lab engineer / Lab assistant of any failure of safety preventive measures and safe practices as soon you notice it. Be alert and proceed with caution at all times in the laboratory.

- Conduct yourself in a responsible manner at all times in the EE Labs.
- Follow all written and verbal instructions carefully. If you do not understand a direction or part of a procedure, ASK YOUR LAB ENGINEER / LAB ASSISTANT BEFORE PROCEEDING WITH THE ACTIVITY.
- Never work alone in the laboratory. No student may work in EE Labs without the presence of the Lab engineer / Lab assistant.
- Perform only those experiments authorized by your teacher. Carefully follow all instructions, both written and oral. Unauthorized experiments are not allowed.
- Be prepared for your work in the EE Labs. Read all procedures thoroughly before entering the laboratory. Never fool around in the laboratory. Horseplay, practical jokes, and pranks are dangerous and prohibited.
- Always work in a well-ventilated area.
- Observe good housekeeping practices. Work areas should be kept clean and tidy at all times.
- Experiments must be personally monitored at all times. Do not wander around the room, distract other students, startle other students or interfere with the laboratory experiments of others.
- Dress properly during a laboratory activity. Long hair, dangling jewelry, and loose or baggy clothing are a hazard in the laboratory. Long hair must be tied back, and dangling jewelry and baggy clothing must be secured. Shoes must completely cover the foot.
- Know the locations and operating procedures of all safety equipment including fire extinguisher. Know what to do if there is a fire during a lab period; "Turn off equipment, if possible and exit EE lab immediately."

Appendix C: Guidelines on Preparing Lab Reports

Each student will maintain a lab notebook for each lab course. He will write a report for each experiment he performs in his notebook. A format has been developed for writing these lab reports.

Lab Report Format

1. **Introduction**: Introduce the new constructs/ commands being used, and their significance.
2. **Objective**: What are the learning goals of the experiment?
3. **Design**: How do the new constructs facilitate achievement of the objectives; if possible, a comparison in terms of efficacy and computational tractability with the alternate constructs? Include the circuit diagram and code with explanation.
4. **Issues**: The bugs encountered and the way they were removed.
5. **Conclusions**: What conclusions can be drawn from experiment?
6. **Application**: Suggest a real world application where this exercise may apply.
7. Answers to post lab questions (if any).

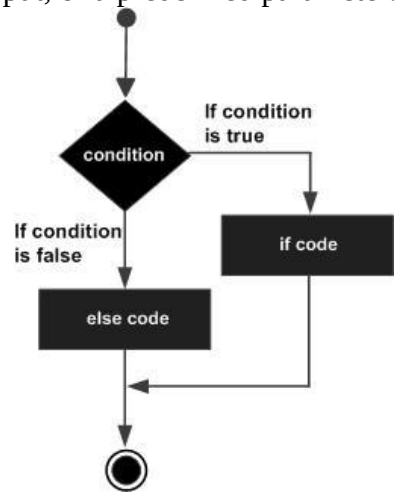
Sample Lab Report for Programming Labs

Introduction

The ability to control the flow of the program, letting it make decisions on what code to execute, is important to the programmer. The if-else statement allows the programmer to control if a program enters a section of code or not based on whether a given condition is true or false. If-else statements control *conditional branching*.

```
if ( expression )
  statement1
else
  statement2
```

If the value of *expression* is nonzero, *statement1* is executed. If the optional **else** is present, *statement2* is executed if the value of *expression* is zero. In this lab, we use this construct to select an action based upon the user's input, or a predefined parameter.



Objective:

To use if-else statements for facilitation of programming objectives: A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. We have written a C++ program that reads in a five-digit integer and determines whether it is a palindrome.

Design:

Code with explanation:

The objective was achieved with the following code:

```
#include<iostream>

Usingnamespacestd;
Intmain()
{
    inti,temp,d,revrs=0;

    cout<<"enter the number to check :";
        cin>>i;
    temp=i;
    while(temp>0)
    {
        d=temp%10;
        temp/=10;
        revrs=revrs*10+d;

        }
    if(revrs==i)
    cout<<i<<" is palindorme";
    else
    cout<<i<<" is not palindrome";

    }
}
```

Screen shots of the output for various inputs are shown in Figure 2:

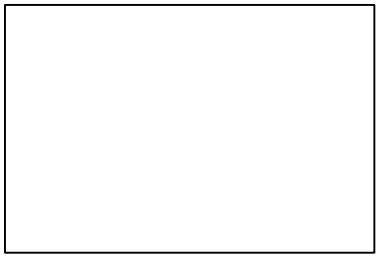


Fig.2. Screen shot of the output

The conditional statement made this implementation possible; without conditional branching, it is not possible to achieve this objective.

Issues:

Encountered bugs and issues; how were they identified and resolved.

Conclusions:

The output indicates correct execution of the code.

Applications:

If-else statements are a basic construct for programming to handle decisions.

Answers of Post Lab Questions: