Dhananjay V. Gadre
Sarthak Gupta

Getting Started with Tiva ARM Cortex M4 Microcontrollers

A Lab Manual for Tiva LaunchPad Evaluation Kit

Springer

Getting Started with Tiva ARM Cortex M4 Microcontrollers

Dhananjay V. Gadre · Sarthak Gupta

Getting Started with Tiva ARM Cortex M4 Microcontrollers

A Lab Manual for Tiva LaunchPad Evaluation Kit

Springer

Dhananjay V. Gadre
Division of Electronics and Communications Engineering
Netaji Subhas Institute of Technology
New Delhi
India

Sarthak Gupta
TI Centre for Embedded Product Design
Netaji Subhas Institute of Technology
New Delhi
India

ISBN 978-81-322-3764-8 ISBN 978-81-322-3766-2 (eBook)
https://doi.org/10.1007/978-81-322-3766-2

Library of Congress Control Number: 2017948637

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer (India) Pvt. Ltd.
The registered company address is: 7th Floor, Vijaya Building, 17 Barakhamba Road, New Delhi 110 001, India

To Professor T. J. Joseph (Newman College, Thodupuza, Kerala). In solidarity.

–Dhananjay V. Gadre

To my grandparents, parents
and
my sisters who inspired me to work hard
and
give my best

–Sarthak Gupta

Acknowledgements

The need for this lab manual arose because Texas Instruments (TI) launched the new TIVA family of Cortex-M microcontroller platform after they obsoleted the Stellaris ARM microcontroller platform in 2012. With the launch of TIVA platform, corresponding LaunchPad evaluation kits were also made available. The LaunchPad evaluation kit is very versatile for hardware debugging; however, it lacks suitable sensors for physical interfacing that are critically required for a lab course. This led to the development of the Padma board as described in this text.

The following people extended great help in the development of the Padma Board and in completing this book: Dr. C. P. Ravikumar, the then director of university relations at Texas Instruments India; Sagar Juneja also at the Texas Instruments India office at the time together with Vaibhav Ostwal, were very efficient in catering to our needs whether they be of chip samples or evaluation kits. Since taking over the reins of the university relations program at Texas Instruments, India, Sanjay Srivastava and his team have been very helpful and responsive to our requirements. Thanks are due to several of my students at the Centre for Electronic Design and Technology (CEDT) at NSIT who helped in various forms through the development of this manuscript and before that during the design of the Padma Board, most notably Rohit Dureja and Anshuman Mishra.

I am proud of Sarthak Gupta who has done an admirable job as my co-author!

I thank Swati Meherishi, my editor at Springer for her continued support and persistence. She is a big reason for this book to see the light of the day!

And most of all, my wife Sangeeta and son Chaitanya for their love, care, and understanding without which I could not have started this project.

Dhananjay V. Gadre
New Delhi, India

Contents

About the Authors

Dhananjay V. Gadre (New Delhi, India) completed his M.Sc. (Electronic Science) from the University of Delhi and M.Engg. (Computer Engineering) from the University of Idaho, USA. In his professional career of more than 27 years, he has taught at the SGTB Khalsa College, University of Delhi, worked as a scientific officer at the Inter-University Centre for Astronomy and Astrophysics (IUCAA), Pune, and since 2001, has been with the Electronics and Communication Engineering Division, Netaji Subhas Institute of Technology (NSIT), New Delhi, currently as an Associate Professor. He directs two open access laboratories at NSIT, namely Centre for Electronics Design and Technology (CEDT) and TI Centre for Embedded Product Design (TI-CEPD). Professor Gadre is the author of several professional articles and five books. One of his books has been translated into Chinese and another one into Greek. His recent book "TinyAVR Microcontroller Projects for the Evil Genius", published by McGraw-Hill International, consists of more than 30 hands-on projects and has been translated into Chinese and Russian. He is a licensed radio amateur with a call sign VU2NOX and hopes to design and build an amateur radio satellite in the near future.

Sarthak Gupta completed his Bachelor of Engineering (Electronics and Communication Engineering) from the Netaji Subhas Institute of Technology (NSIT), Delhi in 2014. During college he worked extensively on embedded systems design using AVR- and ARM-based microcontrollers. He also worked on reconfigurable hardware like field-programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs). From June 2013 to May 2014, he worked as a Texas Instruments (India) intern at the TI Centre for Embedded Product Design (TI-CEPD) under the tutelage of Prof. Gadre. From July 2014 to December 2015, he worked at Texas Instruments (India) at their Bangalore office as a design engineer working on the design and verification of ARM Cortex M0+ based microcontrollers, memory controllers, and memory test chips. From July 2017, Sarthak plans to join the Indian Institute of Science (IISC), Bangalore as a postgraduate student pursuing Masters of Technology (M.Tech.) course in Electronic Systems Engineering.

Chapter 1
Introduction

This lab manual allows the user to get acquainted with the Tiva C Series microcontroller family based on ARM Cortex M4 through a hands-on approach by performing experiments on a hardware evaluation kit, namely, the EK-TM4C123GXL LaunchPad from Texas Instruments. Texas Instruments offers many Tiva-based LaunchPad kits, but for the remainder of the text, we will refer to this as the Tiva Launchpad. The Tiva Launchpad offers great value for money but as is the common feature of all the LaunchPad kits, there are not many user interfaces that a beginner could use to learn the various features of the microcontroller. To alleviate this problem, we have designed a motherboard with many popular input and output devices and connected them to a set of connectors that allows the user to plug the Tiva LaunchPad onto the motherboard using these connectors and get access to these peripheral devices. This motherboard is called Padmaboard.[1]

The experiments proposed in the manual are divided on basis of the microcontroller peripherals required to perform those experiments. The difficulty level of experiments increases as reader progresses through the manual. The later experiments will require combination of multiple microcontroller peripherals together to perform them. Apart from experiments mentioned in the manual, many more experiments can be performed by combination of various features of the PadmaBoard. Apart from performing academic experiments, PadmaBoard can also be used in initial prototyping and testing of many projects.

This manual also emphasizes on the development of standalone projects based on Tiva C Series microcontroller family. This includes the development of hardware as well as software part for any microcontroller of Tiva C series family.

[1] PadmaBoard is the peripheral motherboard of Tiva LaunchPad connected to it through its expansion headers. PadmaBoard is designed to enable the user to perform various experiments using Tiva LaunchPad.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_1

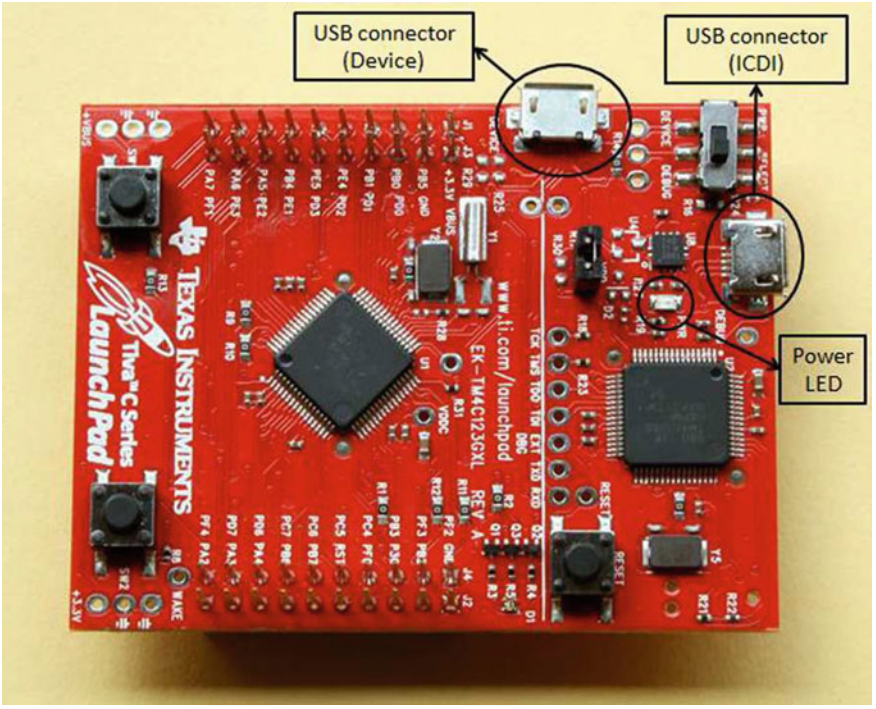

Fig. 1 Tiva LaunchPad

1 Tiva LaunchPad

Tiva C Series EK-TM4C123GXL LaunchPad Evaluation Kit from Texas Instruments is referred in this lab manual for the user to get acquainted with ARM cortex M4 Tiva C series microcontroller family. This evaluation kit includes:

1. Tiva LaunchPad Circuit Board (as shown in Fig. 1)
2. USB micro-B plug to USB-A plug cable

There are two USB micro-B connectors, one is for ICDI (In-circuit Debug Interface) for debug applications and another one is for the user to connect the Tiva microcontroller to external Host as a USB device. Connect USB cable to the ICDI USB Micro-B connector and make sure that the slide switch is push toward the Debug side. If the USB cable is connected to Device USB port and the slide switch is toward the Device side, it will still power the Tiva LaunchPad but user will not be able to program it directly. After connecting the USB cable with slide switch in correct position, yellow power LED will light up. This LED will remain ON indicating that LaunchPad is receiving power from source through USB port.

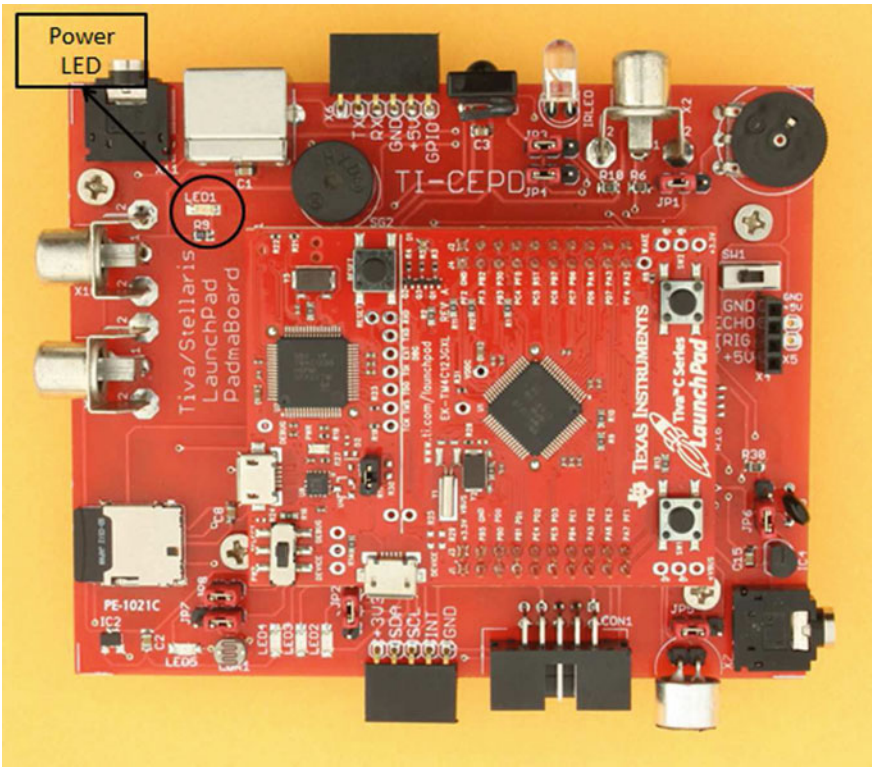

Fig. 2 Tiva LaunchPad with PadmaBoard

2 PadmaBoard

PadmaBoard is a peripheral motherboard of Tiva LaunchPad. This board is placed beneath the Tiva LaunchPad and connected to Tiva LaunchPad through its bottom side of expansion headers as shown in Fig. 2. This board allows the user to utilize various peripherals of microcontroller. The user can perform various experiments based on peripherals like Universal Asynchronous Receiver and Transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), and Analog-to-Digital Converter (ADC), which are not possible with just Tiva LaunchPad.

When PadmaBoard is connected to Tiva LaunchPad. Now power Tiva LaunchPad through USB port, then power LED LED1 on PadmaBoard will light up and will remain ON indicating that PadmaBoard is receiving power from Tiva LaunchPad.

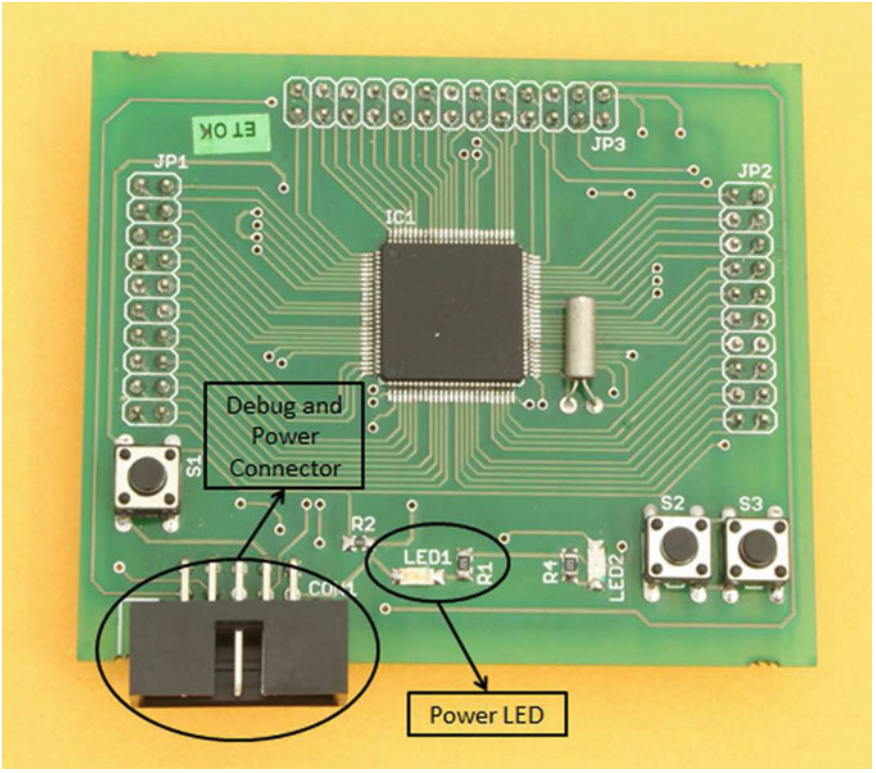

Fig. 3 Tiva TM4C1231H6PZ Microcontroller Breakout Board

3 Tiva C Series Microcontroller Breakout Board

To develop standalone projects on Tiva C Series microcontroller family, a breakout board of Tiva TM4C1231H6PZ[2] microcontroller is used. Figure 3 shows the breakout board. This board can be programmed (or debug) and powered through FRC connector. Also, breakout board can be powered through the expansion headers. Whenever the breakout board is powered up, power LED LED1 will light up and will remain ON indicating that it is receiving power from the source.

[2]TM4C1231H6PZ is a 100pin in LQFP (Low profile Quad Flat Package) microcontroller based on ARM Cortex-M4 processor belonging to Tiva C Series microcontroller family.

4 Look Ahead!

Once it is ensured that the kits (PadmaBoard and Tiva LaunchPad) are running properly, it is time to review the coverage of this lab manual.

Chapter 2 provides overview ARM cortex M4 architecture along with its peripherals. However it does not cover the detail coverage on ARM processors, for that reader is advised to go through the user guide of processor architecture from ARM. This chapter also includes brief description of peripherals of the microcontrollers belonging to Texas Instrument's Tiva C Series family.

Chapter 3 discuses about the Tiva C Series TM4C123G LaunchPad Evaluation Board along with its features like power supply, clock, reset, on board ICDI port, LEDs, switches, and expansion headers.

Chapter 4 discuses about the PadmaBoard (Peripheral Motherboard of Tiva C Series LaunchPad) in particular about various onboard features in terms of hardware circuit diagram and operation. Peripherals included are temperature sensor, audio input, light sensor, magnetic field sensor, UART connector (for Bluetooth, RS232, RS485 communication), ultrasonic sensor connector, IR transmitter and receiver, Serial LCD and 16 keys Keypad, SD card interface, TV and potentiometer, RTC, dual DAC and audio output, and I2C bus connector (for connecting other external I2C-based modules available in market).

Chapter 5 deals with the development of standalone ARM projects, implemented using Texas Instrument's Tiva C Series Microcontrollers family. Also includes various ways of programing TIVA C Series microcontrollers.

Chapter 6 deals with installation of software tools that are necessary to program Tiva LaunchPad or any Tiva C Series microcontrollers. This chapter illustrates the setup of a complete toolchain to program Tiva LaunchPad or any other Texas Instruments Tiva C series microcontrollers by using open-source tools.

Chapter 7 deals with the basic structure of embedded C programs. It also includes the basic programs for Tiva C Series microcontrollers using register access.

Chapter 8 deals with the driver library provided by Texas Instruments for their Tiva C Series microcontrollers. This driver library eases the programming of microcontrollers as user does not need to go into much details by using these libraries. Although this library contains a number of functions, but the basic functions that are most commonly used are illustrated in this chapter.

Chapter 9 deals with the control of digital input and output ports of Tiva C Series microcontrollers. Also, the chapter deals with peripherals which are controlled with digital IO pins such as LEDs, Switches, buzzer, and ultrasonic module.

Chapter 10 deals with the various interrupts sources offered by the Tiva C series microcontrollers. Also, this microcontroller family has a pretty nicely ordered interrupt structure and has large number of interrupt sources among which any digital pin can act as interrupt source. This chapter includes a description on using digital pin as an interrupt source.

Chapter 11 deals with the timers. Tiva C Series microcontrollers have 16-bit, 32-bit, and 64-bit timers. This chapter describes the various modes of timers in

which they can be operated in. Also deals with the generation of delays, PWM, interrupts, etc., by using timers.

Chapter 12 describes Universal Asynchronous Receiver and Transmitter (UART) peripheral on Tiva C Series microcontrollers, which is used to communicate with the host computer and other bluetooth devices.

Chapter 13 deals with the various modes of operation of microcontroller such as run mode, sleep mode, and deep-sleep mode for power saving.

Chapter 14 deals with the multichannel Analog-to-Digital Converter (ADC) present on Tiva C Series microcontrollers. This chapter also describes the various peripherals which provide the analog output such as potentiometer, audio input either through MIC or 3.5 mm audio jack, Hall effect sensor, LDR (Light- Dependent Resistor), and temperature sensor using LM35 or thermistor.

Chapter 15 deals with the serial communication peripherals which includes SPI (Serial Peripheral Interface) and I2C (Inter-Integrated Circuit). This chapter also includes applications of such serial communication peripherals like real-time clock (RTC), digital-to-analog converters using I2C peripheral and microSD card, and television display using SPI peripheral.

Chapter 16 deals with the various ways of user interface (input as well as output) such as 16×2 LCD, 16 keys keypad, and PS/2 keyboard. Also, includes the use of shift registers as 16×2 LCD and 16 keys keypad is connected serially through the shift registers.

Chapter 17 focuses on development of standalone projects by using Tiva C Series microcontrollers instead of Tiva LaunchPad. This chapter will help the reader in developing their own standalone circuits or projects independent of the Tiva LaunchPad.

5 List of Experiments

The experiments listed below can be implemented using Tiva LaunchPad and PadmaBoard. The list also includes experiments related to standalone implementation of Tiva microcontrollers. The list is divided into two parts (Tables 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11):

- Experiments using Register Access
- Experiments using API (Application Programming Interface)

Experiment list under API is further divided on basis of the requirement of microcontroller peripherals to perform that experiment. A brief description for each experiment is mentioned below in tabular format.

- **Experiments Using Register Access**

Table 1 Overview of experiments based on register access

S. No.	Experiment name	Brief description
1.	Blinky	The first experiment performed on any microcontroller. Blink an LED
2.	Switchy	One step ahead of blinky. Use a switch to control an LED

- **Experiments Using API (Application Programming Interface)**

a. **Digital Input/Output**

Table 2 Overview of experiments based on digital input/output

S. No.	Experiment name	Brief description
3.	API blinky	Blink an LED—using API
4.	API switchy	Use API to implement a switch controlling a LED
5.	Running LEDs	Blink the LEDs in sequence to produce a pattern
6.	LED as light sensor	Blink an LED at rate proportional to the intensity of light falling on it
7.	Switch toggle	Control the state of LED using a switch
8.	Electronic dice	Display a random number in binary format using 3 LEDs when a switch is pressed
9.	Live morse generation	A simple Morse code generator using two switches
10.	Morse recorder	Record a complete morse message and play it back at the desired speed
11.	Car parking sensor	Use the ultrasonic sensor to detect imminent collisions and generate an audible warning

b. **Interrupt**

Table 3 Overview of experiments based on interrupt

S. No.	Experiment name	Brief description
12.	Interrupt switchy	Switchy-using interrupts

c. **Timer**

Table 4 Overview of experiments based on timer

S. No.	Experiment name	Brief description
13.	Software PWM	Use software PWM to vary the brightness of an LED
14.	Hardware PWM	Implement PWM using PWM feature of the timer module
15.	Systick timer blinky	Blinky-using systick timer
16.	Obstacle Sensor	Use the IR LED and TSOP pair to detect close range obstacles
17.	Remote control	Use a television remote to toggle LEDs
18.	IrDA	Transfer data between two boards using infrared transmitter receiver
19.	Watchdog timer	Learn how to use the Watchdog Timer

d. **Universal Asynchronous Receiver and Transmitter (UART)**

Table 5 Overview of experiments based on UART

S. No.	Experiment name	Brief description
20.	UART Echo	Re-transmit any data received by UART module
21.	Bluetooth control	Control an LED from a bluetooth device
22.	UART intensity control	Use data received on UART to control the intensity of an LED
23.	Color generator	Generate colors on the RGB LED using host PC for input
24.	Ultrasonic ranger	Send the distance measured by ultrasonic ranger to a host PC
25.	RS232 communication	A demonstration of RS232 protocol using two boards
26.	RS485 communication	A demonstration of RS485 protocol using three boards

e. **Power Management and System Control**

Table 6 Overview of experiments based on power management and system control

S. No.	Experiment name	Brief description
27.	PLL	Use internal PLL for generating a high-frequency system clock
28.	Runtime PLL	Modify the system clock frequency during runtime using PLL
29.	Sleep mode and deep-sleep mode	Performing various modes of operation by putting system into sleep and deep-sleep mode
30.	RTOS	Learn the basics of a real time operating system running on Tiva

f. **Analog to Digital Converter (ADC)**

Table 7 Overview of experiments based on ADC

S. No.	Experiment name	Brief description
31.	Thumbwheel	Read the position of thumbwheel potentiometer using ADC and send it to a host PC
32.	Controlled temperature sensor	Use LM35 to sense ambient temperature and send to a host PC
33.	Thumbwheel intensity control	Use thumbwheel potentiometer to control the brightness of an LED
34.	Mini VU meter	A small VU meter implemented on LEDs
35.	Sound control	Use sound level to control the brightness of an LED
36.	Temperature on RGB	Use temperature to control the color of RGB LED
37.	Temperature over Bluetooth	Use a Bluetooth module to wirelessly send ambient temperature data to a host PC
38.	Improved ultrasonic ranger	Add a temperature-based correction factor to ultrasonic distance measurement
39.	Temperature alarm	Sound the buzzer when ambient temperature is outside a set range
40.	Thermistor linearization	Linearization of thermistor using characteristic equation
41.	Thermistor - lookup	Use lookup tables to obtain temperature data from a thermistor
42.	Hall effect sensor	Observe the change in magnetic field on host PC
43.	Speedometer	Calculate speed using the hall effect sensor and a magnet on a wheel
44.	Automatic night lamp	Sense the intensity of ambient light and change intensity of LED according to it
45.	Light alarm	Trigger the buzzer when intensity of light is beyond a certain range
46.	Sound measurement	Display the sound intensity values on a host PC
47.	Digital filters implementation	Implement digital filters with the output shown on three LEDs

g. **Inter-Integrated Circuits (I2C)**

Table 8 Overview of experiments based on I2C

S. No.	Experiment name	Brief description
48.	Sine wave generator	Generate a low frequency (<1 kHz) sine wave using I2C based DAC
49.	Real-time clock	Use the PCF8563 RTC to store and display current time
50.	Alarm clock	Raise an alarm buzzer at a pre-set time
51.	Twilight calculator	Display the twilight times (dawn and dusk) of a particular location
52.	Sun tracker	Determine position of the sun based on time and date
53.	High frequency Sine wave generator	Generate higher frequency (>1 kHz) sine waves using high-speed mode of I2C
54.	Lissajous figures	Display Lissajous figures on an oscilloscope using dual DACs
55.	Oscilloscope clock	Display a clock face with the current time on an oscilloscope
56.	Classic Brick game	Play the popular bricks game on an oscilloscope
57.	Chaos	Plot chaos equation on an oscilloscope

h. **Serial Peripheral Interface (SPI)**

Table 9 Overview of experiments based on SPI

S. No.	Experiment name	Brief description
58.	Tiva on a TV	Display a welcome screen on television using composite video input
59.	Weather channel	Display ambient temperature on television screen
60.	Hello SD card!	Create a Hello World! text file on microSD card
61.	Temperature recorder	Record temperature data in a text file on the micro SD card whenever a switch is pressed
62.	Temperature logger	Log the ambient temperature along with current time to a text file in microSD card at fixed time intervals
63.	Voice recorder	Record audio on microSD card and play it back
64.	WAV player	Play WAV files present from microSD card

i. **User Input and Output Devices**

Table 10 Overview of experiments based on sand output devices

S. No.	Experiment name	Brief description
65.	Hello LCD!	Display Hello World! on a 16×2 LCD
66.	PS/2 keyboard	Use the PS/2 keyboard to control the RGB LED
67.	Scrolling display	Print a scrolling message on 16×2 LCD
68.	Calculator	A simple calculator using the 16×2 LCD and 4 × 4 Keypad
69.	VU meter	A VU meter using 16×2 LCD display
70.	Digital filters on LCD	Implement digital filters, showing the output on a 16×2 LCD
71.	Spectrometer	Display the sound spectrum on a 16×2 LCD display using FFT
72.	Digital clock	Display the current time and temperature on 16×2 LCD
73.	Text editor	Type a message with the keyboard on the 16×2 LCD. Store it in microSD card
74.	Piano	Use 16-key keypad as a piano
75.	Tone recorder	Use the keypad to record a tune and then play it back
76.	Universal remote	Generate remote control signals using keypad for input

j. **Standalone Projects Using Tiva Microcontrollers**

Table 11 Overview of standalone projects using Tiva microcontrollers

S. No.	Experiment name	Brief description
77.	Controlled Blinky	Blinky with blink rate controlled by two switches
78.	UART-based LED control	Control the intensity of an LED using data received from UART
79.	Temperature display	Display the ambient temperature on 7-segment LED displays
80.	Talking range finder	An ultrasonic range finder with audible distance readout

Chapter 2
ARM Cortex-M4 Core and Tiva C Series Peripherals

ARM is one of the leading semiconductor IP companies founded in 1990, as Advanced Risk Machine out of the collaboration between Apple Computer, Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced it ARM6 processor design and VLSI Technology was the first to license its design. After that many other leading semiconductor companies including Texas Instruments, STMicroelectronics, Freescale, Broadcom, Atmel, nVIDIA, etc., have licensed ARM's processor and IP designs.

Over the years, ARM has developed many processor and IP designs, which have been bought by over 250 companies. Among their offerings is the ARM Cortex family, based on the ARMv6 and ARMv7 architecture. The ARM Cortex family subdivided into three profiles as mentioned below:

1. **Cortex A Profile** - It is based on ARMv7-A architecture, the processors are categorized for the applications requiring the capability to run complex operating systems like Android, Microsoft Windows, Linux. This class of processors are commonly found in smartphones, tablets, digital television, set-top boxes, personal digital assistants (PDAs). Processors like Cotex A-50 deliver high performance, with the ability to execute 64-bit instructions.
2. **Cortex R Profile** - It is based on ARMv7-R architecture, the processors are categorized for real-time applications where high reliability, availability, fault tolerance, and real-time responses are required. These processors typically run a Real-Time Operating system (RTOS), along with appropriate user code. Cortex R processors are often used in Automotive Control Systems like anti-lock braking systems, mass storage controllers, and wireless and wired sensor networks.
3. **Cortex M Profile** - This profile uses ARMv6-M architecture design for its lower end processors such as Cortex-M0, Cortex-M0 plus and ARMv7-M architecture design for its higher end processor design such as Cortex-M3 and Cortex-M4. The processors are optimized for cost- and power-sensitive applications like smart

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_2

metering, automotive and industrial control systems, and human interface devices. The profile consists of Cortex-M0 and Cortex-M0 plus for ultra-low power and Cortex-M3 and Cortex-M4 for digital signal processing applications.

1 Overview of ARM Cortex-M4 Architecture

ARM Cortex-M4 is a 32-bit processor designed mainly to have high processing performance with faster interrupt handling capabilities along with low power consumption. It is built with a 3-stage pipelined Harvard architecture ideal for the embedded applications, which require high-end processing. The performance of the Cortex-M4 is enhanced with a single-precision floating point computation unit, MAC (Multiply with Accumulate) capability, and single-cycle multiplication. In Cortex M4, components are integrated tightly to reduce system area hence improving the interrupt latencies.

Cortex M4 implements the Thumb 2 instruction set. Thumb 2 instruction set includes 16-bit thumb instructions along with the 32-bit ARM instructions. This widens the scope of instructions that can be implemented on the processor, giving it a code density similar to the Thumb instruction set with performance similar to the ARM instruction set on 32-bit memory map. The processor can be switched to ultra-low power consumption by changing its state to sleep mode or deep-sleep mode. The processor has WIC (Wake-up Interrupt Controller) which helps the processor to return back to normal mode of execution of operation (or run mode) by detecting the interrupt while in deep-sleep mode. Cortex-M4 processor architecture provides the optional trace port interface in addition to debug interface.[1] Figure 1 shows the Cortex-M4 architecture along with optional components.

Cortex M4 features and benefits are summarized below [1][2], some of these have already been discussed above, while the remaining will be discussed in subsequent sections:

- 32-bit processor along with 32-bit data path, 32-bit register bank, and 32-bit memory interfaces
- Follows the Harvard architecture with 3-stage pipelined fetch, decode and execute
- Nested Vectored Interrupt Controller (NVIC) is closely integrated with the processor core which results in low interrupt latency, is configurable from 1 to 240 for external interrupts
- Optional Floating point Unit (FPU) which supports single- precision data processing operations, along with hardware support for addition, subtraction, multiplication along with optional accumulate, division and square root operations
- An optional Memory Protection Unit (MPU) that permits control of individual regions in memory

[1] Trace module keeps the record of every register and memory location value for each clock cycle unlike the debug module which captures the register or memory location values at a particular instant. It also helps in determining the time taken by each application segment.

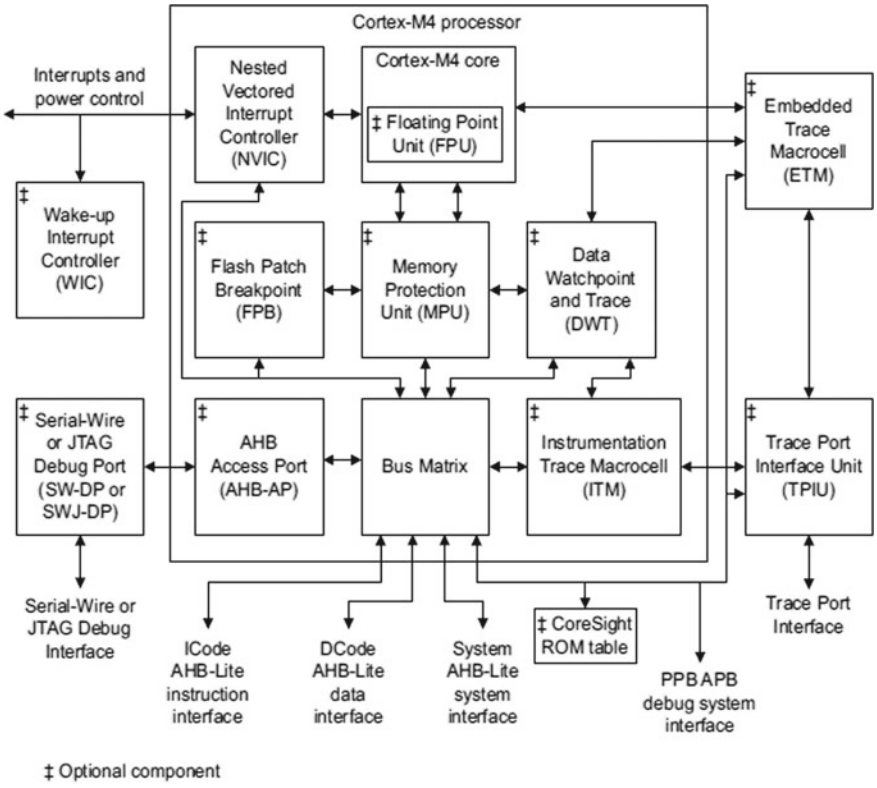

Fig. 1 Cortex-M4 architecture block diagram (ARM Cortex-M4 technical reference manual, revision r0p1)

- Configurable Processor Mode and Privilege Levels for better execution of applications with appropriate resource allocation depending on the application.

2 Cortex-M4 Core Peripherals

This section describes the core peripherals of Cortex-M4 processor.

2.1 Nested Vectored Interrupt Controller (NVIC)

It is an embedded interrupt controller that supports the configuration of external interrupts with maximum number of 240 interrupts. It allows to assign a programmable priority level from 0 to 255, 0 being the highest priority level. It also supports

the tail chaining of interrupts. Through tail chaining the low latency of interrupts is achieved. As in tail chaining on completion of the exception or interrupt if there is another pending exception then the processor skips the stack pop operation which is require returning to the interrupted program instead of this it transfer the control to the pending exception handler.

Along with the above features NVIC supports Non-Maskable Interrupt (NMI) and also Wake-up Interrupt Controller (WIC) for providing ultra-low power sleep mode support.

2.2 Floating Point Unit (FPU)

This unit is capable of performing single-precision (or 32-bit) floating point operations. Its supports add, subtract, multiply and accumulate, and square root operations. It also provides conversions between fixed-point and floating point data formats, and floating point instructions. The FPU single-precision (or 32-bit) registers can also be accessed as 16-bit double word registers for load, store, and move operations. The functionality provided by the FPU for floating point computation is compliant with the ANSI/IEEE Std 754-2008, IEEE Standard for Binary Floating Point Arithmetic, referred to as the IEEE 754 standard.

2.3 System Control Block (SCB)

The SCB is the programmers model interface to processor. It provides system implementation information, control configuration, system control, and reporting system exceptions.

2.4 System Timer-SysTick

Cortex-M4 processor has 24-bit down count system timer, which counts down from the reload value to zero. On reaching zero, system timer wraps up, reloads the value stored in reload value register and resumes counting down during the subsequent clocks.

2.5 Memory Protection Unit (MPU)

The MPU divides the memory into number of regions and assign each regions with memory location, size, access permission, and memory attributes. Assignment of

these attributes to memory region affects the memory access of that region. In Cortex-M4 there are memory regions mentioned below:

- Eight separate memory region, from 0 to 7
- Background Region

Background region will have the same memory attributes as that of the default memory but background region is only accessible through the privileged software only. It is generally used by the embedded OS for preventing process to access of the prohibited memory location. As whenever the process tries to access the prohibited memory location, MPU will generate the memory manage fault exception which might cause termination of the process.

3 Programmer's Model

This section of chapter deals with the operation modes in which processor can be operated, privilege levels and overview of core registers for software execution.

3.1 Processor Modes

Processor can be operated in following two modes:

- **Thread Mode**: This mode is used by the processor to execute application software. Processor enters the thread mode after coming out of reset.
- **Handler Mode**: In this mode processor handles the exceptions. After finishing all the exception processing, processor returns back to the thread mode.

3.2 Privilege Levels

The code can be executed as privilege code or as unprivileged code. By the executing code as unprivileged code, it limits or restricts the access to some of the resources for its execution. Whereas by executing code as privileged code, it made all resources accessible for its execution.

- **Unprivileged**: Unprivileged code cannot access the system timer, NVIC, or system control block. Also, code may have restricted memory and peripheral access. There are certain instructions to which unprivileged code will be having limited access like instruction to move data between general-purpose registers and special registers. So, unprivileged code is executed at unprivileged level.
- **Privileged**: Privileged code can use all the instructions and has access to all the resources. So, privileged code is executed at privileged level. In thread mode,

Table 1 Privilege levels, stacks used and usage of the different processor modes[a]

Processor mode	Use	Privilege level	Stack used
Thread mode	Applications	Privileged or unprivileged	Main stack or process stack
Handler mode	Exception handlers	Privileged	Main stack

[a]This table is referenced from Cortex-M4 Devices Generic User Guide by ARM

the CONTROL register controls whether the software execution is privileged or unprivileged, but in Handler mode software execution is always privileged.

3.3 Stacks

The processor implements a descending stack. The stack pointer will contain the address of last item that is stacked in the memory. When processor pushes a new item, the stack pointer will decrement and the new item will be placed at decremented memory location. There are two stacks implemented in processor: main stack and process stack.

In thread mode both the main and process stack can be used, and CONTROL register controls which stack will be used. But, in handler mode only main stack is used by processor.

Table 1 summarizes the privileged or unprivileged execution of code, stack used depending on the processor mode.

3.4 Core Registers

Cortex-M4 has 32-bit registers which are described below:

1. 13 General purpose registers, R0–R12
2. Stack Pointer (SP), R13, this stack pointer can be aliased to either Main Stack Pointer and Process Stack Pointer
3. Link Register (LR), R14
4. Program Counter (PC), R15
5. Special registers like program status register (PSR), exception mask registers and control register

Figure 2 shows the register map of Cortex M4 processor.

- **General-purpose registers**: R0–R12 are for data operations. They are divided into two sets of registers:

  Low Registers: Registers R0–R7 are under this set. These registers are accessible by all instructions.

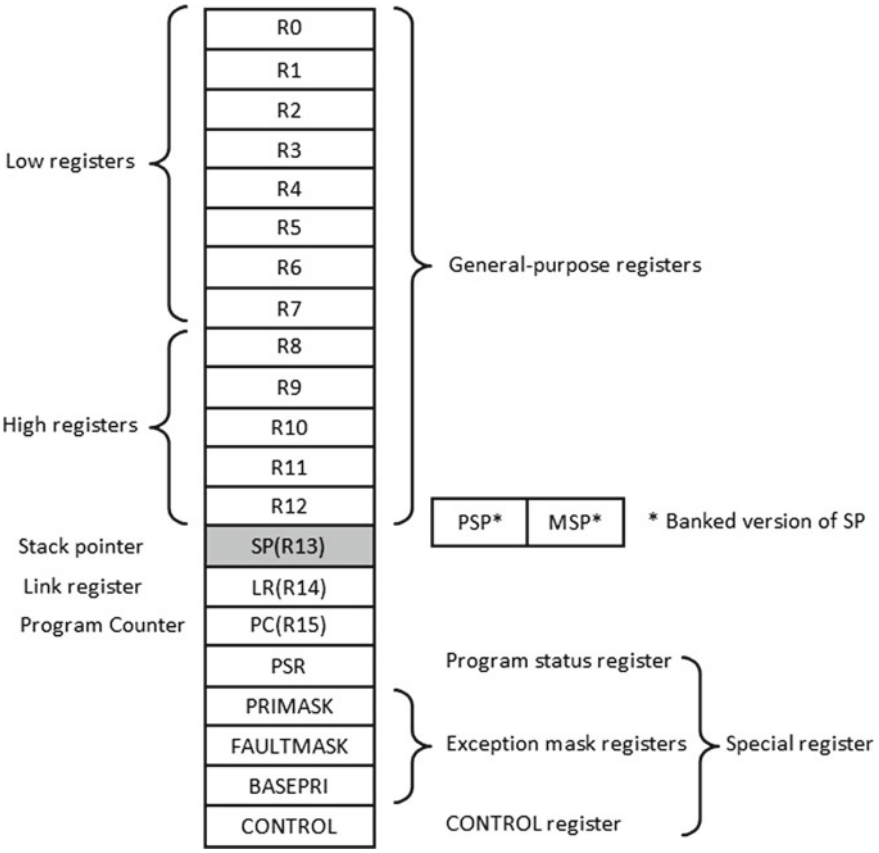

Fig. 2 Register map (Cortex-M4 devices generic user guide, ARM)

High Registers: Registers R8–R12 are under this set. These registers are accessible only by 32 bit instructions not by 16 bit instructions.

- **Stack Pointer (SP)**: Register R13 is used as stack pointer for the main stack and process stack. The SP ignores the bits [1:0], hence SP is auto-aligned to word or 32-bit memory boundaries. In handler mode SP always points to main stack or will act as Main Stack Pointer (MSP), whereas in thread mode SP can be configured to be the Main Stack Pointer (MSP) or the Process Stack Pointer (PSP) through CONTROL register.
- **Link Register (LR)**: Register R14 is used as link register. The LR stores the return address from program counter during execution of branching instruction. Also, LR is used while returning from exception handling.
- **Program Counter (PC)**: Register R15 is used as program counter. It stores the address of the next instruction to be executed. In 32 bit PC value, bit [0] is always

zero so that PC remains aligned to word (32 bit) or half-word (16 bit) instruction boundaries.

- **Program Status Register (PSR)**: Program status register consist of various flags such as arithmetic and logic processing flags and execution status flags. PSR is combined of Application Program Status Register (APSR), Interrupt Program Status Register (IPSR), and Execution Program Status Register (EPSR).
- **Exception Mask Register**: These registers are used to disable the exceptions to be handled by the processor. As, there can be certain unnecessary exceptions which can cause the hindrance to the task performed by processor.

  Priority Mask Register (PRIMASK): By enabling the bit [0] of this register it disables all the exceptions whose priority can be configured. Hence, this register will disables all interrupts except Non- Maskable Interrupt (NMI) and hard fault.
  Fault Mask Register (FAULTMASK): By enabling the bit [0] of this register it disables all the interrupts except Non-Maskable Interrupt (NMI).
  Base Priority Mask Register (BASEPRI): In this register 8-bit priority value is written and this will disables all the interrupts having priority equal to or lower priority level than the BASEPRI value.

- **Control Register (CONTROL)**: It controls the stack and privilege levels used by the code during its execution, when the processor is in thread mode. Also, it indicates whether the Floating Point Unit is active or not.

4 Memory Model

This section deals with the fixed default memory map of the ARM Cortex-M4 processor, memory endianness, and features like bit banding.

4.1 Memory Map

Since ARM Cortex-M4 is a 32 bit processor, it can have up to 4GB of addressable memory. This addressable memory space is used by the code memory, SRAM, external and internal peripherals, and there is also a vendor-specific address space. Figure 3 illustrates the default memory map of ARM Cortex-M4 processor.

4.2 Bit Banding

Bit Banding is the optional feature provided by ARM along with Cortex-M4 System Development Kit. It depends on the client company whether they want to include

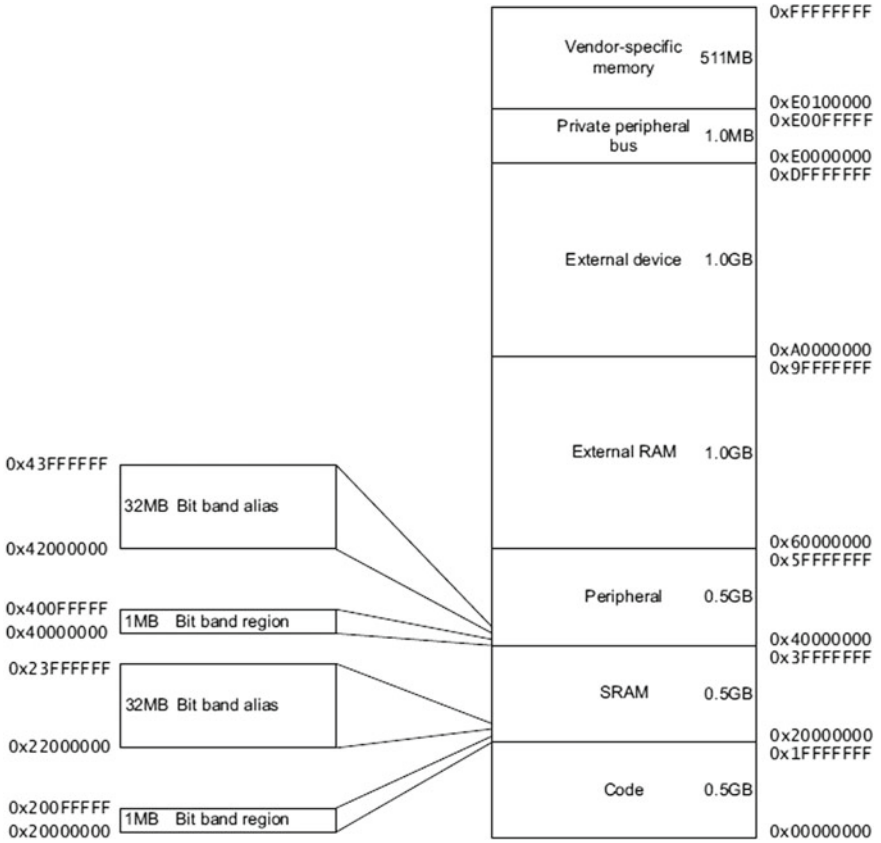

Fig. 3 Memory map (Cortex-M4 devices generic user guide, ARM)

this feature into their Cortex-M4 processor based system.[2] In bit banding there are two regions *bit-band alias region* and *bit-band region*. As shown in Fig. 3 Cortex-M4 memory map has two 32MB bit-band alias region map to corresponding two 1MB bit-band region. These two paired regions are located in peripheral memory region and SRAM region. Each unique word in bit-band alias region is mapped to each unique bit in bit-band region. Hence, each bit in bit-band region can be updated by writing a value to the corresponding word in bit-band alias region. This process is illustrated through a Fig. 4.

[2]Texas Instruments provides Bit Banding feature in ARM Cortex-M4 processor based Tiva C Series microcontroller family.

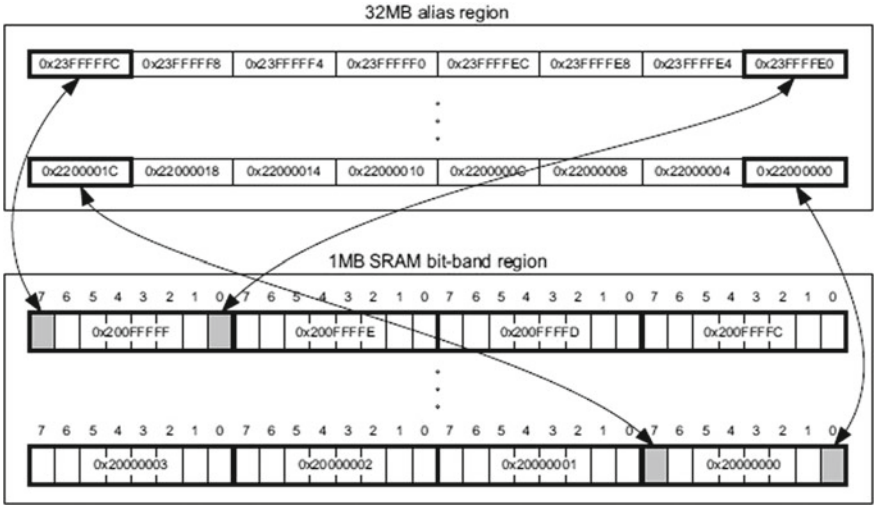

Fig. 4 Bit banding (Cortex-M4 devices generic user guide, ARM)

4.3 Memory Endianness

Cortex-M4 has 32-bit (or 4 bytes or a word) memory bus width. Hence, the processor looks at memory as linear array, where each element is of 8 bits or 1 byte. So, if a processor has to store a word it will be stored in 4 consecutive array elements. For example, first word will stored at 0–3 bytes, second word will be stored at 4–7 bytes, and so on. These words can be stored in two possible formats:

- Big Endian Format
- Little Endian Format

Big Endian Format

In this format, first most significant byte (MSbyte) is stored at the directed memory location (lets say A), then the next MSbyte is stored at consecutive memory location (at A+1), and so on. Hence the least significant byte (LSbyte) is stored at memory location A+3. This is illustrated in Fig. 5.

Little Endian Format

In this format, first LSbyte is stored at the directed memory location (lets say A), then the next LSbyte is stored at consecutive memory location (at A+1), and so on. Hence, the MSbyte is stored at memory location A+3. This is illustrated in Fig. 6.

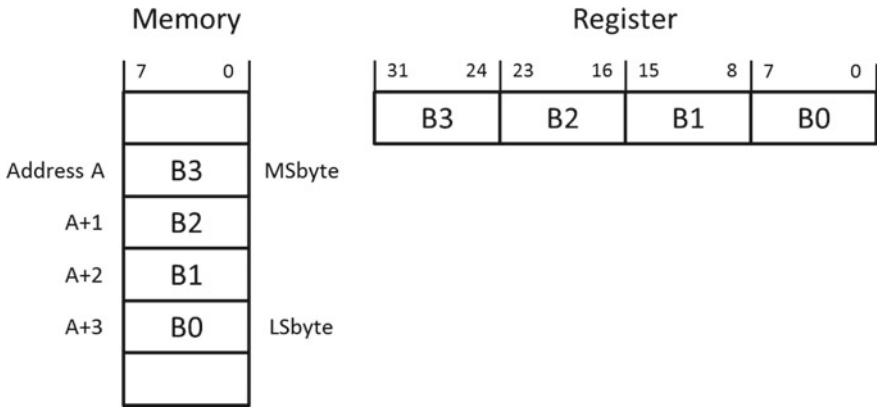

Fig. 5 Big endian format

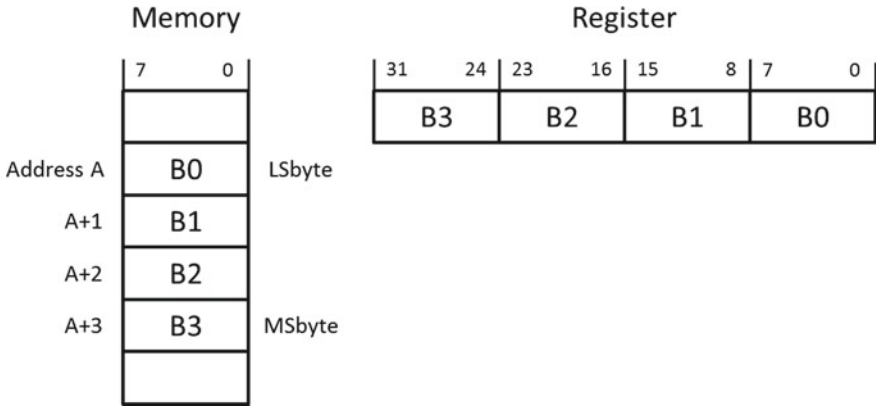

Fig. 6 Little endian format

5 Advanced Microcontroller Bus Architecture (AMBA)

Advanced Microcontroller Bus Architecture (AMBA) is the bus architecture designed by the ARM for on-chip communication in high-end microcontrollers. Based on the need for performance, power, frequency there are distinct buses defined under this architecture:

- Advanced High-Performance Bus (AHB)
- Advanced Peripheral Bus (APB)

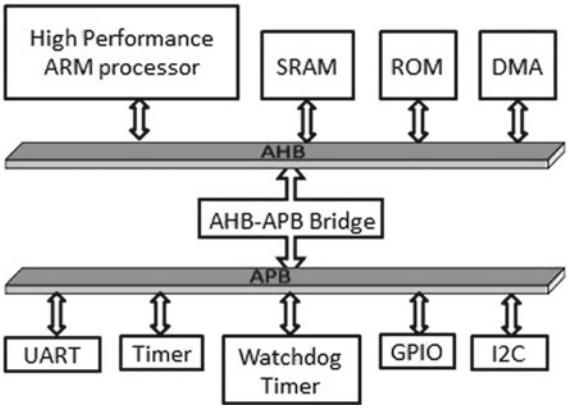

Fig. 7 Basic AMBA-based ARM system

5.1 Advanced High-Performance Bus (AHB)

AHB is for high-performance and high-frequency peripherals, like on-chip processors, memories, or other IPs like DMA (Direct Memory Access). The AHB can have multiple masters, controlling multiple slaves.[3] Hence, all critical peripherals of the system are placed on this bus. This bus have a high bandwidth interface between the interfacing elements. It also supports pipelined and burst operations.[4]

5.2 Advanced Peripheral Bus (APB)

APB is designed for low-power peripherals, which are operating at much slower speeds or frequency than the processor. Hence, they need not to be operated at frequencies as critical peripherals like memories need to be operated. Hence, by placing these peripherals on APB, the power consumption can be minimized.

Figure 7 shows the general orientation of the peripherals on the bus. Of course, position of peripherals can change as per the need of the system.

6 Texas Instruments Tiva C Series Family

Tiva C Series microcontroller family of Texas Instruments is based on ARM Cortex-M4 processor. Before starting with Tiva C Series microcontroller evaluation board

[3] AHB has a component, AHB arbiter which ensure that at a time only one master initiates the data transfer.

[4] Burst operations are the continuous data transaction by a master on the incremental address space. Increment can be of byte, half word, or word. APB does not support the burst operations.

EK-TM4C123GXL which uses TM4C123GH6PM microcontroller, there are some prime features highlighted below for the mentioned microcontroller. There are wide range of microcontrollers in the family, so it is not necessary that all the microcontrollers of this family has all these mentioned features.

- ARM Cortex-M4 Processor Core
  - 80 Mhz Operation, 100 DMIPS performance
  - Thumb-2 mixed 16-/32-bit instruction set delivers the high performance expected of a 32-bit
  - Floating Point Unit
- On-Chip Memory
  - 256KB single cycle flash
  - 32KB single cycle SRAM
- GPIOs
  - 43 GPIOs, 5V tolerant input configuration
  - Fast toggle capable of a change every clock cycle for ports on AHB, every two clock cycles for ports on APB
  - Programmable control for GPIO interrupts
  - Programmable control for GPIO pad configuration
- General Purpose Timers
  - GPTM (General-Purpose Timer Module) contains six GPTM blocks of 16/32-bit and six GPTM blocks of 32/64-bit
  - Each block is capable to perform split operation or combined operation, i.e., each 16/32-bit GPTM block has two 16-bit timer or one 32-bit timer
  - As one 32/64-bit Real-time Clock
  - For Pulse Width Modulation
  - For ADC event trigger
- Analog-to-Digital Converter (ADC)
  - Two ADC modules
  - Twelve analog input channels
  - Single ended and differential input configurations
  - On-chip internal temperature sensor
  - Sample rate of 1000 thousand samples per second
- Universal Asynchronous Receiver and Transmitter
  - Eight fully programmable UART modules
  - Fully programmable serial interface characteristics
- Synchronous Serial Interface (SSI)
  - Four fully programmable SSI modules

  - Master of Slave Operation
  - Programmable clock bit rate and operation
  - Programmable data frame size from 4 to 16 bits

- Inter-Integrated Circuit
  - Four fully programmable I2C modules
  - Devices on the I2C bus can be designated as either master or a slave
  - Four I2C modes: Master Transmit, Master Receive, Slave Transmit, Slave Receive
  - Four transmission speeds, standard (100 kbps), fast mode (400 kbps), fast mode plus (1 Mbps), High-speed mode (3.3Mbps)
  - Master and slave interrupt generation

- Power Management
  - On-chip Low Drop (LDO) voltage regulator with programmable output user adjustable from 2.25 to 2.75V
  - Low-power options on controller: Sleep and Deep-Sleep Modes
  - Low-power options for peripherals: software controls shutdown of individual peripherals
  - 3.3V supply brownout detection and reporting via interrupt or reset

- Flexible Reset Sources
  - Power-on Reset (POR)
  - Reset pin assertion
  - Brownout (BOR) detector alerts to system power drops
  - Software reset
  - Watchdog timer reset

- JTAG
  - IEEE 1149.1-1990 compatible Test Access Port (TAP) controller
  - Four-bit Instruction Register (IR) chain for storing JTAG instructions
  - IEEE standard instructions: BYPASS, IDCODE, SAMPLE/PRELOAD, and EXTEST
  - Integrated ARM Serial Wire Debug (SWD)

Chapter 3
Tiva C Series LaunchPad

This chapter deals with a detailed description of the Tiva C Series TM4C123G LaunchPad. The Tiva LaunchPad is an evaluation board (EK-TM4C123GXL) from Texas Instruments, which uses ARM Cortex-M4F based microcontroller of Tiva C Series. The letter "F" in the ARM Cortex-M4F denotes that the microcontroller has a dedicated floating point unit and hence is capable of catering to several signal processing applications. This evaluation board is low cost and a great platform to get acquainted with ARM Cortex M4 microcontrollers. This evaluation board has excellent software and hardware support provided by Texas Instruments as well as other third-party manufacturers. There are several add-on daughter boards provided by Texas Instruments and other third- party manufacturers, with features like LCD display, keypad, sensors, etc., which can be plugged to the expansion headers of the LaunchPad. Such boards are termed as BoosterPacks. Essentially, these BoosterPacks add additional hardware interfaces to the Tiva LaunchPad. A user could easily develop their own custom booster pack or daughter board for a particular requirement. A similar custom board, the PadmaBoard, is presented in this manual. With the help of the PadmaBoard, one can perform several experiments using the Tiva LaunchPad and can improve their understanding of embedded systems to a great extent. PadmaBoard is discussed in detail in next chapter.

1 Board Overview

The Tiva LaunchPad features the TM4C123GH6PM microcontroller of Tiva C Series family, which supports several advanced features as compared to other micrcontrollers belonging to the same family. The key additional features supported by the microcontroller, in addition to the ones previously discussed, are as follows:

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_3

1. Universal Serial Bus (USB) controller which supports USB Host, Device, and OTG functions.
2. Motion Control Pulse Width Modulator (MC PWM) module.

The Fig. 1 shows the illustrative display of the Tiva LaunchPad. The brief overview of features on Tiva LaunchPad are enumerated below:

- USB micro-A and micro-B connector for USB device, host and OTG (On the GO) functionality
- Power select switch between ICDI and USB Device connectors
- On-Board ICDI (In-Circuit Debug Interface)
- Reset Switch
- Programmable RGB LED
- Two user switches
- Expansion headers to connect additional hardware to the Tiva LaunchPad

The LaunchPad comprises of two microcontrollers (TM4C123GH6PM) on the board, one is the target microcontroller on which the user applications run and the other microcontroller is preprogrammed to act as the In-circuit Debug Interface (ICDI). This allows the user to independently use the ICDI part of the board to program other Tiva C Series micrcontrollers, by disconnecting the on-board target microcontroller, the details of which are explained in Chap. 5.

2 Hardware Description

2.1 Power Supply

The Tiva C Series TM4C123G LaunchPad can powered from two different sources:

- On-board ICDI USB connector
- USB device connector

The power selection is done using a slide switch SW3 to select power input from one of the two on-board USB micro-B connectors.

2.2 Hibernate

The microcontroller on Tiva LaunchPad supports the Hibernation functionality. In hibernation mode when processor and peripherals are idle then the power can be removed from them with only hibernation module being powered up. The hibernation module can be powered up by using external battery or through the auxiliary power supply. Power can be restored based on the WAKE signal (external signal connected to switch SW2) or after a certain time by using built-in Real-Time Clock (RTC).

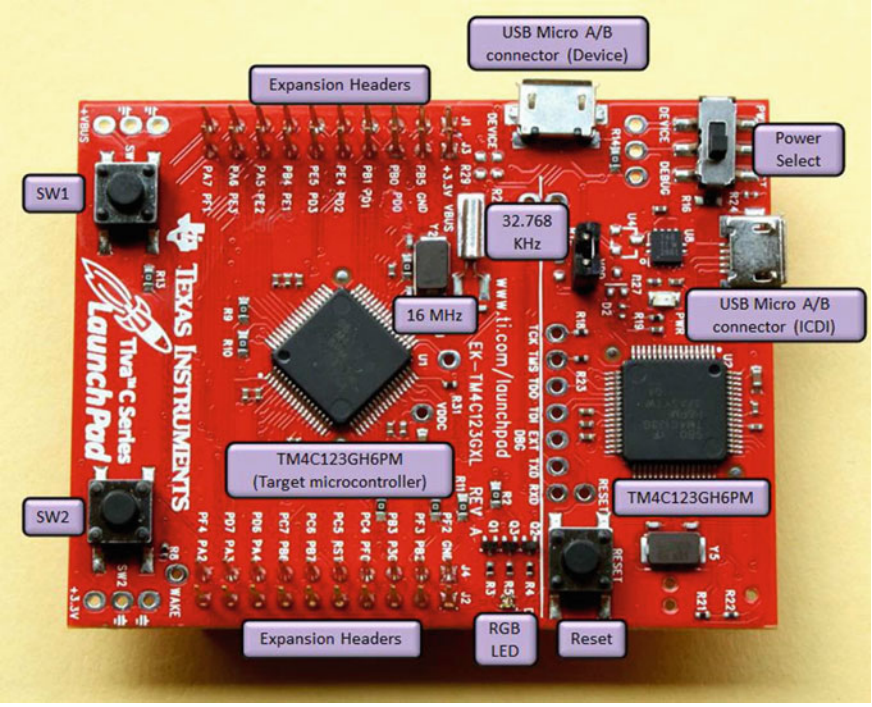

Fig. 1 Tiva C series TM4C123G LaunchPad

Also, Tiva LaunchPad has 32.768 KHz crystal as clock source input for hibernation module.

2.3 Clock

The Tiva C Series LaunchPad uses 16 MHz crystal as clock source input for the internal main clock. An internal PLL, can be configured by the software to generate different frequencies ranging from 3.125 to 80 MHz for the core. The hibernation module has a different clock source, and uses external 32.768 KHz crystal.

2.4 Reset

The reset signal of microcontroller, TM4C123GH6PM is active low. The reset signal is connected to the reset switch and to the ICDI circuit for a debugger-controlled reset. The reset signal is pulled up using 10 kohm resistor, so that microcontroller

Table 1 In-circuit debug interface pin mapping

S. No.	GPIO pin	Pin function
1.	PC0	TCK/SWCLK
2.	PC1	TMS/SWDIO
3.	PC2	TDI
4.	PC3	TDO/SWO

Table 2 RGB LED and user switches pin mapping

S. No.	GPIO pin	Pin function
1.	PF1	RGB LED (Red)
2.	PF2	RGB LED (Blue)
3.	PF3	RGB LED (Green)
4.	PF4	User Switch, SW1
5.	PF0	User Switch, SW2

resets only when reset signal is triggered. External reset can be asserted under three conditions:

- Power-on reset
- RESET switch pressed
- By the ICDI circuit when instructed by the debugger

2.5 In-Circuit Debug Interface (ICDI)

Tiva C Series LaunchPad has a on-board In-Circuit Debug Interface (ICDI), which is used for the programming of on-board microcontroller, TM4C123GH6PM. ICDI can also be used to program other micrcontrollers belonging to the Tiva C Series family. The ICDI supports only JTAG debugging. To use Serial Wire Debug interface external debugger need to be connected. The pin mapping of ICDI pins is shown in Table 1.

2.6 LEDs and Switches

The Tiva C Series LaunchPad has programmable RGB LED and two user switches. Two user switches can be used by user for controlled input. The evaluation board also has a green power LED. Table 2 shows the pin mapping of features to the microcontroller on Tiva LaunchPad.

2.7 Microcontroller Expansion Headers

The Tiva C Series LaunchPad has two double rows of headers namely, J1, J2, J3, and J4. These headers are connected to various GPIOs and peripherals (like UART, I2C, SPI, etc.) of the microcontroller TM4C123GH6PM and also to the power supplies available on the board (+5, +3.3 V and GND). These headers provide the scope to interface more hardware features with Tiva LaunchPad. There are daughter boards (Booster Packs) available for Tiva LaunchPad manufactured by Texas Instruments or by third-party manufactures. Power to these boards can be transferred from these headers, so there is no need to provide power circuitry (voltage regulators, power input connectors, etc.) on these boards, unless the power requirement by daughter boards is very high like driving motors or high watt LEDs.

A similar board, PadmaBoard which has a lot of features on it to exercise Tiva LaunchPad to its fullest capability is referred in this manual while performing experiments. The detailed description of PadmaBoard is discussed in next chapter. One can also develop there own daughter boards as per their need.

Chapter 4
PadmaBoard—Peripheral Motherboard of Tiva C Series LaunchPad

This chapter focuses on detailed explanation of features of the PadmaBoard.[1] PadmaBoard can be connected to the Tiva LaunchPad through the expansion headers present on Tiva LaunchPad. As Tiva LaunchPad is a evaluation board for Tiva C Series ARM Cortex-M4-based microcontroller which has various peripherals. But to implement these peripherals, there are not enough features on the Tiva LaunchPad. But PadmaBoard has various on-board features which are illustrated in subsequent sections. Using these features user can develop and prototype many applications.

1 Board Overview

PadmaBoard has various features as shown in Fig. 1 which gives the user to interact with wide spectrum of peripherals and can develop various applications using combinations of available features. Brief overview of the features is mentioned below:

- Thermistor (temperature-dependent resistor) for taking analog values using ADC channel
- A LM35 temperature sensor for taking temperature readings using ADC channel
- 3.5 mm Audio Jack for audio input through Aux connected to the ADC channel
- Microphone amplifier with high gain and sensitivity connected to the ADC channel
- 5 wire interface FRC connector to drive serial LCD and Keypad
- I2C bus connector to connect other external I2C-based modules
- User programmable ultra bright LEDs
- LDR (Light-Dependent Resistor) connected to the ADC channel

[1] Padma is sanskrit word for the lotus, a flower with large number of petals. Also, it is noted that in Buddhist, Chinese, Hindu, Japanese cultures most of the dieties are depicted as resting over the lotus flower. So, it is named as PadmaBoard as Tiva LaunchPad is seated above the PadmaBoard and the PadmaBoard has many features just like Lotus has many petals around it.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_4

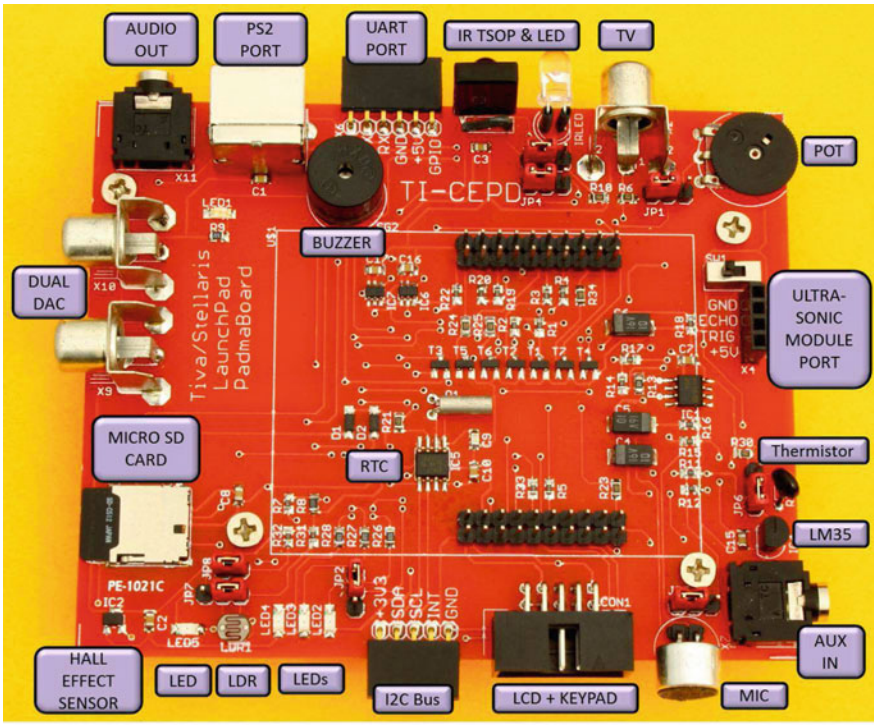

Fig. 1 PadmaBoard

- Ambient light sensor using a LED operated in reverse bias
- Hall effect sensor to sense the magnetic field is connected to the ADC channel
- MicroSD Card interface using SSI (Serial Synchronous Interface)
- RTC (Real-Time Clock), PCF8563 using I2C interface and +3 V 20 mm lithium cell as battery backup for RTC IC
- Two 12 bit DAC-7571 using I2C interface, and a 3.5 mm audio jack for audio output
- One PS/2 ports to connect either PS/2 keyboard or mice
- 6 pin connector of UART pins and 1 GPIO to which UART peripherals modules like bluetooth modules can be attached
- Buzzer to generate various tones
- IR TSOP receiver and IR LED to receive and transmit data and receive data through them
- TV, RCA jack connected using the Master Out pin of SPI module 1 and 1 GPIO
- Thumbwheel potentiometer for reading analog voltage through the ADC channel
- 4 pin connector for ultrasonic module HC-SR04 along with a slide switch to control laser

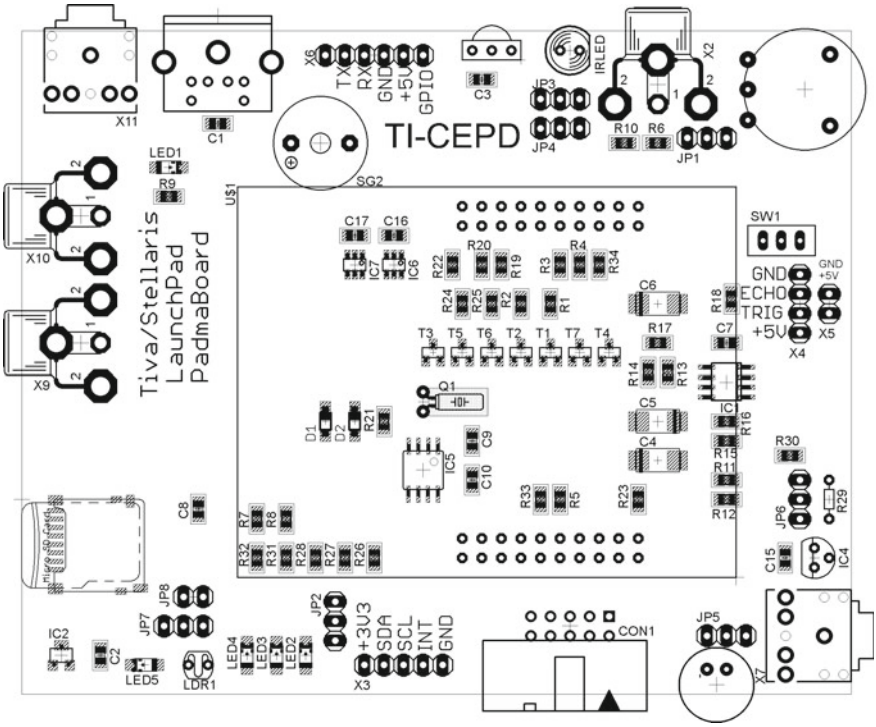

Fig. 2 PadmaBoard PCB—component side

2 Schematic and Layout

Figure 3 shows the schematic of PadmaBoard. However, each feature of PadmaBoard along with their respective schematic part is explained in subsequent sections. PCB[2] designed for the PadmaBoard is double sided PCB, i.e., PCB has two layers top layer and bottom layer. Figure 2 represents the top layer of PCB or component side of PCB.

[2]PCB is a acronym for Printed Circuit Board. It is a board which supports the various electronic components along with various tracks and pads which connects the different components on the board. So to design the PCB first schematic is created by using component symbols and then the layout of components used is done. Layout will contain the actual footprints of the electronic components.

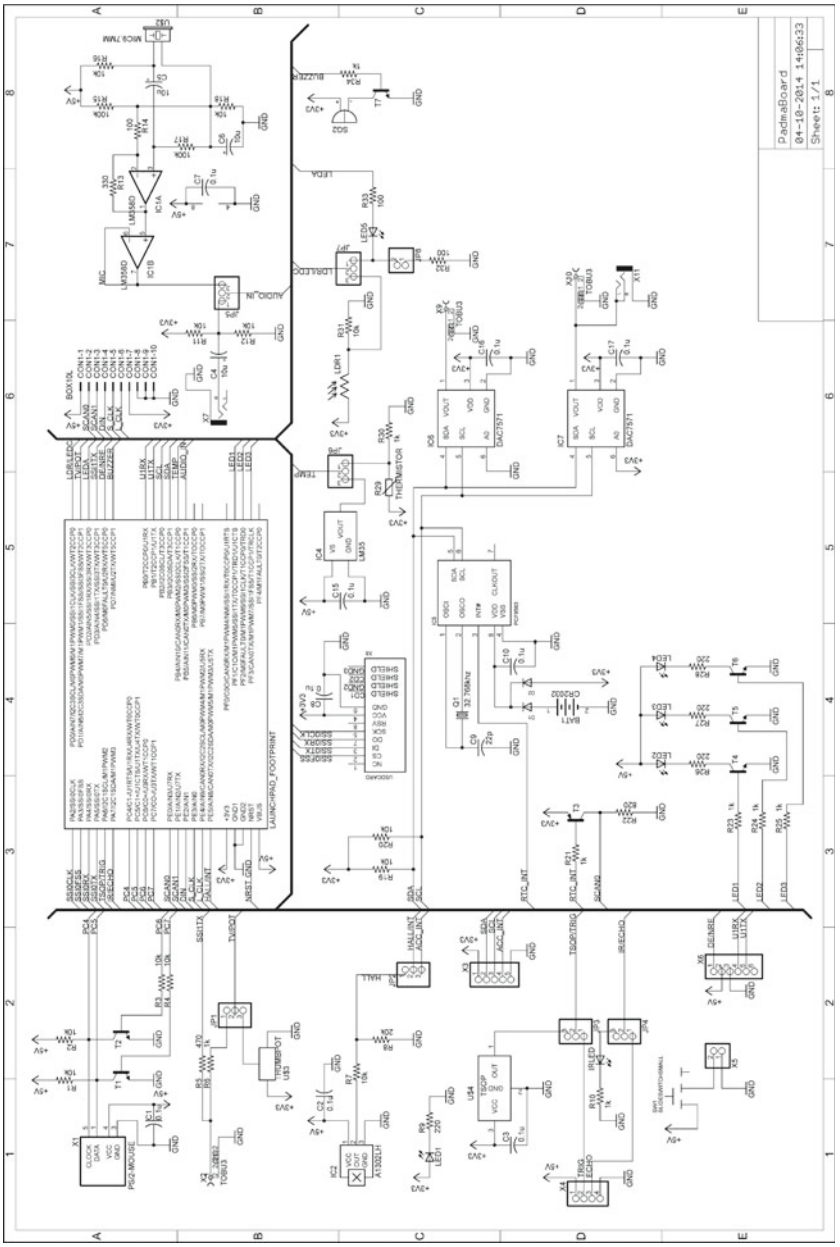

Fig. 3 PadmaBoard schematic

3 Pin Assignment to Peripherals

There are 35 headers as GPIO's on LaunchPad. Among 35 headers 2 pins connected to the 2 switches on LaunchPad and 3 pins are connected to the RGB LED. And PD0 and PB6 are shorted with 0 ohm resistance to make Tiva LaunchPad compatible with MSP430 booster packs. Also PD1 and PB7 are shorted in same manner for the same reason. So, besides the mentioned pins there are 28 pins left on the LaunchPad (Table 1).

35 – 2(for switches) – 3(for RGB LED)[3] *–2(2 pair of pins shorted with each other) = 28 pins*

Multiplexed peripherals:

- TV and potentiometer using JP1
- Hall Effect sensor and Interrupt pin of I2C peripheral bus connector (X3) using JP2
- TSOP and IR Led with Ultrasonic Sensor using JP3 and JP4
- Microphone and audio jack 3.5 mm using a jumper, JP5
- Thermistor and LM35 using a jumper, JP6
- LDR, LED using 2 jumpers (JP7 and JP8), and, 1 jumper to give GND to LED (JP8) when PD0 is not connected to LED.

4 Peripherals Description

4.1 Temperature Sensor

To evaluate the temperature of the surroundings, LM35 (IC4) and thermistor (R29) is provided on the board. Figure 4 shows the schematic of temperature sensing part of the PadmaBoard. The two are connected to the same analog channel (AIN11) through a jumper (JP6) to select the analog values either of LM35 or thermistor. The LM35 is powered by +5 V coming from source power through USB Micro-B connector. The thermistor is a temperature-dependent resistance, therefore, a potential divider is made using another 1k series resistance with end supply voltages from +3.3 V and GND.

4.2 Audio Input

For audio sampling there is a provision for MIC on board which is followed by audio amplifier. The audio amplifier is a two stage, ac coupled microphone amplifier as shown in Fig. 5. The electret microphone which is used is a capacitive microphone

[3]RGB LED pins have been extended to 3 different bright LEDs on PadmaBoard.

Table 1 Pin selection for peripherals on PadmaBoard

S. No.	Peripheral	Pin assignment	Type of pin required	Function
1.	Temperature sensor (LM35, thermistor)	PB4	1 Analog pin	To read temperature
2.	Microphone, audio jack 3.5 mm	PB5	1 Analog pin	To sample audio
3.	Hall effect sensor	PE5	1 Analog pin	To sample the magnetic field strength
4.	Buzzer	PD7	1 GPIO pin	To generate alarm
5.	3 LEDs	PF1, PF2, PF3	3 GPIO pins	Better visualisation
6.	Ultrasonic, IR and TSOP sensor	PA6 and PA7	2 GPIO pins	Obstacle detection
7.	16×2 LCD and 16 keys keypad	PE0, PE1, PE2, PE3, PE4	5 GPIO pins	To display and read keypad
8.	LDR, LED (as sensor or as output)	PD0, PD2	1 Analog pin, 1 GPIO pin	To measure light intensity using LDR or using LED as sensor
9.	Bluetooth, RS232, RS485	PB0, PB1, PD6	UART Module 2 (Rx, Tx), 1 GPIO	To perform serial operation both wired and wireless
10.	MicroSD card	PA2, PA3, PA4, PA5	SSI module 0 (SCK, CS, MISO, MOSI)	To store data
11.	RTC	PB2, PB3, PE0	I2C module 0 (SCL, SDA), 1 GPIO pin	To keep the track of time when power is off
12.	2 DAC, 1 audio jack 3.5 mm	PB2, PB3	I2C module 0 (SDA, SCL)	2 DAC are used to display in XY mode on oscilloscope or 1 DAC with audio amplifier is used to generate audio
13.	I2C bus extension	PB2, PB3, PE5	I2C module 0 (SDA, SCL), 1 GPIO pin	Extended the I2C bus to connect additional I2C modules and 1 GPIO pin for interrupt
14.	1 PS/2 connector (keyboard or mice)	PC4, PC5, PC6, PC7	4 GPIO pins	User interface
15.	TV, potentiometer	PD1, PD3	SSI module 1 (MOSI), 1 GPIO pin	Display unit and potentiometer to generate variable analog values

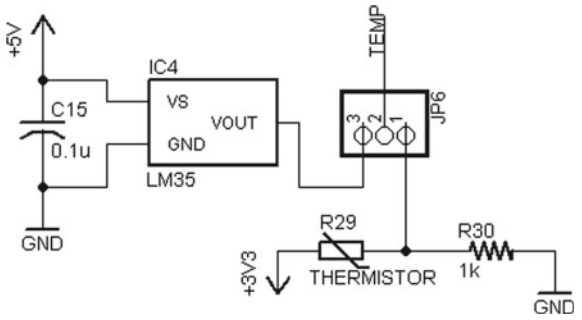

Fig. 4 Temperature sensing using either LM35 or thermistor

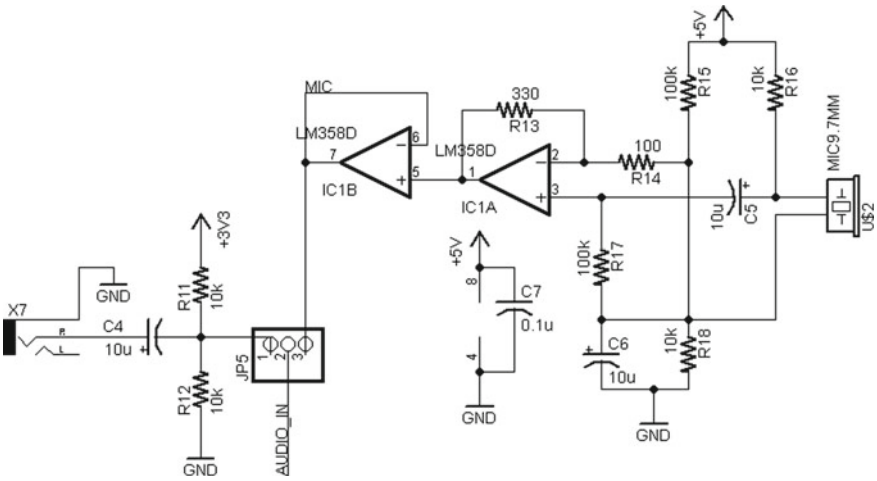

Fig. 5 Audio input using MIC or 3.5 mm audio jack

and it requires a biasing voltage of +5 V supplied through R16. The supply voltage to the LM358 is +5 V and is taken directly from USB bus. The LM358 is a two stage amplifier, the first stage is in non-inverting configuration and the other is a voltage follower. Resistors R15 and R18 provide a bias voltage equal to one tenth the supply voltage to input of the first op-amp. But the DC gain of the op-amp is very low, by virtue of high resistance R17 and the capacitor C5. For alternating voltage, the capacitive impedance is low and provides high AC gain determined by resistors R13 and R14. The output of the LM358 is giving the amplified value that will fetch to the ADC of microcontroller.

Also, audio input can be given through 3.5 mm audio jack. Since the audio input involves the negative voltages, therefore a clamping circuit is cascaded in front of audio jack to set the offset to 2.5 V. 10uf capacitor is added to remove noise. The output of it is connected to the same analog channel through a selection jumper (JP5), which helps in selecting audio input either from MIC or 3.5 mm audio jack.

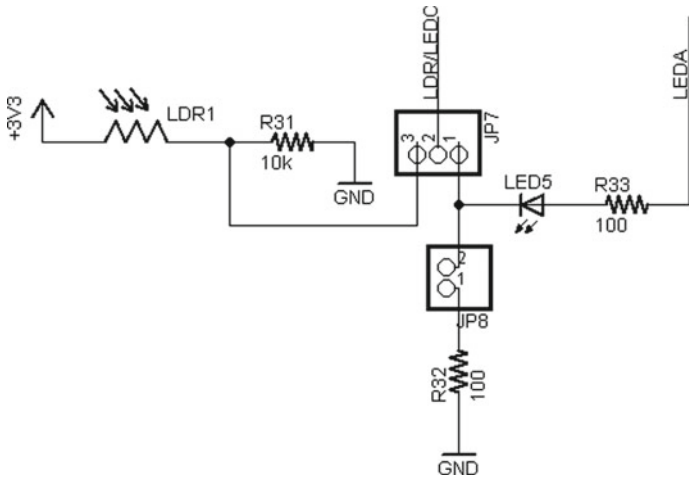

Fig. 6 Light intensity sensing using LED or LDR

4.3 Light Sensor

Light intensity can be sensed on the PadmaBoard by either using LDR (Light Dependent Resistor) or LED in reverse bias using the jumpers JP7 and JP8 as depicted in Fig. 6. LED can be connected in reverse bias and forward bias as per jumper (JP7) position. When LDR is selected jumper JP7 and jumper JP8 is placed. With this configuration of jumpers LDR will be used for Light sensing and LED can be used in forward bias. If jumper JP7 is connected towards LED side and jumper JP8 is removed, then LED2 can be put in reverse bias and will be used in measuring the light intensity. When the LED is in reverse bias its internal capacitance charges to the voltage applied. It is then connected to an input pin, which measures the time until the voltage across the internal capacitance becomes zero again. This measure of time/counter is inversely proportional to the light falling on the LED. Resistance R32 is protection resistor when JP7 is connected toward LED side and cathode of pin is high, it will not short logic high to GND.

4.4 I2C Bus Connector and Magnetic Field Sensor

Multiple I2C slaves can be connected to the same I2C bus having a condition that each slave should have a different slave address. This slave address is of 7-bit, so there are 128 different slave addresses possible. On PadmaBoard, there are three I2C slaves present on the I2C module 0, so additional number of I2C slaves can be connected. Example, if there is an accelerometer (whose slave address does not coincide with the existing slaves) can be added to the same I2C module 0. So a 5

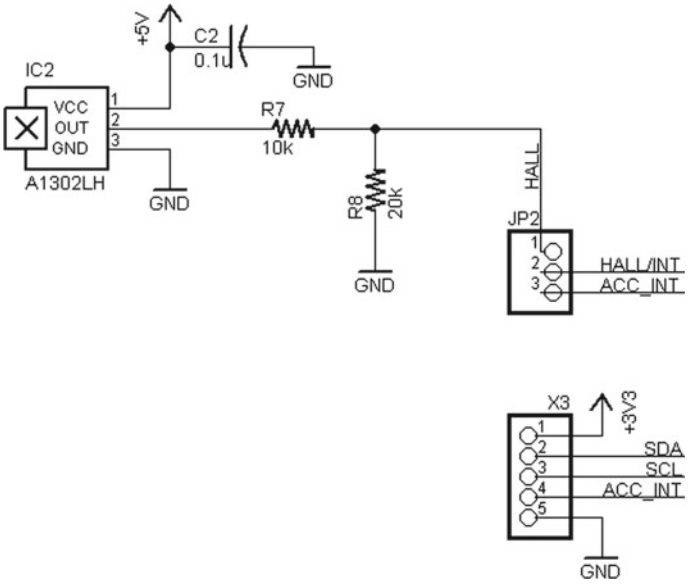

Fig. 7 I2C bus connector and hall effect sensor

pin connector as shown in Fig. 7 is given 3.3 V, SDA, SCL, GND and one GPIO. This GPIO can be used as an interrupt pin of I2C slave attached. This GPIO pin is multiplexed with the analog input of hall effect sensor through jumper JP2. The analog output of hall effect sensor varies from 0 to +5 V, so, resistance divider is applied to scale down the output variation to 0 to +3.3 V.

4.5 IR Transmitter and Receiver; and Ultrasonic Sensor Connector

Ultrasonic module is used to measure the distance of objects, it offers noncontact range detection with high accuracy and stable readings. However it is affected by temperature of the surrounding, but with the help of on board temperature sensor temperature compensated readings can be obtained. A 4-pin connector given on PadmaBoard is compatible with Ultrasonic module HC-SR04. However, there is one slide switch which gives the +5 V to connector X5, where a laser can be mounted just point the direction where it is measuring it.

IR LED as transmitter and IR TSOP (34xxx) receiver is used to perform remote IR operations. However, the output of TSOP receiver is multiplexed with the trigger pin of ultrasonic module through a jumper JP3. And, the IR Led is multiplexed with the Echo pin of ultrasonic module through a jumper, JP4 as shown in Fig. 8.

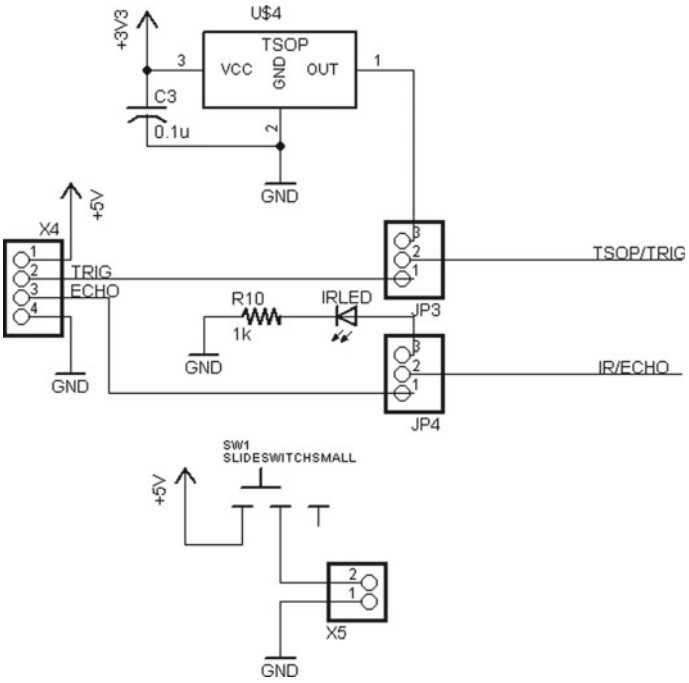

Fig. 8 IR LED and TSOP receiver; and ultrasonic module connector

4.6 Buzzer

Buzzer is used to generate tones of various frequencies. The current requirement of buzzer is bit more so, it is connected through a transistor as depicted in Fig. 9. Also, buzzer itself has around 55 ohms of internal resistance which makes around 60 mA of current flow through buzzer giving desirable volume of tone generated by buzzer.

4.7 Three LEDs

PadmaBoard has three LEDs connected to same pins as RGB LED on the Tiva LaunchPad. These three LEDs are connected through transistors in active high configuration. These three separate LEDs provide better visualization of count. For example, in VU meter the intensity of sound can be mapped on these three LEDs. The schematic for these three LEDs is shown in Fig. 10.

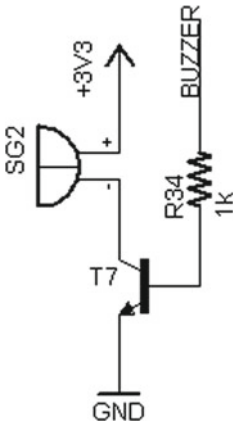

Fig. 9 Buzzer

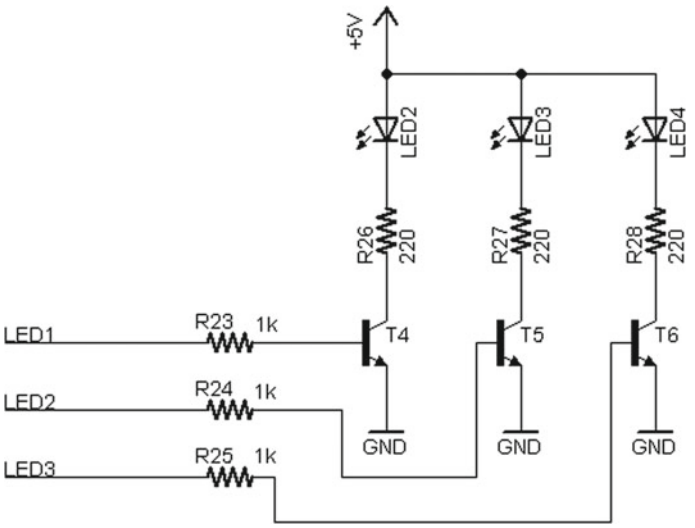

Fig. 10 Uni-color LEDs on PadmaBoard

4.8 Serial Communication Port Using UART Protocol

A serial communication port, X6 is given whose connector as shown in Fig. 11 is fully compatible with Bluetooth module HC-05 which is easily available in the market, to perform wireless operations. This connector has 2 UART pins and 1 GPIO pin. Also, other UART based communication links can be established through this port example RS232 or RS485.

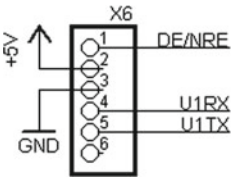

Fig. 11 UART connector (fully compatible with Bluetooth module HC-05)

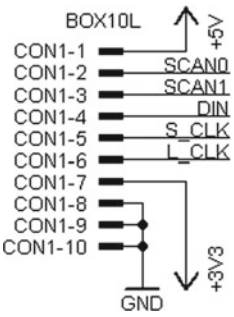

Fig. 12 5×2 box connector on PadmaBoard

4.9 Serial LCD with 16 Keys Keypad

There is 5×2 box connector to connect 16×2 LCD and 16 keys keypad to PadmaBoard. The 16×2 LCD and 16 keys keypad are driven by serial data stream. LCD and keypad are driven using serial data to save the number of GPIO pins. 16×2 LCD require 6 GPIO pins (in 4-bit mode) and 16 keys keypad requires 8 GPIO pins (when arranged in 4×4 matrix). Total number of GPIO pins require to drive them is 14 pins. Whereas by driving them using serial data number of GPIO pins required can be reduced significantly.

On the LCD keypad board serial data passed through the shift register, 74HC595 which converts that serial data to parallel data and sends to LCD and Keypad. The 16 keys of keypad are placed in 8×2 fashion so that there are 2 scan lines for the 16 keys keypad. The box connector will be supply +5 V from USB connector, GND, +3.3 V, serial data (DIN), serial clock (S_CLK), Latching pulse (L_CLK) and two scan lines from keypad (SCAN0, SCAN1) as depicted in Fig. 12.

On the LCD Keypad serial board as shown in Fig. 13 there are 16 40xx omron buttons, which are pulled down to GND through resistance R1 and R2. The supply voltage of the shift registers 74HC595 (IC1, IC2) can be connected with either +5 V or +3.3 V through a three way smd jumper. Capacitors C3 and C4 are decoupling capacitors for IC1 and IC2. 16×2 LCD is supplied with +5 V and GND as supply voltage. Pot PR1 is used for the contrast adjustment of the LCD.

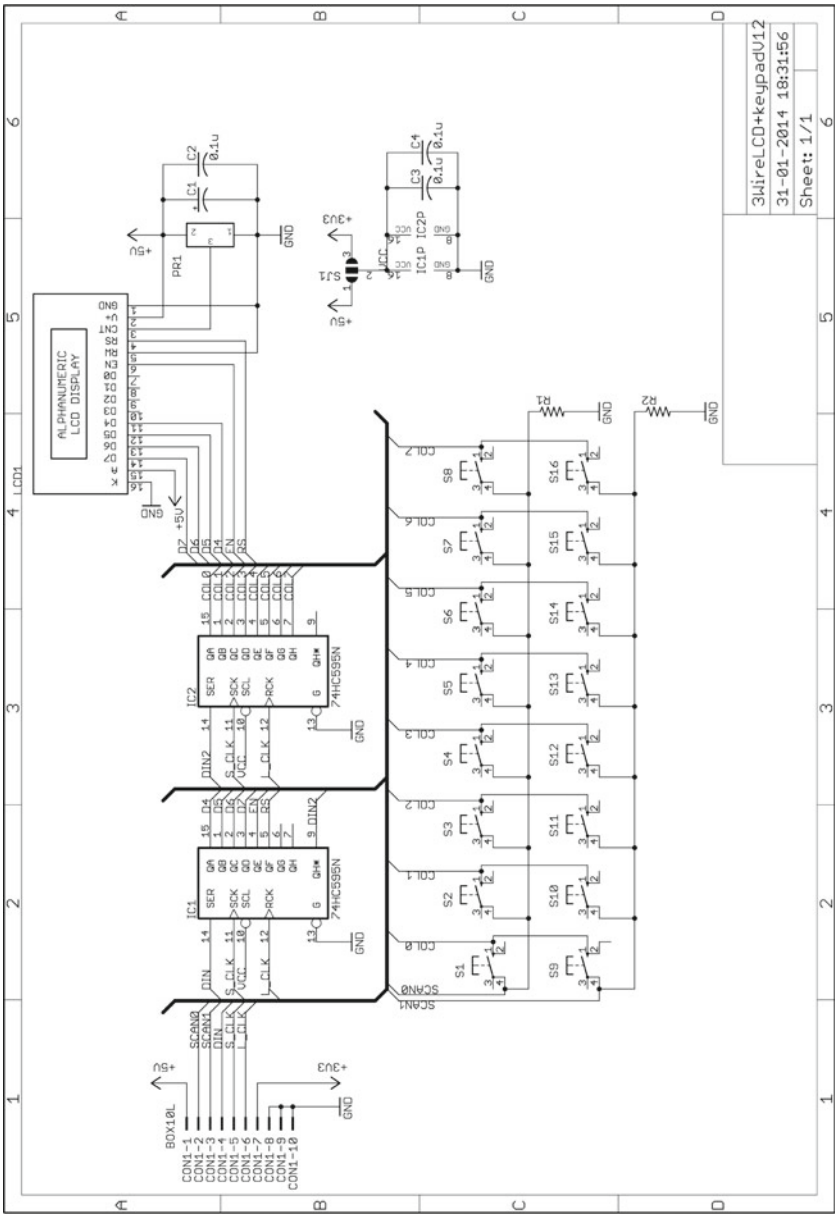

Fig. 13 LCD and keypad with 5-wire serial interface

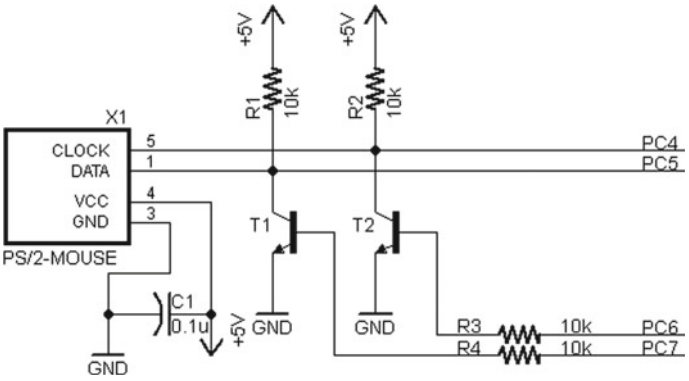

Fig. 14 PS/2 connector

4.10 PS/2 Connector

PS/2 connector implements a bidirectional synchronous serial protocol. To implement the bidirectional flow of data 4 GPIOs of microcontroller and 2 transistors are used as shown in Fig. 14. GPIOs connected to the collector end of transistors are configured as input and GPIOs controlling the base voltage of transistors are configured as output, in this way two bidirectional lines are obtained. PS/2 keyboard or PS/2 mice can be connected to this on board PS/2 connector.

4.11 MicroSD Card Interface

MicroSD card interface is essential when there is need to store large data which can be easily accessed by PC or mobiles. Or, to process some large data which is stored in microSD card and can be read during processing. MicroSD card interface follows SPI protocol and is connected to SSI module 0 of the microcontroller as shown in Fig. 15.

4.12 TV and Potentiometer

On PadmaBoard, television unit can be connected as display device. Television can be plugged to on board RCA jack which is connected through a 2 bit DAC using 2 resistors R5 and R6 and the input impedance of the television set (usually around 75 ohms). Two-bit DAC is controlled by the MOSI pin (SSI1TX) of SSI module 1 and its chip select pin SSI1FSS.

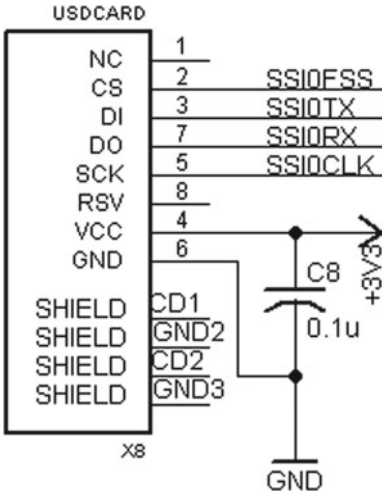

Fig. 15 MicroSD card interface

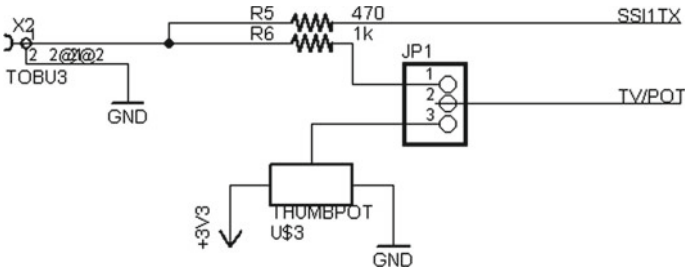

Fig. 16 TV RCA jack and potentiometer

The Chip select pin is also an analog pin, so it is connected to thumbwheel potentiometer through a jumper, JP1. Through a potentiometer voltage can be varied between 0 and 3.3 V. This schematic is depicted in Fig. 16.

4.13 Real-Time Clock (RTC)

RTC follows I2C protocol and is connected to I2C module 0 of the microcontroller as shown in Fig. 17. RTC used is PCF8563 having 32.768 kHz as external oscillator. Supply for RTC is either from +3 V 20 mm lithium battery or +3.3 V from Launch-Pad. Capacitor C10 is decoupling capacitor for RTC PCF8563. RTC interrupt pin is multiplexed with the SCAN0 line of 16 keys keypad through a PNP transistor (T3).

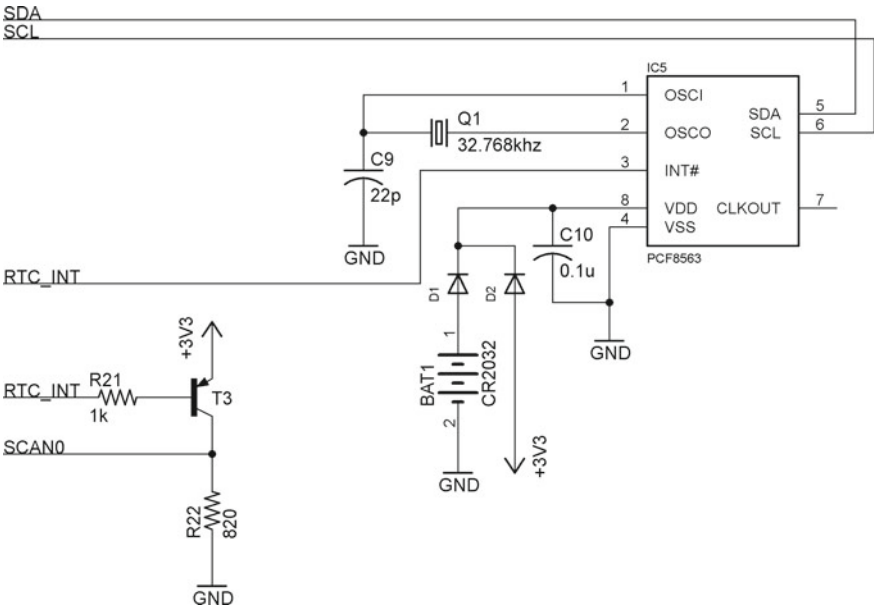

Fig. 17 Real-time clock (RTC)

4.14 Dual DAC with Audio Out

Two 12-bit I2C-based DAC 7571 (IC6 and IC7) are used and connected to I2C0 module of the microcontroller as depicted in Fig. 18. The address of each two DAC7571 can be unique by using address A0 external pin. So, the slave address of IC6 is 0×4C and for IC7 is 0×4D. These two DAC can be used to implement Lissajous figures on the dual channel oscilloscope in X-Y mode. And output of DAC (IC7) is also connected to the 3.5 mm audio jack, which is used to generate audio output.

5 Jumper Selection

Since, many peripherals are multiplexed with jumpers. So, the knowledge of proper placement of jumpers before doing the experiments is essential. There are 8 jumpers which need to be selected while performing the experiments related to them.

1. Jumper, JP1 multiplexes the TV output signal with the potentiometer (or thumbpot). The position for the jumpers to select between them is shown in Fig. 19.
2. Jumper, JP2 multiplexes hall effect sensor output with interrupt coming from I2C bus connector, X3. Figure 20 represents the proper placement of jumper for selection.

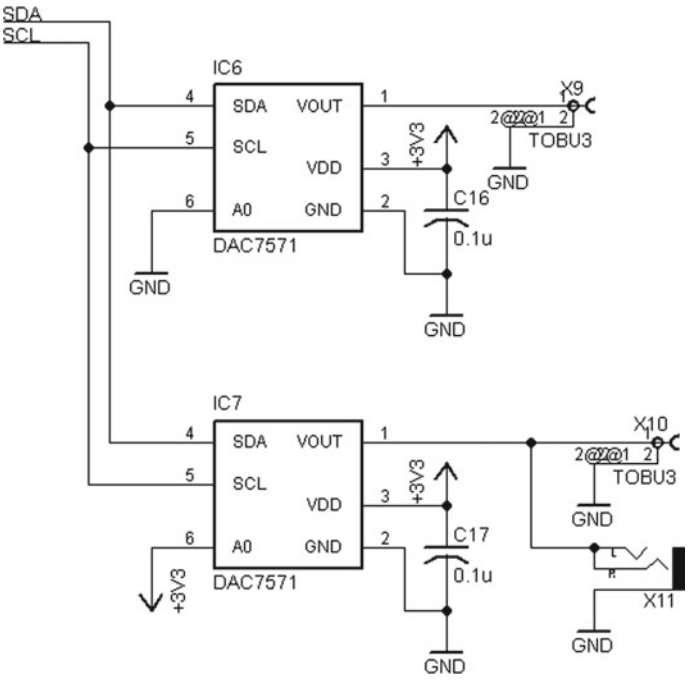

Fig. 18 Dual DAC with audio output

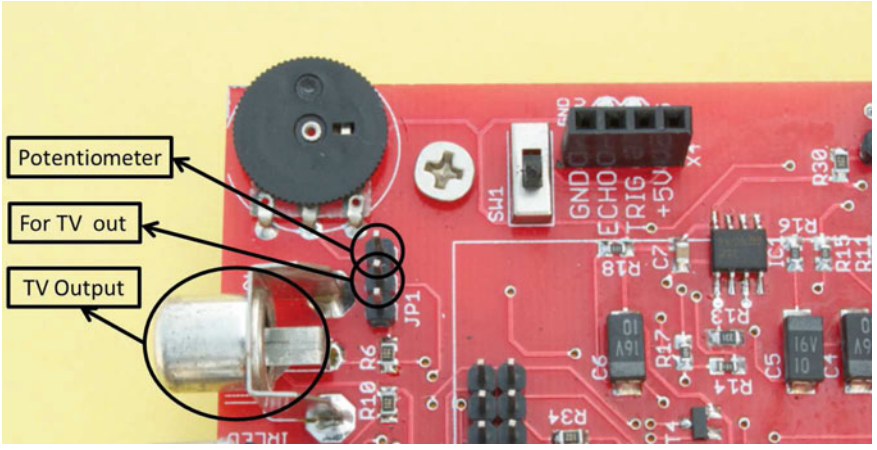

Fig. 19 Jumper selection between TV output and potentiometer

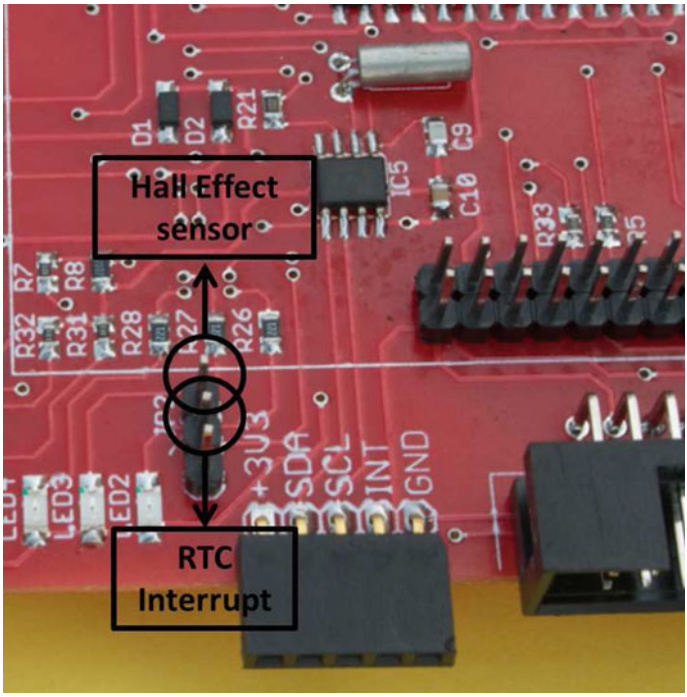

Fig. 20 Jumper selection between hall effect sensor output and interrupt from I2C bus connector, X3

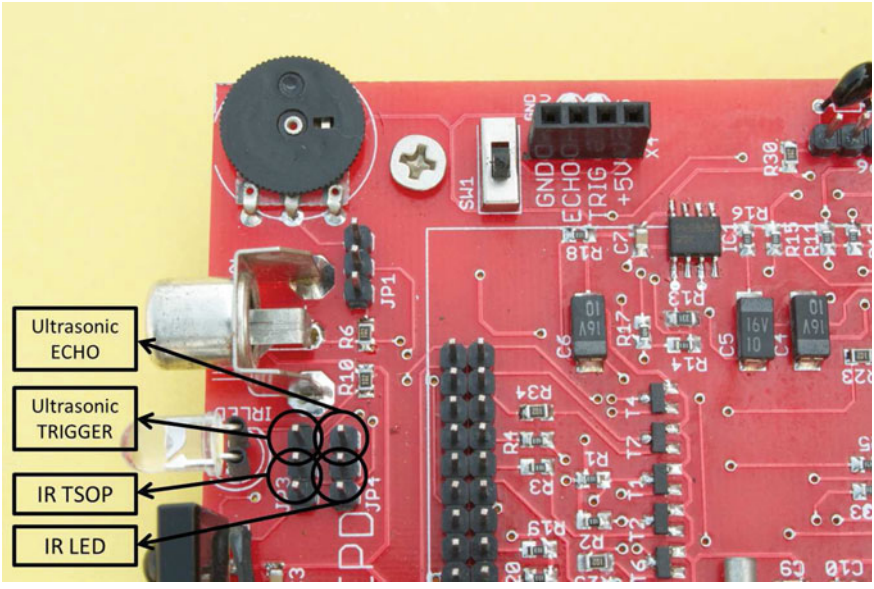

Fig. 21 Jumper selection between IR LED, IR TSOP and ultrasonic module

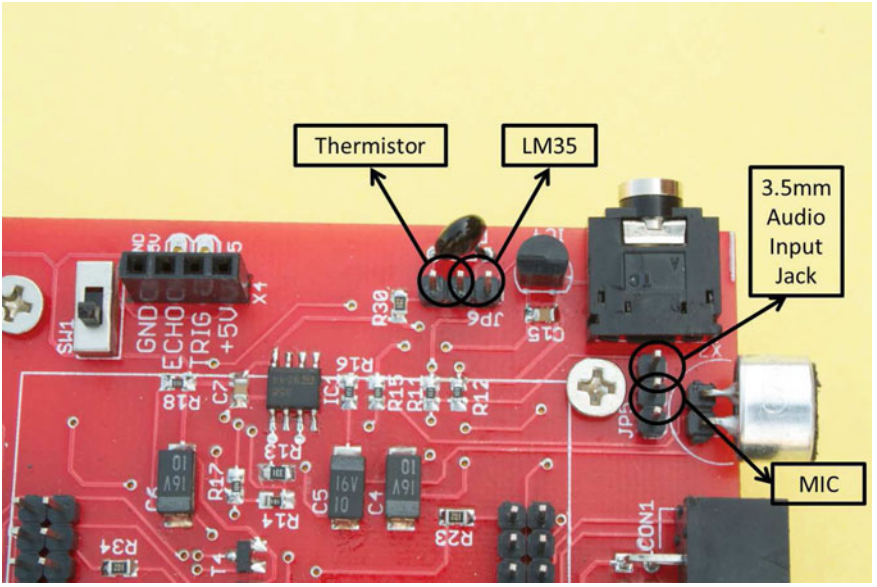

Fig. 22 Jumper selection for temperature and audio input

3. Jumper, JP3 multiplexes IR TSOP output with the trigger pin of the ultrasonic module. And, jumper, JP4 multiplexes IR LED with the echo pin of the ultrasonic module. Figure 21 represents the proper placement of jumpers for selection.

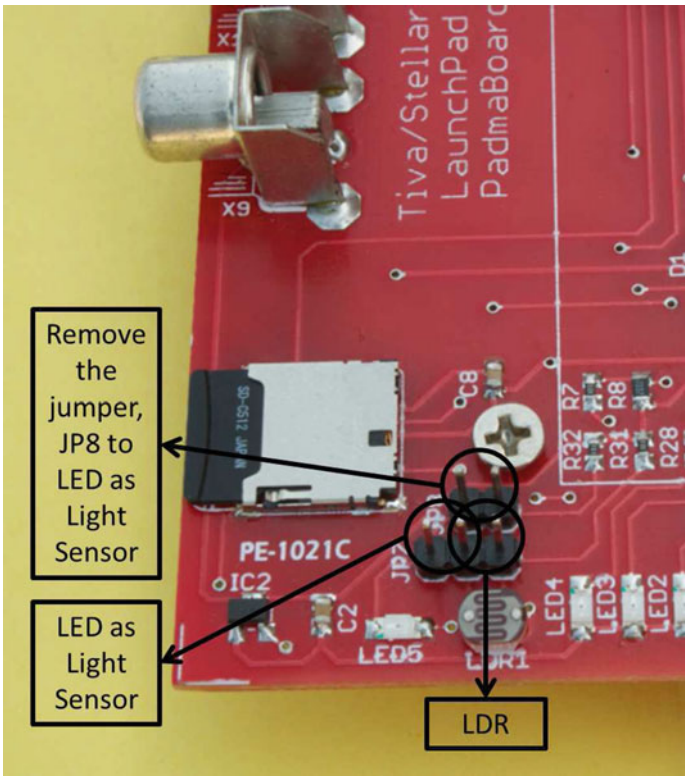

Fig. 23 Jumper selection for using LDR or LEd as light sensor

4. Jumper, JP5 multiplexes audio input either through MIC or 3.5 mm audio jack. And, jumper, JP6 multiplexes temperature input either from LM35 or from thermistor. The placement for selection jumpers is shown in Fig. 22.
5. Jumper, JP7 multiplexes light intensity measurement either through LDR or LED as light sensor. And, jumper, JP8 is to enable the LED5 in forward biased mode. The placement for selection jumpers is shown in Fig. 23.

Chapter 5
Tiva C Series Microcontroller Breakout Board

This chapter focuses on hardware basics for the development of standalone projects using Tiva C Series microcontrollers. By standalone project, it refers to the project which does not use any evaluation board. So, this helps the developer to choose microcontroller as per their requirement and can develop their own customized system using that microcontroller. To perform experiments related to standalone projects, a breakout board of the Tiva C series microcontroller using TM4C1231H6PZ, a 100 pin LQFP (Low-Profile Quad Flat Package) is referred in this manual. This microcontroller do not have the advanced features like motion control PWM or USB controller, but it has all the basic features of Tiva C Series family of microcontroller which uses ARM Cortex-M4F processor. Also, the pin out of all 100 pin LQFP microcontroller belonging to Tiva C Series family are same. So, any of the microcontroller belonging to Tiva C Series family can also be soldered on the breakout board PCB instead of the TM4C1231H6PZ. Though the breakout board, PCB has also been tested with TM4C123GH6PZ microcontroller (which supports almost all of the advanced features) in addition to previously mentioned TM4C1231H6PZ.

1 Board Overview

Breakout board is just not the extension of microcontroller pins, but it has all the necessary hardware requirements for the microcontroller to operate like 20 MHz crystal for internal main clock source, 32.768 KHz crystal for Hibernation module clock source, reset switch and the decoupling capacitors. In addition to mentioned essential components, breakout board has programmable features on it like LED and two user switches. To program or debug the onboard microcontroller JTAG debug signals, UART RX and TX, and power lines (+3.3, +5 V, and GND) are mapped to the pins of box connector (5×2) or debug connector. With the presence of such all important and minimal circuitry available on breakout board, the developer can

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_5

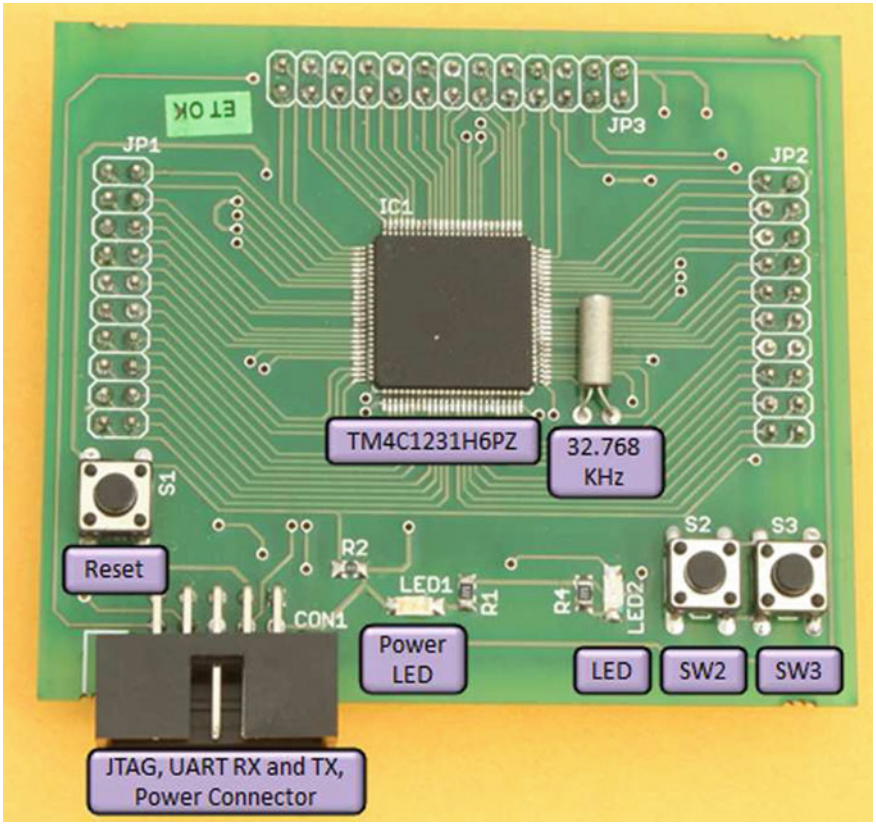

Fig. 1 Tiva breakout board (top view)

focus on additional hardware or features required to develop the system (application specific) around the microcontroller. Figure 1 shows top view of the breakout board and Fig. 2 shows bottom view of the breakout board.

The developers can even prepare their own breakout boards as per their choice over the microcontrollers. For example, if developer is using microcontroller A for one prototype and microcontroller B for another prototype, then he or she can design the breakout board which will be generic to both the microcontroller A and B. This will reduce the overall prototyping cost. Also from other perspective, developers need not buy the evaluation kits which can further lower the prototyping cost. Instead of evaluation boards, developers can have the breakout board which will be generic to most of the microcontrollers.[1] And once they have developed the breakout board and tested it, they can easily focuses on development of the system around it.

[1]The breakout board can be utilized for other microcontrollers only if they have same pin mapping.

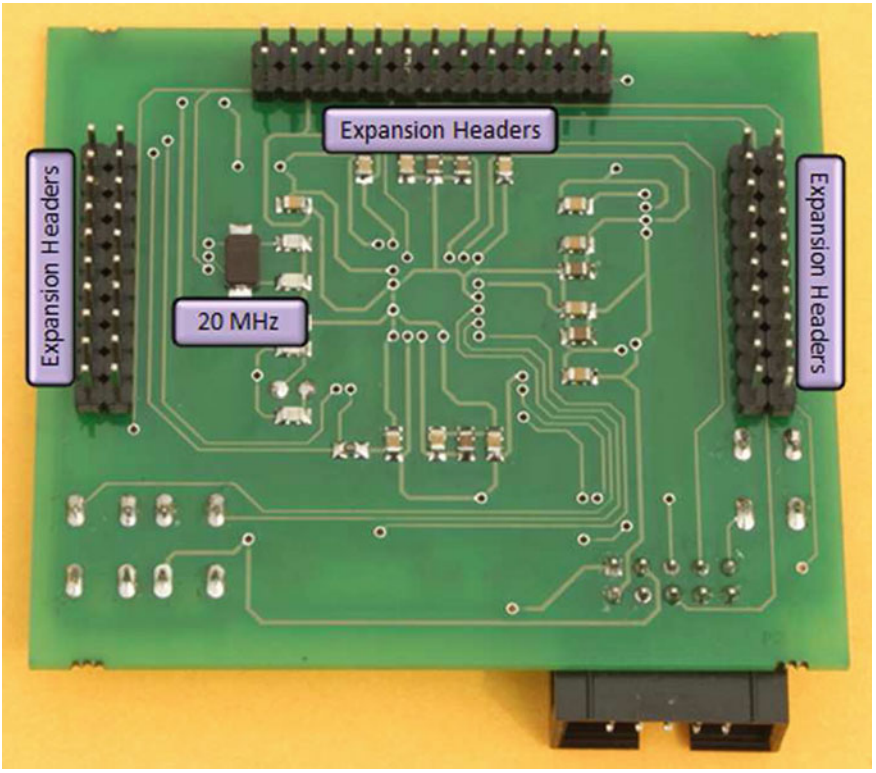

Fig. 2 Tiva breakout board (bottom view)

2 Schematic and Layout

Figure 4 shows the schematic of breakout board for TM4C1231H6PZ. PCB designed for the breakout board is double sided PCB. Figure 3 represents the top layer of PCB or component side of PCB.

3 Hardware Description

3.1 Power Supply

Breakout board does not have an onboard power regulator, so it takes regulated power input from connectors. The breakout board can be powered using debug connector which has pins for +5, +3.3 V and GND. Also, extension headers have +5, +3.3 V and GND pins, so that whenever the daughter board for breakout board is designed,

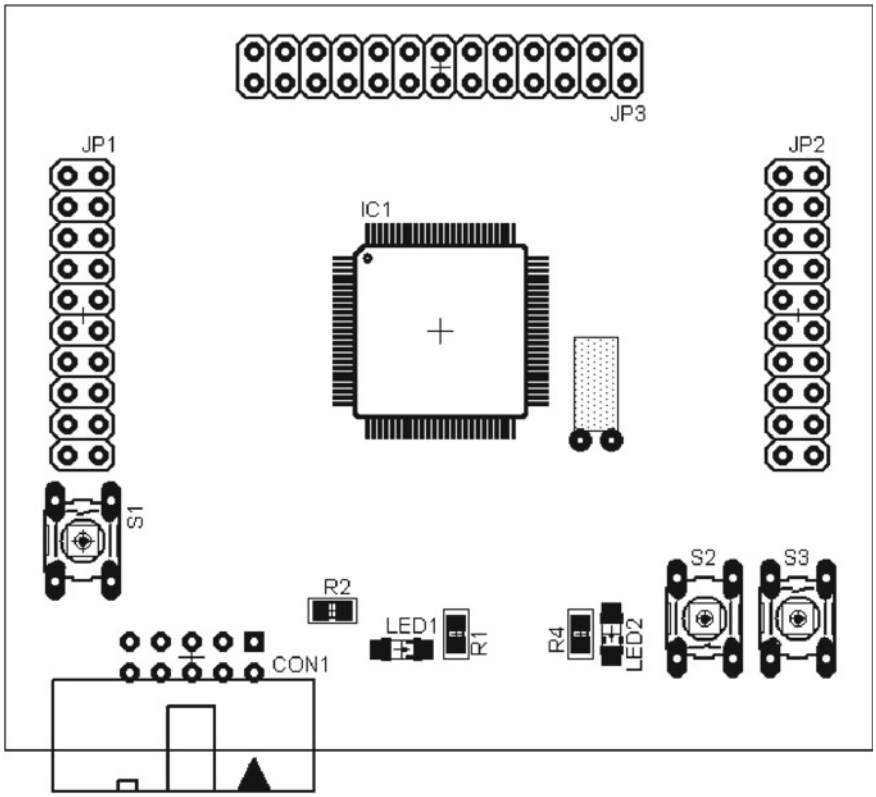

Fig. 3 Tiva breakout board PCB—component side

it can be powered from that daughter board. It is necessary for the breakout board to have a common GND between it and the JTAG debugger. Also, it is important that breakout board is powered either through debugger or through daughter board, not from both at the same time.

3.2 Hibernate

Microcontroller on breakout board supports the Hibernation functionality. In hibernation mode when the processor and the peripherals are idle, then the power can be removed with only hibernation module powered up. The hibernation module can be powered up using external battery or the auxiliary power supply. But on the breakout board "VBAT" pin for external battery source has been pulled up with 3.3 V. Power can be restored based on the WAKE signal (extended to the expansion headers) or

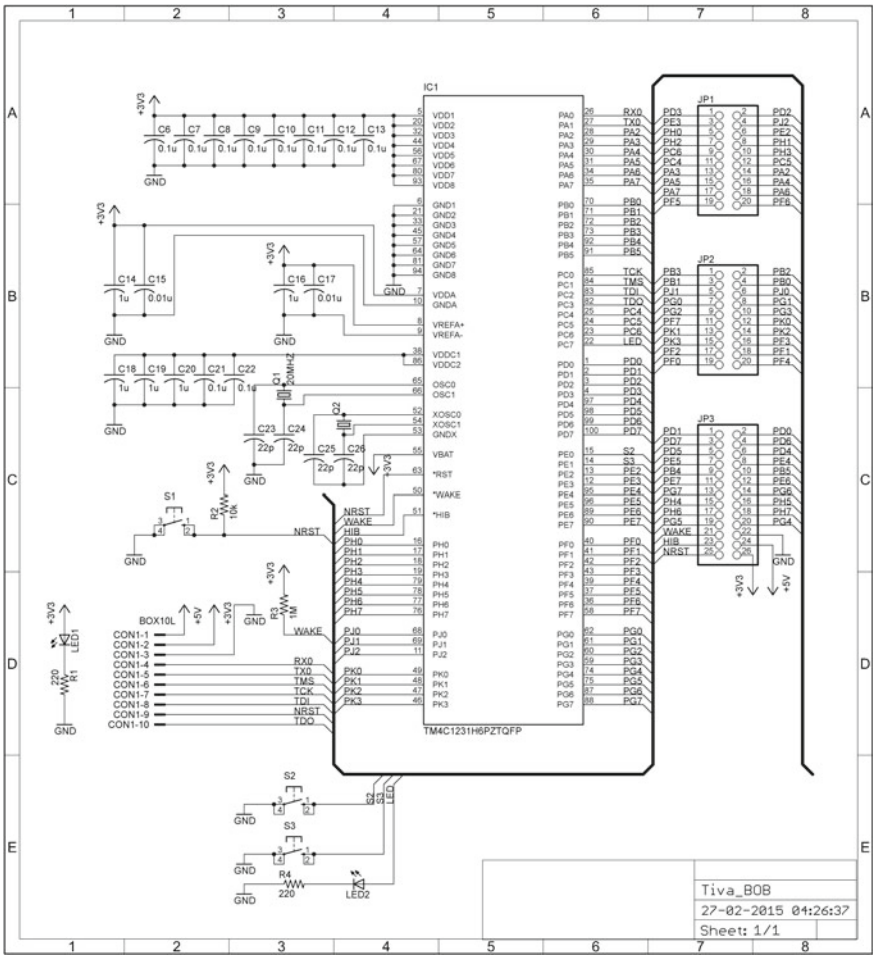

Fig. 4 Tiva breakout board schematic

after a certain time by using built-in real-time clock (RTC). Also, breakout board has 32.768 KHz crystal as clock source input for the Hibernation module.

3.3 Clock

The breakout board has 20 MHz crystal as clock source input for the internal main clock. An internal PLL, can be configured by the software to generate different frequencies for the core ranging from 3.125 to 80 MHz. The hibernation module has a different clock source, and uses external 32.768 KHz crystal.

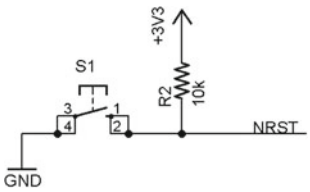

Fig. 5 Tiva breakout board reset

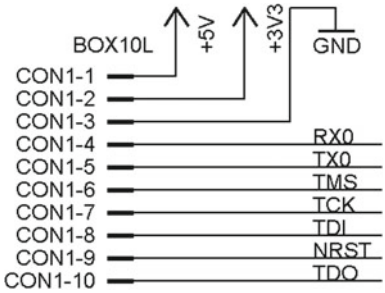

Fig. 6 Debug connector pin mapping

3.4 Reset

The reset signal of microcontroller, TM4C123GH6PM is active low and is connected to reset switch, S1 and also to debug connector for a debugger-controlled reset. The reset signal is pulled up using 10 kohm resistor as shown in Fig. 5, so that the microcontroller resets only when reset signal is triggered. External reset can be asserted under three conditions

- Power-on reset
- RESET switch pressed
- When instructed by the external debugger

3.5 Debug Connector

The breakout board has a 5×2 box connector. All debug pins of JTAG and SWD are mapped to this connector. Also, RX and TX pins of UART Module 0 is mapped to this connector. The UART lines can be connected to the USB-UART bridge of Tiva LaunchPad to communicate with host PC. The pin mapping of debug connector is shown in Fig. 6.

Table 1 User switches and LED pin mapping

S. No.	GPIO pin	Pin function
1.	PE0	User switch, S2
2.	PE1	User switch, S3
3.	PC7	LED

3.6 LED and Switches

The breakout board has two user switches, S2 and S3 and one programmable LED, LED2. These two essential features can help in determining whether the breakout board is in working condition or not. The breakout board also has the power LED, LED1 which indicates that 3.3 V power is available on the board. Table 1 shows pin mapping of user switches and LED on the breakout board.

3.7 Microcontroller Expansion Headers

The breakout board has three double rows of headers, JP1, JP2, and JP3. Among these two are 20 pin (10×2) headers and one is 26 pin (13×2) header. These headers are extension for the GPIO pins of the microcontroller. Total of 60 GPIOs are extended to these expansion headers. Along with these GPIOs power supply +5, +3.3 V and GND are also present on the expansion headers. The daughter board designed for the breakout board will be plugged through these expansion headers. The expansion headers are shown in Fig. 2.

4 Programming Tiva C Series Microcontrollers

The Tiva C Series microcontroller can be debug using JTAG debugger or serial wire debugger. The onboard ICDI of the Tiva LaunchPad can be used as debugger. Also, there are bridges available like FT2232 (from Future Technology Devices International Ltd.) which convert the USB data to the JTAG data or UART data or Serial Wire Debug (SWD) data signals. These bridges are programmable and they need to be uploaded with the configuration file into them. Generally, a EEPROM is used along such bridges which stores the configuration file regarding the conversion of data. In this chapter, Tiva C Series TM4C123G LaunchPad is used to program the breakout board and also used as USB-UART bridge to communicate with host PC. Follow the steps mentioned below to use Tiva LaunchPad as external debugger for Tiva breakout board:

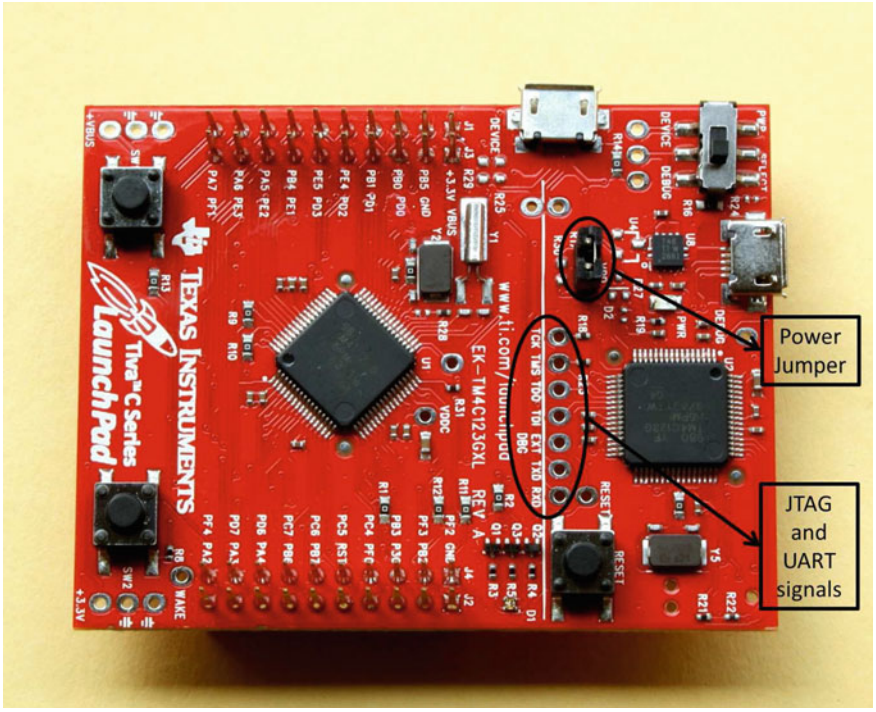

Fig. 7 Tiva LaunchPad as debug out

- Remove the power jumper from the Tiva LaunchPad present just below green power LED as shown in Fig. 7. By removing the power jumper, Tiva LaunchPad target microcontroller is no more powered and will not get programmed or corrupted by the JTAG signals which are meant for the breakout board. Also, it prevents Tiva LaunchPad target microcontroller from interfering with the JTAG signals.
- Connect the JTAG signals from Tiva LaunchPad to the breakout board along with common ground. It may need to solder row of headers adjacent to the reset switch on the Tiva LaunchPad as shown in Fig. 7. Connect TCK, TMS, TDO, and TDI of the breakout board to the Tiva LaunchPad pinouts as mentioned on the silk screen of Tiva LaunchPad.
- Connect the UART signals if required. Connect the TXD (of Tiva LaunchPad) to the UART TX pin, PA1 of breakout board microcontroller and RXD (of Tiva LaunchPad) to the UART RX pin, PA0 of breakout board microcontroller. These TXD and RXD are connected to UART RX and TX pins respectively of the ICDI microcontroller of Tiva LaunchPad.
- Now program the Tiva LaunchPad using LM Flash Programmer (select the Manual Configuration).

Chapter 6
GNU ARM Toolchain

This chapter deals with the setting up of the environment for programming Tiva C Series ARM Cortex-M4 based microcontrollers. The same methodology can be applied to program the other ARM-based microcontrollers of different families. Usually, the semiconductor companies who designed these microcontrollers also provide a programming platform or environment along with their microcontrollers as the software support, like Code Composer Studio from Texas Instruments to program its products like Tiva, MSP430, C2000, etc. But having an understanding of setting up a programming environment helps in migrating from one microcontroller to another easily. Though the program codes and code snippets which referred in the further chapters are compatible with both Code Composer Studio and the custom programing environment, explained in the chapter as both the environments are referring to the same set of libraries.

1 Introduction

Since, microcontrollers are digital in nature. The basic language these microcontrollers interact is in 0 and 1. Hence the program code for them must also be in binary, i.e., only in 0 and 1. But writing the code in 0 and 1 is bit difficult, so the code is written in high-level languages[1] like C, C++, python, etc. And this program code written in high-level language is then compiled by the compiler into a binary which the microcontrollers can understand.

[1] High-level languages are easier to understand as compared to the machine language. They are generally preferred because they help the programmers to concentrate more on application development without worrying much about the underlying abstraction layers.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_6

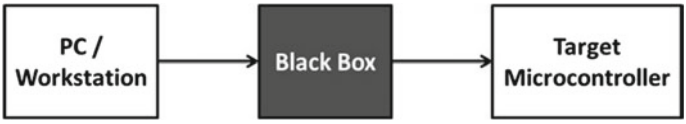

Fig. 1 Black box model for the Tiva C series microcontroller programming environment

As shown in Fig. 1, consider a black box which is converting user code written in high-level language into the final executable file which will be programmed in the microcontroller.

So, subsequent sections will discuss about the inside components of black box like compilers, linkers, assemblers which convert the high-level language code to machine language. All the inside components will be discussed in terms of their individual roles and integration of them to create the programming environment. In general, all the necessary steps are needed to compile the code for the microcontrollers of family ARM Cortex-M. After that, step-by-step instructions are mentioned to set up the programming environment for Tiva C Series Microcontroller family.

2 Programming Environment Components

The black box mentioned above can be divided into two parts

1. **Integrated Development Environment (IDE)**—The IDE is the front end of the black box and is the user interface for the application code editor or project explorer of the application.
2. **IDE Plug-ins**—As, IDE is a very generic tool, which can be used for application development for various platforms like Android, ARM-based microcontrollers, web-based applications, etc. So, installation of the plug-in with IDE is required to manage the projects related to development platform. For example, in this manual GNUARM Plug-in is used for managing ARM-based projects in Eclipse IDE.
3. **Toolchain**—The toolchain is running on backend. It is the main component of the black box, as it is responsible for the conversion of application code in high-level language to machine language through a series of compiler, linker, and assembler. It carries out the build process of the project, i.e., everything from preprocessing, compiling, assembling, and finally generating a executable file, which can be programmed to the microcontroller.
4. **Flash Tool**—After generating the executable file, it need to be transferred into the target hardware. Hardware manufactures do provide such software free of cost, however they hide the underlying development layers of such software to keep it proprietary.

Figure 2 shows the build process of the Tiva C Series microcontroller family. The steps for the build process of the project are briefly described in the subsections below.

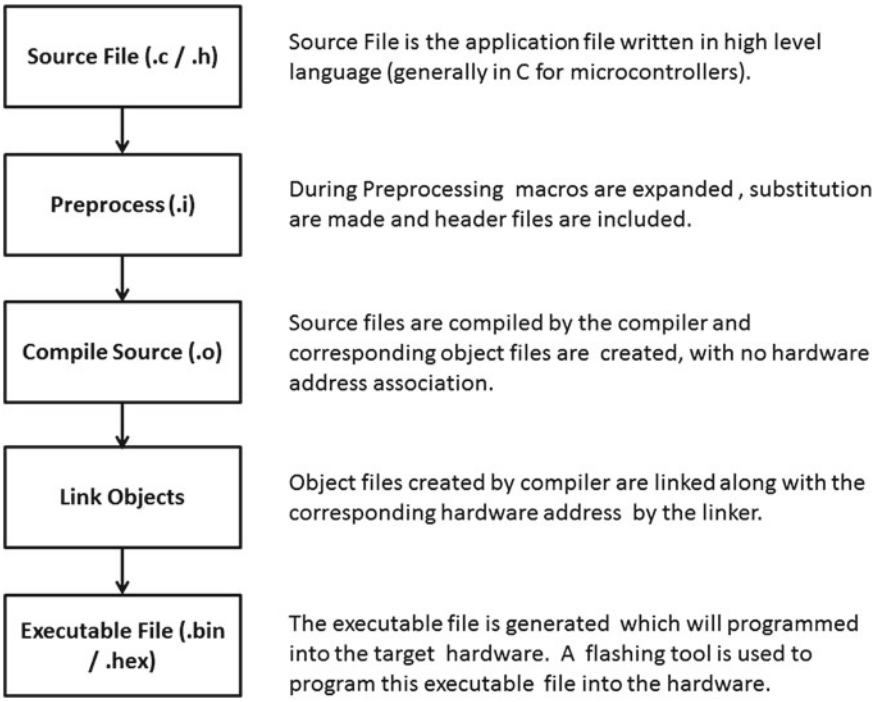

Fig. 2 Build process for Tiva C series

2.1 Preprocessing

Preprocessing is the step which is done before the compiling of the source file. It mainly performs the following tasks:

- Macro-expansion and substitution
- Conditional Compilation[2]
- Inclusion of Header Files

The C preprocessor is used in building C projects, and is known as CPP. All the preprocessor directives start with a pound symbol(#) like #define, #include, #ifdef, #undef, etc.

[2]In Conditional Compilation, source code which does not fullfill the condition is ignored and rest is compiled. Examples of conditional compilation in C are "ifdef" statements. It generally provides a configurability feature to the application development.

2.2 Compiling

A compiler is a special program that converts the source code generally written in high-level language into a assembly code. Depending on the compiler, it can also optimize the code in terms of size and execution speed. When compiler is used to compile, the source code which will be running on the platform different from the platform on which it is compiled, is called cross compiler. Almost all the programming environment for microcontrollers use cross compilers as the host platform in which the application source code is developed is different than the target hardware.

2.3 Assembling

Assembly language is low-level language but still it need to be converted into machine language or object files which will be understandable by the hardware. the tool which perform this task is called as assembler. Hence the object file which is generated is sequence of machine instructions that correspond to the high level language program.

2.4 Linking

A project may contain more than one source code. As a source code may refer to a variable or a function which are declared and defined in other source code files. So on compiling the object files are created for each source code file. These references to external variables and functions among the objet files must be linked up or properly resolved. This is task of the linker and is known as linking.

3 Programming Environment for Tiva C Series Microcontroller Family

The Black Box model of the programming environment is discussed above. Now, this section focuses on the development of the programming environment for Tiva C Series Microcontroller family.

- **Eclipse as IDE for C/C++ Developers**
  Eclipse is an open source integrated development environment with a rich frame work of Plug-ins. There have been lots of releases of Eclipse, this toolchain is tested with the release Eclipse Indigo. This IDE is GUI (Graphical User Interface) based for writing and editing the code. This IDE can be linked up with Sourcery GNU ARM Toolchain using the GNUARM plug-in.

- **Sourcery G++ GNU Toolchain for ARM**
  This toolchain is also known as Sourcery CodeBench Lite Edition. The Sourcery CodeBench Lite Edition includes the following features:

  - GNU C and C++ compilers
  - GNU assembler and linker
  - C and C++ runtime libraries
  - GNU debugger

  The different commands invoked during the build process with this toolchain are:
  arm-none-eabi-cpp [options] file... Invokes Preprocessor
  arm-none-eabi-gcc [options] file... Invokes C Compiler
  arm-none-eabi-g++ [options] file... Invokes C++ Compiler
  arm-none-eabi-as [options] file... Invokes Assembler
  Different vendors have different options to invoketh build process. Hence, having a detailed documentation regarding toolchain for corresponding vendor is a prerequisite to build process.
- **GNUARM Plug-in**
  As discussed above the plug-ins help in managing the projects in the IDE. Hence to manage ARM projects GNUARM Plug-in is used with the Eclipse IDE. It integrates the entire build process of a command line to a graphical interface.
- **Texas Instruments Development Package: (TivaWare + ICDI drivers + LM Flash programmer)** Texas Instruments Development Package include TivaWare, ICDI(In-Circuit Debug Interface) drivers and LM Flash Programmer. TivaWare is the peripheral driver library which is compatible with Tiva C Series family. The program codes in following chapters are tested with the TivaWare version 2.1.1.71 dated May, 2015. It contains three major directories:

  • **driverlib**—This directory contains definitions and declarations (.c and .h) of the peripheral drivers like ADC (Analog-to-Digital Converter), UART(Universal Asynchronous Receiver Transmitter), Timers, Interrupts, etc.
  • **inc**—This directory contains header files which have values of macros used in above mentioned driver libraries.
  • **utils**—This directory contains the definitions and declarations (.c and .h) of the drivers which are just the software part and no actual hardware unit regarding it exits, like lookup tables for sine, calculating square roots, etc.
  ICDI drivers are installed in host so that Tiva LaunchPad can be detected by the host. LM Flash Programmer is responsible for programming executable files in the target hardware.

4 Setting up the Development Environment

Install the above-mentioned components (except the ARM Plug-in for IDE, as the process to install ARM Plug-in with IDE is illustrated in this section.) before processing further. Now, open the Eclipse IDE,[3] a window similar to Fig. 3 comes up.

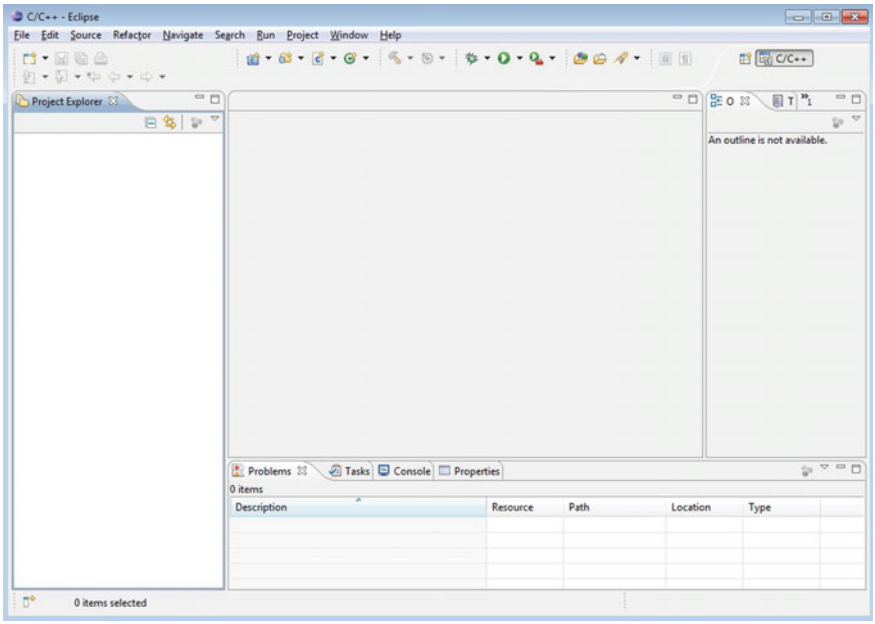

Fig. 3 The eclipse IDE

[3] Make sure that Eclipse IDE is extracted in the root directory, example C:\.

1. Installing GNU ARM plug-in

    a. Go to Help - Install New Software - Add - archive
    b. Browse to the GNUARM plug-in zip file.
    c. A window similar to that shown in Fig. 4 will show.

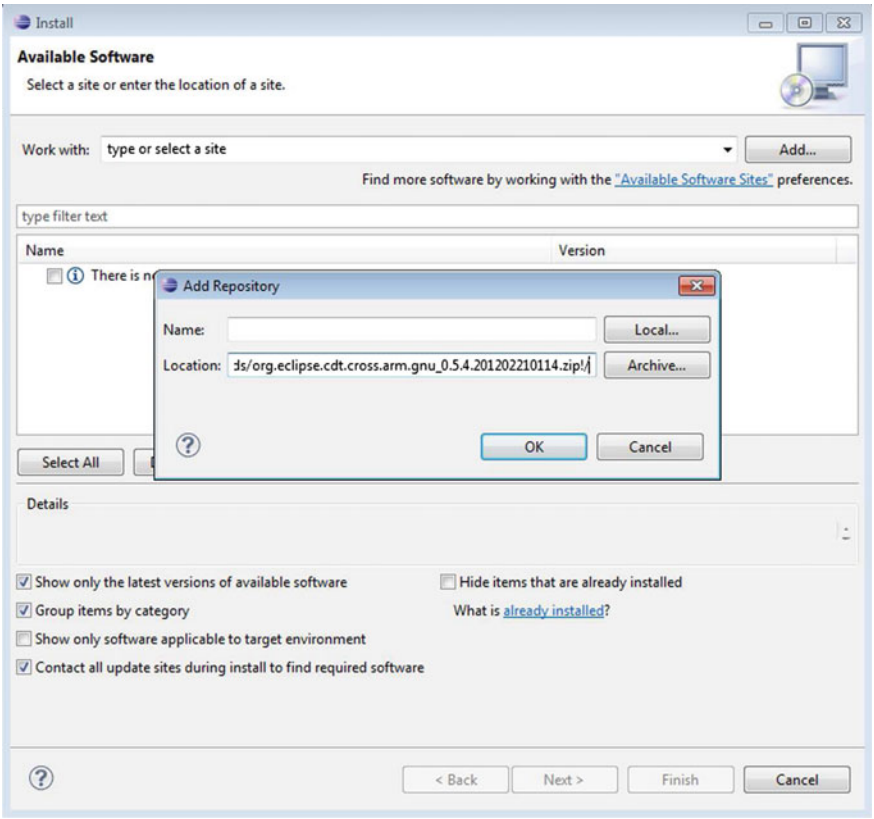

Fig. 4 Add plug-in repository

d. Click Ok and Next to install the plug-in.
e. A window similar to that shown in Fig. 5 will show.

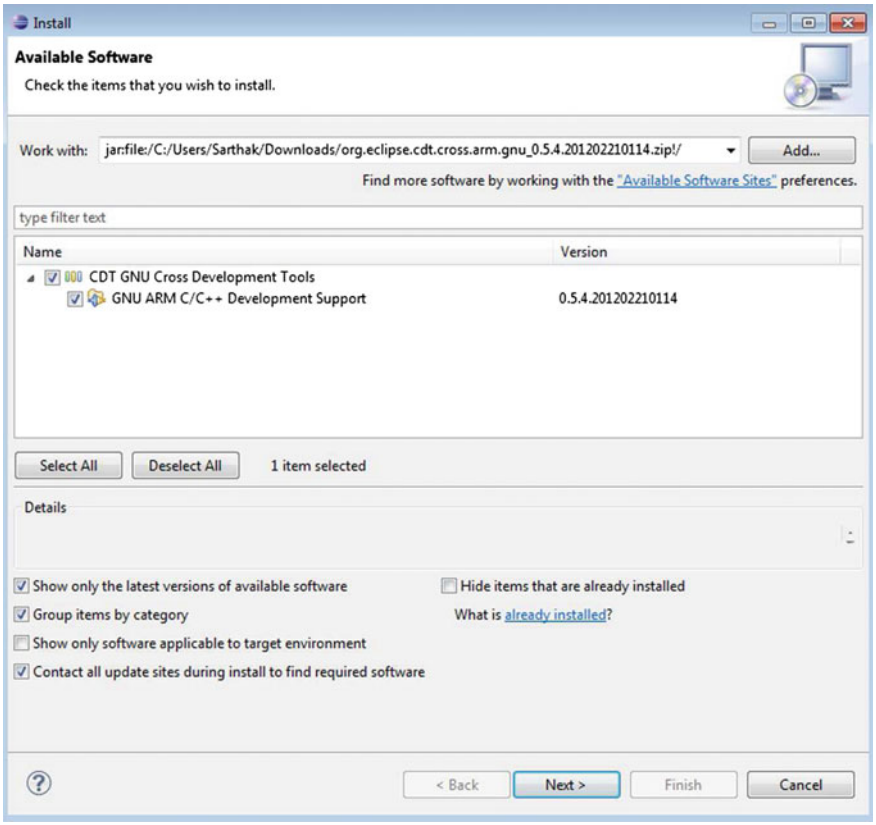

Fig. 5 Plug-in install option

2. Creating the first project

    a. Go to File - New - C Project.
    b. Enter the project name. Make sure the selections are shown in Fig. 6.

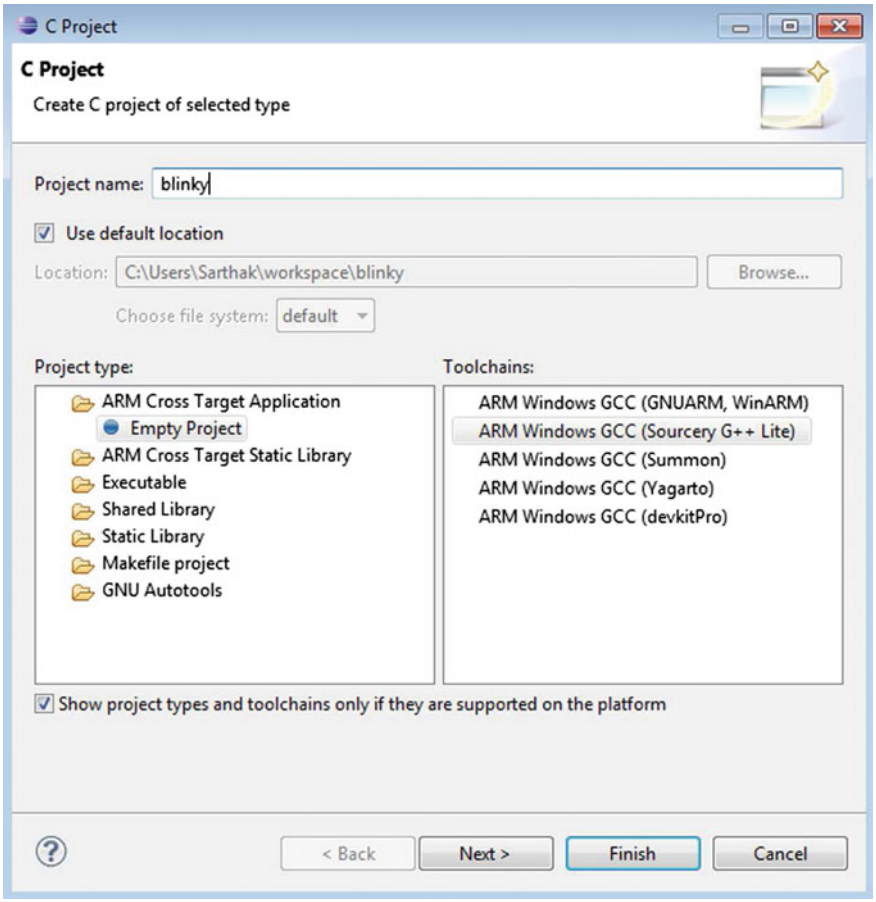

Fig. 6 New project window

c. Click Next.
d. Make sure the selection is same as shown in Fig. 7.
e. Then, press Finish.

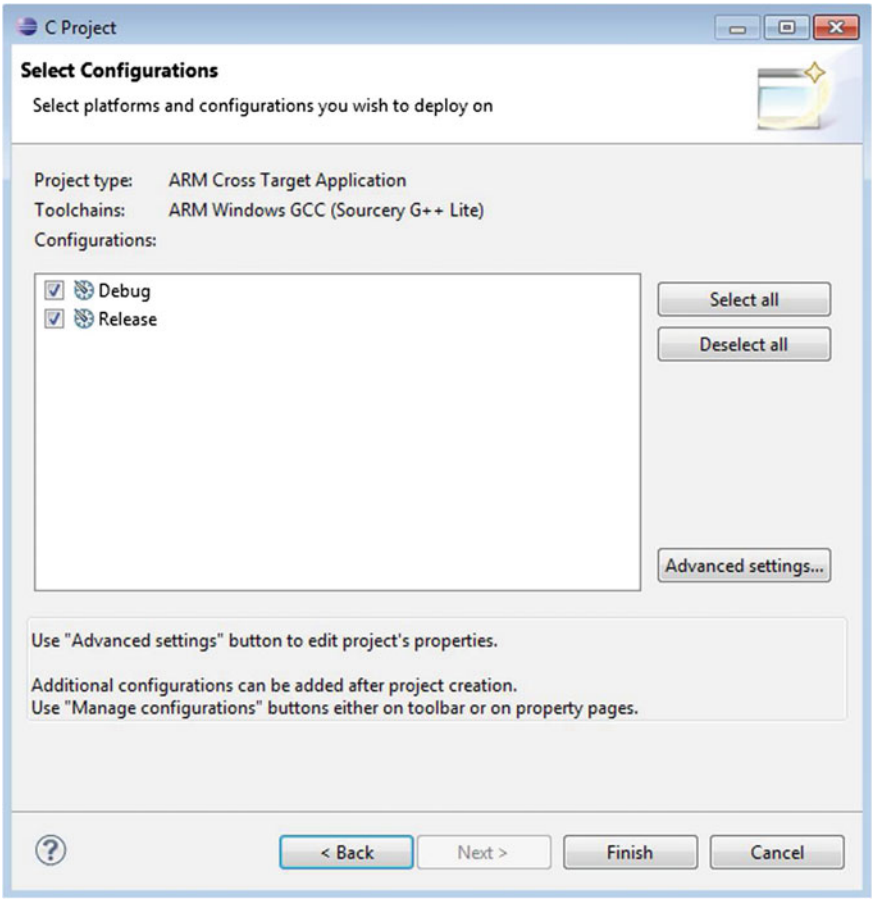

Fig. 7 Configuration window

3. Importing the existing file system.

   a. Go to File - Import - General - File System.
   b. A window similar to that shown in Fig. 8 will pop up.

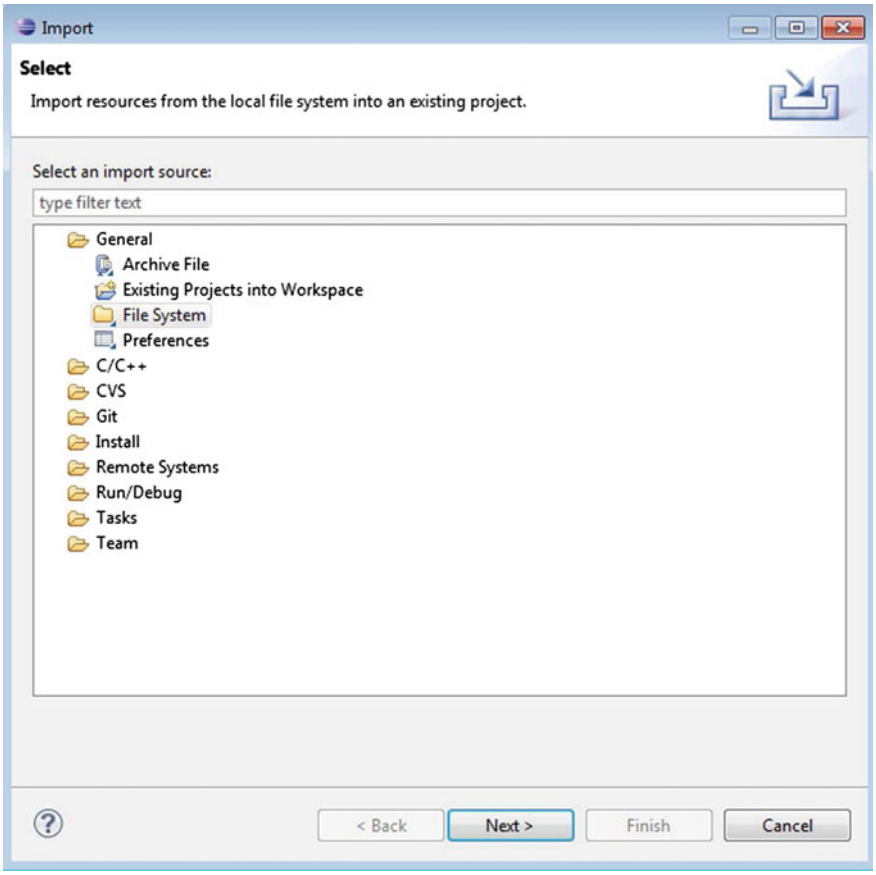

Fig. 8 Import window

   c. Press Next.
   d. Browse and select any project from Tiva Labs folder. Make sure that the same options as shown in Fig. 9 are selected.
   e. Press Finish.

So, till now ARM plug-in with Eclipse IDE is installed and have created a project which is using cross compiler for development. Screen should look similar to Fig. 10.

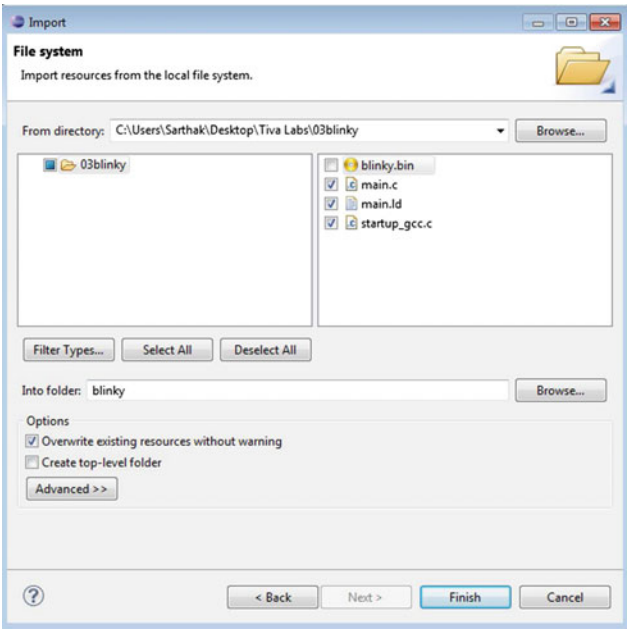

Fig. 9 File system window

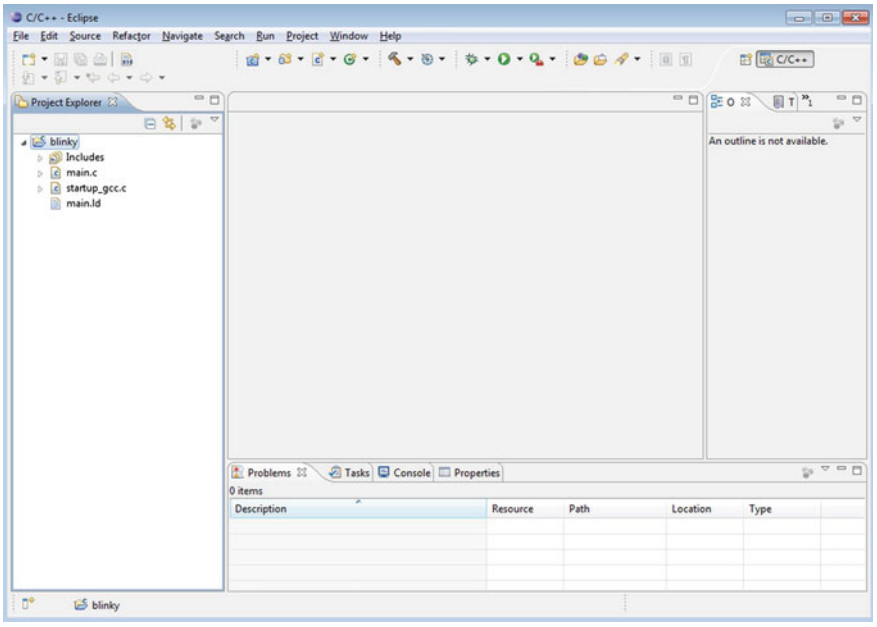

Fig. 10 Eclipse IDE window

4. Setting up project properties.

    a. Go to Project Properties.
    b. Go to C/C++ Build - Settings - Tool Settings and make sure processor selected is Cortex-M4 as shown in Fig. 11.

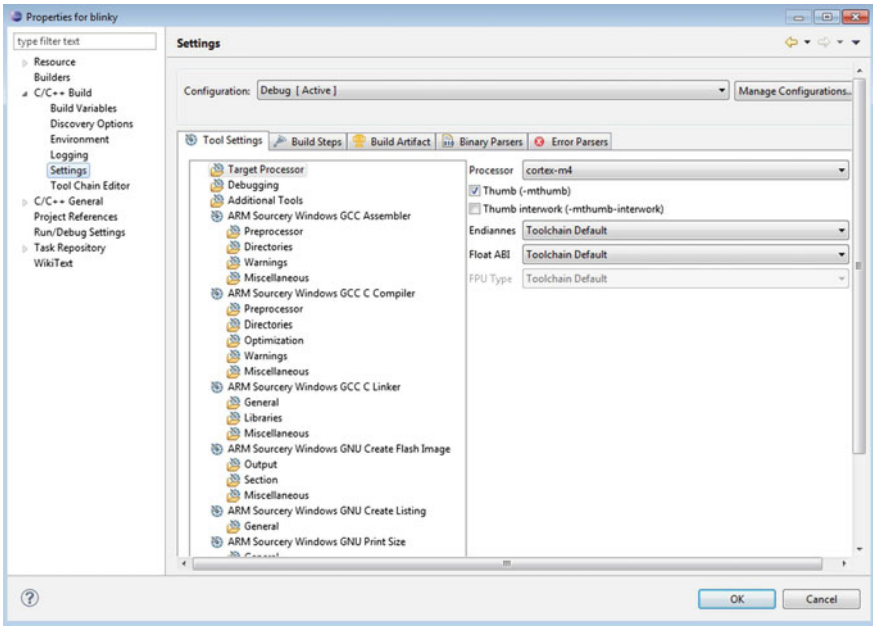

Fig. 11 Project properties

c. In the Project Properties window, go to Tool Settings - ARM Sourcery Windows GCC C Compiler - Preprocessor and define the symbol “gcc” and “PART_TM4C123GH6PM” as shown in Fig. 12.

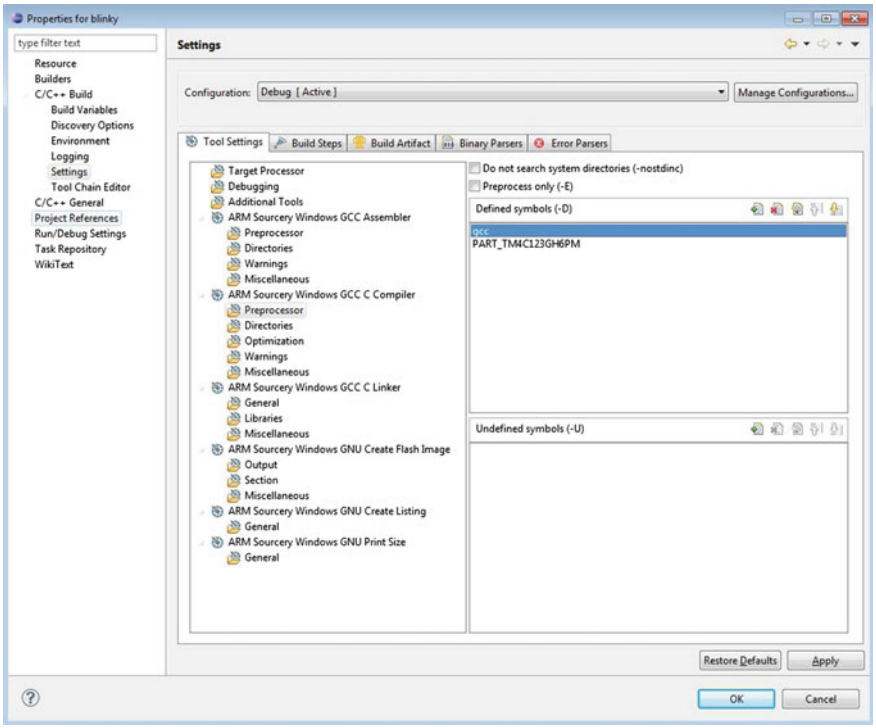

Fig. 12 Setting up preprocessor

d. Go to Tool Settings - ARM Sourcery Windows GCC C Compiler - Directories and include the following directory paths as shown in Figs. 13 and 14.

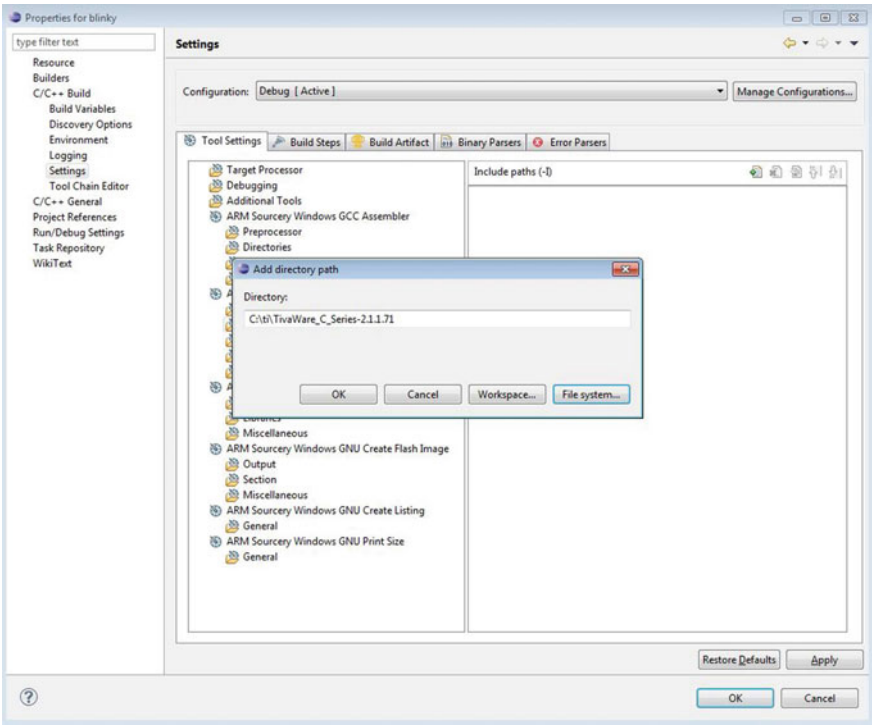

Fig. 13 Adding the directory path

i C:\ti\TivaWare_C_Series-2.1.1.71.
ii C:\ti\TivaWare_C_Series-2.1.1.71\driverlib.
iii C:\ti\TivaWare_C_Series-2.1.1.71\inc.
iv C:\ti\TivaWare_C_Series-2.1.1.71\utils.

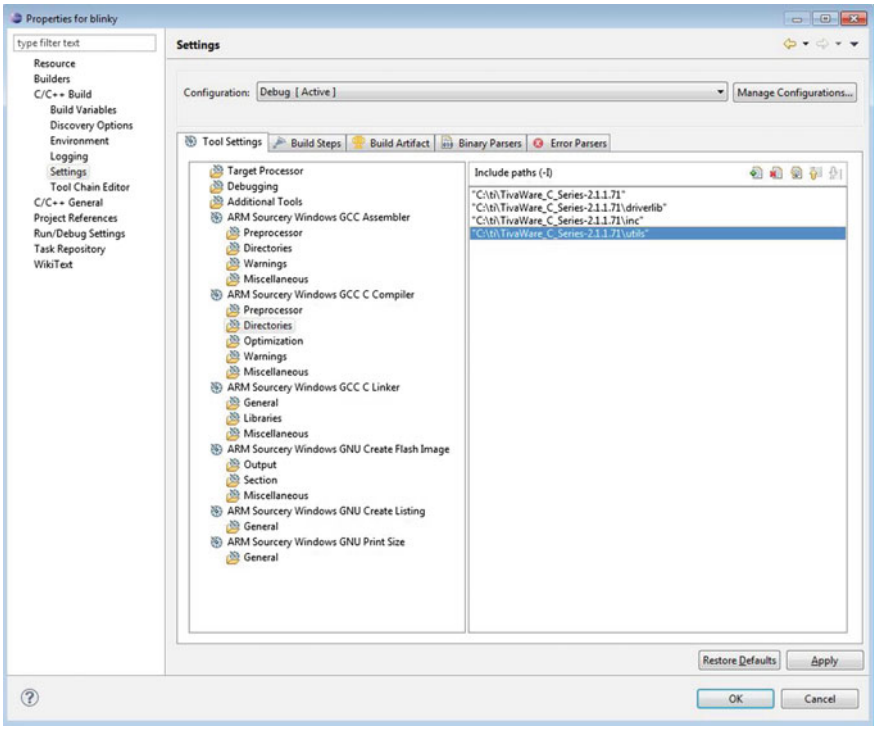

Fig. 14 Including TivaWare directories

e. Setting up the code optimization. Go to ARM Sourcery Windows GCC C Compiler - Optimization. Drop down the Optimization level options and select “None(-O0)” and check Function sections as shown in Fig. 15.

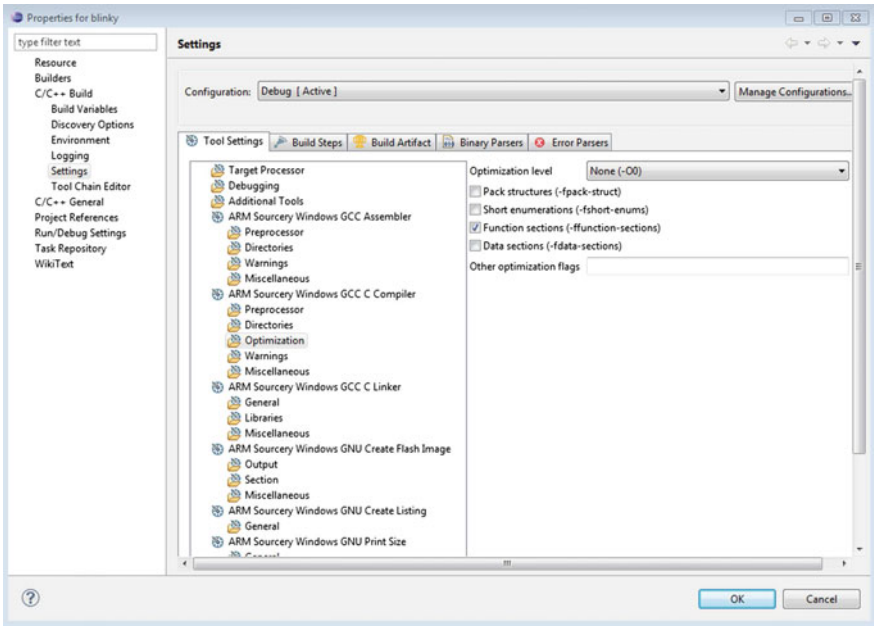

Fig. 15 Setting up optimization

f. Go to Tool Settings - ARM Sourcery Windows GCC C Linker - General and browse to linker file of corresponding project from workspace.
g. Check the options as shown in Fig. 16.

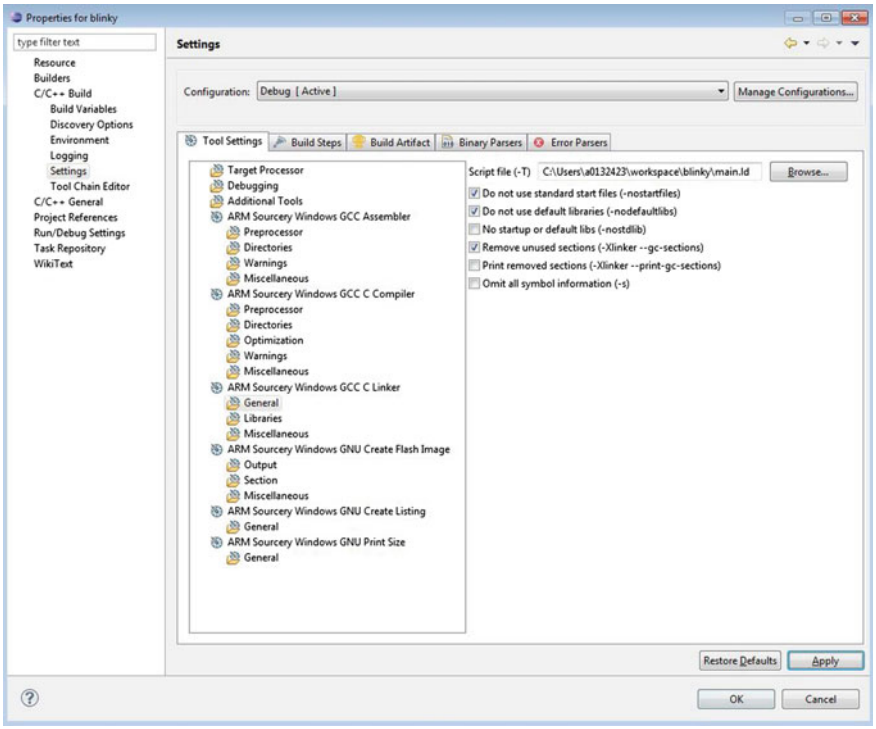

Fig. 16 Adding Linker file

h. Go to Tool Settings - ARM Sourcery Windows GCC C Linker - Libraries.
i Add the following mentioned paths under the Library search path as shown in Fig. 17.
j Click on “Apply” button on the extreme bottom right side of the window to the save the changes.
    i C:\ti\TivaWare_C_Series-2.1.1.71.
    ii C:\ti\TivaWare_C_Series-2.1.1.71 driverlib\gcc.

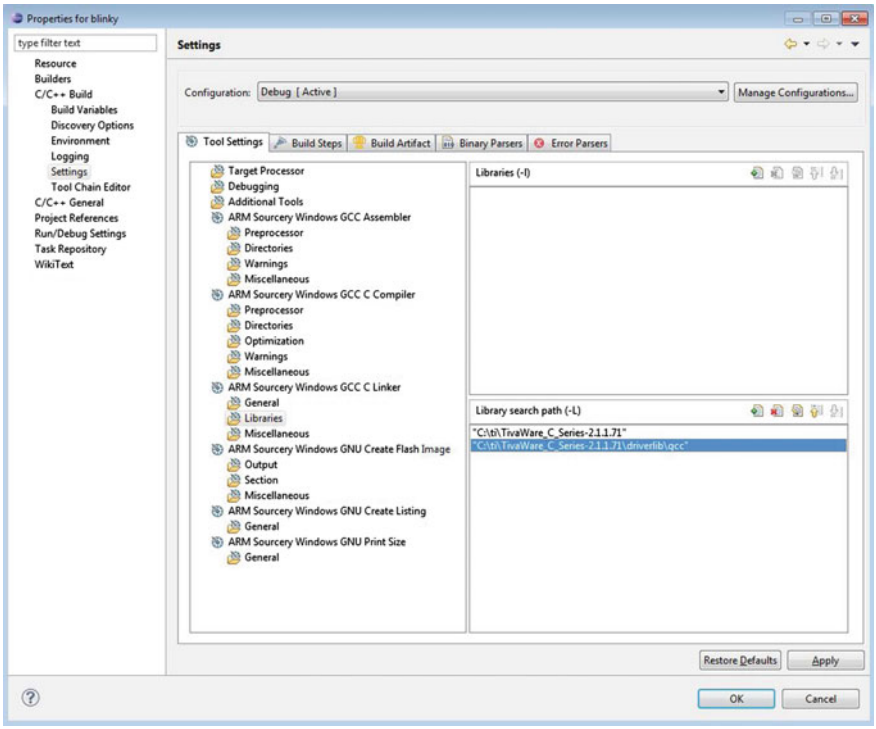

Fig. 17 Adding Linker libraries

k. In the Project Properties Window, drop down the C/C++ General and go to Path and Symbols - Source Location - Link Folder. Link folder “driverlib” and “utils” as shown in Figs. 18, 19 and 20. Click on “Apply” to save the changes.

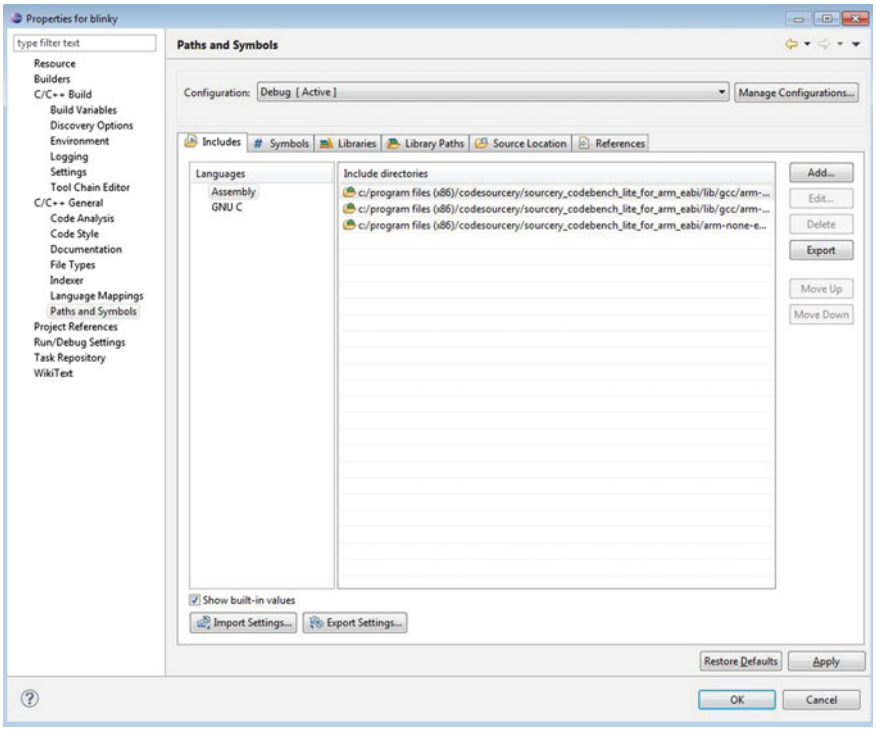

Fig. 18 Path and symbols

l. Click on “Apply” to save the changes.
m. Post-Build Steps. Go to C/C++ Build - Settings - Build Steps - Post-build steps - Command and type the following
arm-none-eabi-objcopy -S -O binary “$ProjName.elf$” “ProjName.bin”
n. Under description, type “Buld Bin File”. This is the description of the Post-Build Command mentioned above.
o. Figure 21 shows the Post-Build Steps window. Click on “Apply” to save the changes and then “OK” to close the project properties window.

5. After returning to main Eclipse IDE window, select main.c.
6. The screen will look like as shown in Fig. 22.

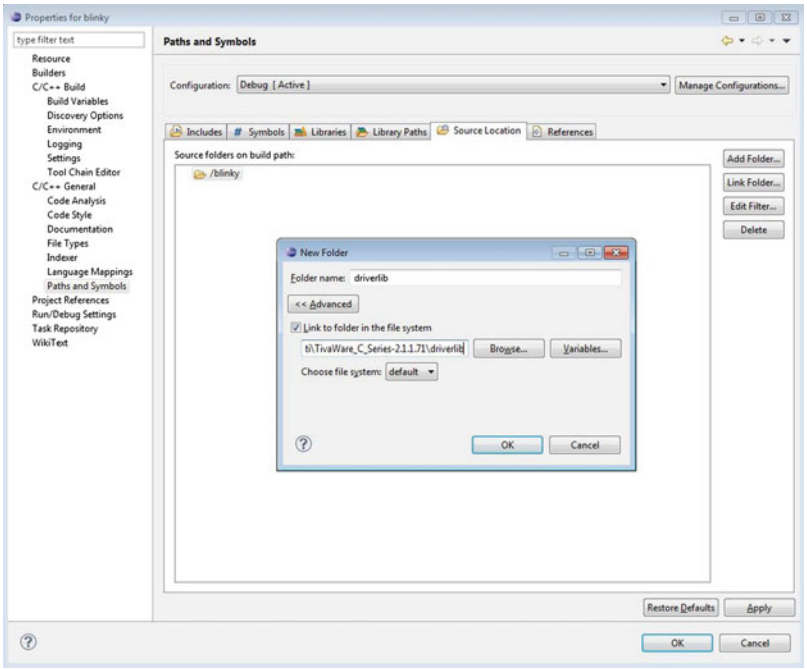

Fig. 19 Linking folder-(a)

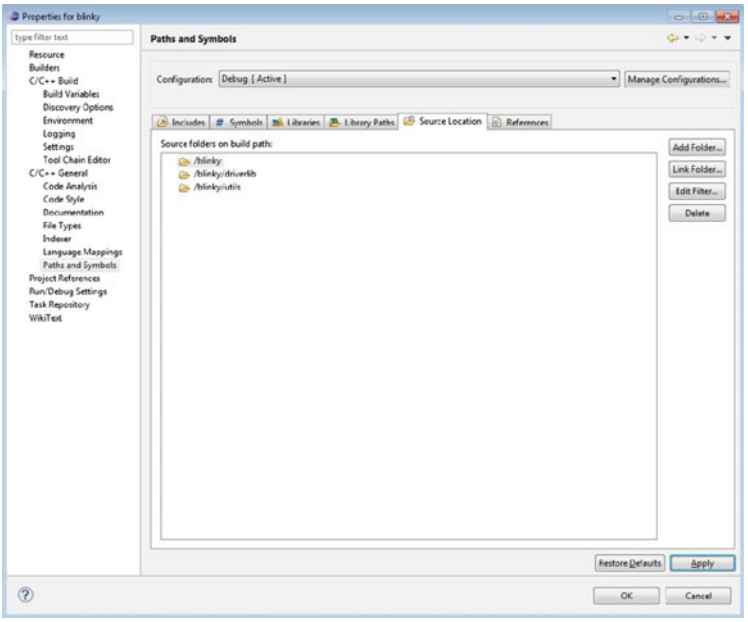

Fig. 20 Linking folder-(b)

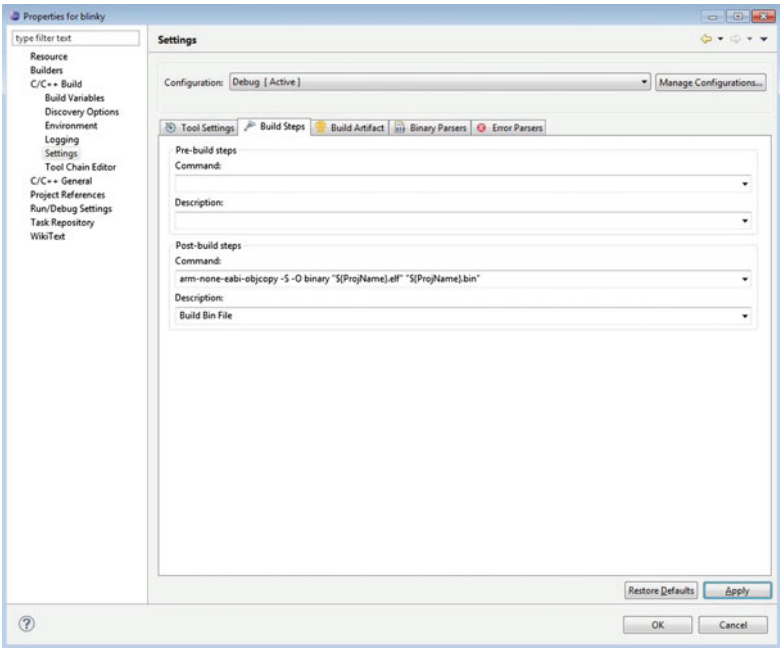

Fig. 21 Post-build steps

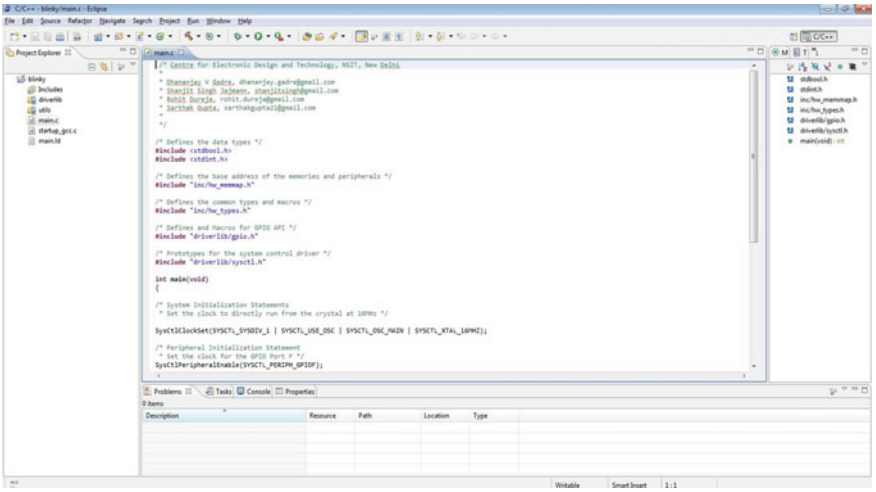

Fig. 22 Main eclipse IDE window

7. Before compiling the project, go the utils directory of the TivaWare (C:\ti\TivaWare_C_Series-2.1.1.71\utils). There are some libraries which refer to third-party applications, which are not required now so delete them. For the

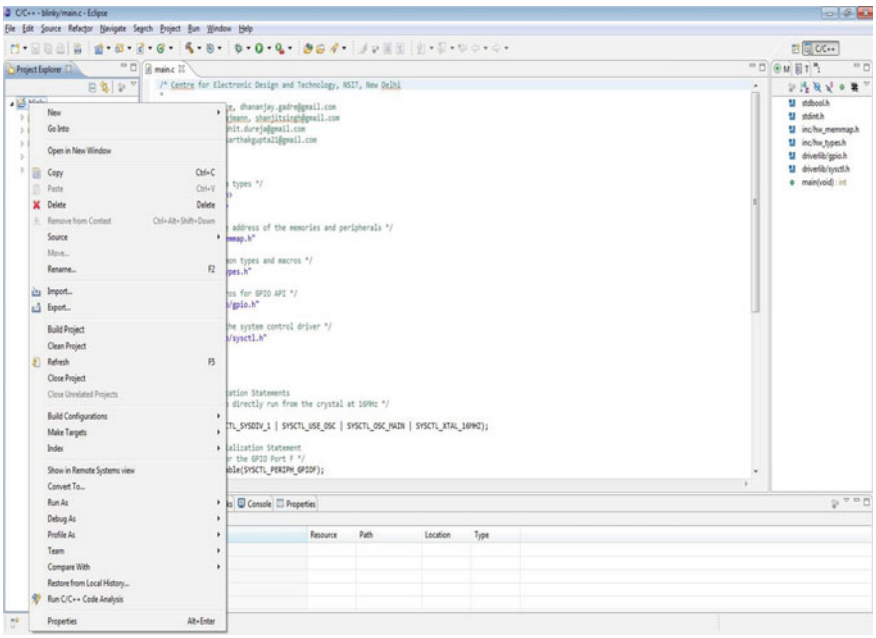

Fig. 23 Clean and build project

convenience of the learner those libraries which need to be deleted are fswrapper.c, fswrapper.h, locator.c, locator.h, lwiplib.c, lwiplib.h, ptpdlib.c, ptpdlib.h, ustdlib.c, ustdlib.h, random.c, random.h, speexlib.c, speexlib.h, swupdate.c, swupdate.h, tftp.c, and tftp.h

8. Now right click on the project, select Clean Project[4] then Build Project as shown in Fig. 23. If everything goes it will build with ease.
9. If the project has been built successfully, drop down the project "blinky", there will a directory created namely "Debug", under this "blinky.bin" file would have been created. This is file which will be programmed into the Tiva microcontroller or in the Tiva LaunchPad.
10. Now, open LM Flash Programmer Utility tool. Figure 24 shows the LM Flash Programmer window, select TM4C123G LaunchPad in the Configuration tab.
11. Click on Program tab, and select the .bin file by browsing to the project in Eclipse workspace folder.
12. Select the below options as shown in Fig. 25.
13. Click on Program which will transfer the .bin to the hardware (Tiva LaunchPad) and red LED will start blinking on it.

[4]By cleaning the project it will remove the intermediate files formed during compilation like object files, elf file,etc.

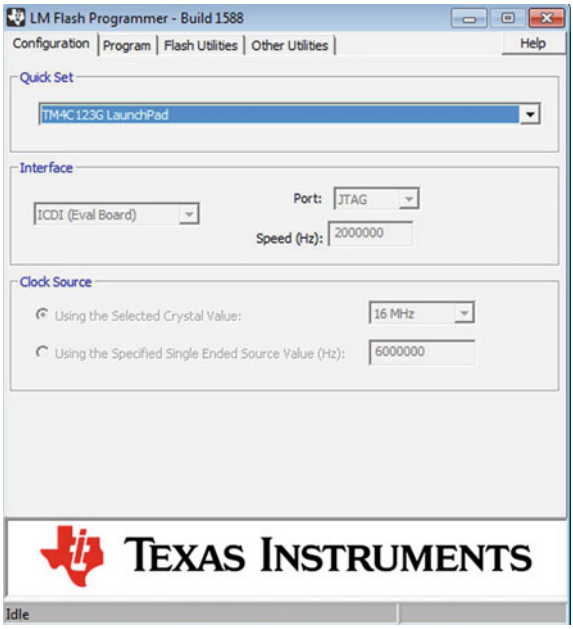

Fig. 24 LM flash programmer (configuration tab)

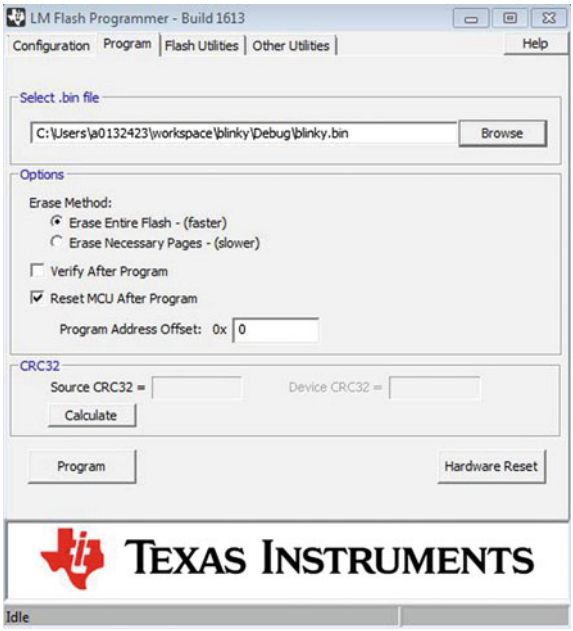

Fig. 25 LM flash programmer (program tab)

Chapter 7
Structure of Embedded C Program

This chapter deals with the basic structure of embedded C programming. Though, the program structure is explained with the reference of programming Tiva microcontrollers, same program structure is applicable with other microcontrollers. Knowledge of program structure helps in understanding the available example programs as all of them are written using the same structure. From the developer point of view it helps in writing applications or programs which can be easily shared and debug easily.

1 Anatomy of Embedded C Program

Embedded C Program can be divided into many sections as discussed below.

1. **Header Files**: When writing the Embedded C Program, including the header files is the first step. Header files will consist of the address locations of registers, definitions regarding peripherals functionality; it may contain definitions and macros for the application-specific features just like SD Card will have libraries-related fat file system.
2. **Global Variables**: These are the variables which are accessible by all the functions in the program. These variables are stored in volatile memory, example RAM.[1]
3. **Function Definition**: After declaring the global variables the definition of functions which is used in the program main loop is defined.
4. **Main Function**: It is the first function of the program where it starts to run.

[1] Global variables by default are stored in the SRAM location. But generally, in microcontrollers, the size of flash memory is more than the size of SRAM, so if there is a large array of coefficients like FIR, filter coefficients are required and they can be stored in flash memory. But these coefficients cannot be updated during run time.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_7

5. **Local Variables**: These variables have the scope limited to the main function only. The local variables are stored in stack,[2] which is a predefined part of RAM.
6. **System Initialization**: Now, the hardware system need to be initialized. This part mainly deals with clock selection; to use internal oscillator, or directly use external oscillator or use PLL (Phase Locked Loop) to select the clock from range of possible clock frequencies. This is lifeline of the program code without which none of the peripherals will work.
7. **Peripheral Initialization**: It includes enabling each peripherals required by the application. For example, to toggle a LED, enable the corresponding GPIO port to which LED is connected.
8. **Peripheral Configuration**: After the peripheral is enabled, now it need to be configured. For example, set the direction of GPIO as input or output, or select an alternate functionality of the GPIO.
9. **Application Loop**: Once the clock is enabled and peripherals are initialized and configured properly, then the application loop is defined which will be the body of the program and will be executed repeatedly.

The sample code for the embedded C program is described below as per the different sections discussed above. It gives the pseudolook of the embedded C program code.

```
/* Including Header Files */
#include "inc/tm4c123gh6pm.h"

/* Global Variables */
int g_var1, gvar2;

void func_1(char x)
{
/* Function Definition */
}

/* Main Function */
int main(void)
{
/* Local variables */
int l_var1,l_var2;

/* System Initialization Section
 * Example: Set the clock to directly run from the crystal at 16MHz */

/* Peripheral Initialization Section
 * Example: Set the clock for the GPIO Port F */
/* Peripheral Configuration Section
 * Example: Set the type (Input/Output) of the GPIO Pin */

while(1)
    {
```

[2]If there is a large array which is needed to be stored in RAM, then that array must be declared globally because by default, the size allocated for stack is less than the size allocated for global variables in RAM for Tiva series.

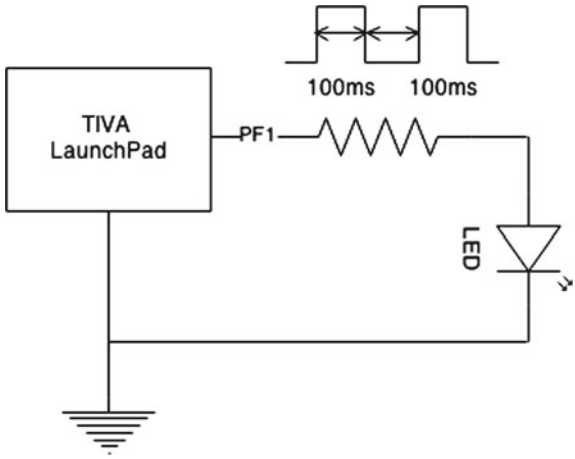

Fig. 1 Block diagram to toggle LED

```
    /* Application Loop */
    }
}
```

The program codes which are mentioned in the subsequent sections and the chapters are also compatible with the Code Composer Studio (CCS) because both the environments (CCS and custom environment as described in previous chapter) refer to same set of peripheral driver libraries and registers.

2 Experiment 1—Blinky

2.1 Objective

Blink a LED on the Tiva LaunchPad with a delay of 100 ms.

2.2 Hardware Description

For this experiment, one of the onboard RGB LED on the Tiva LaunchPad is used. Since, GPIO Pin 1 of Port F, i.e., PF1 controls the red component of RGB LED. So, in order to achieve the required blinking rate, the GPIO pin PF1 will have to be toggled between the logic high and logic low states with the required delay of 100 ms. The basic block diagram for the experiment setup is shown in Fig. 1.

As this chapter aims at structure of the Embedded C Program, the schematic level hardware description is discussed in Chap. 9 under Sect. 1.

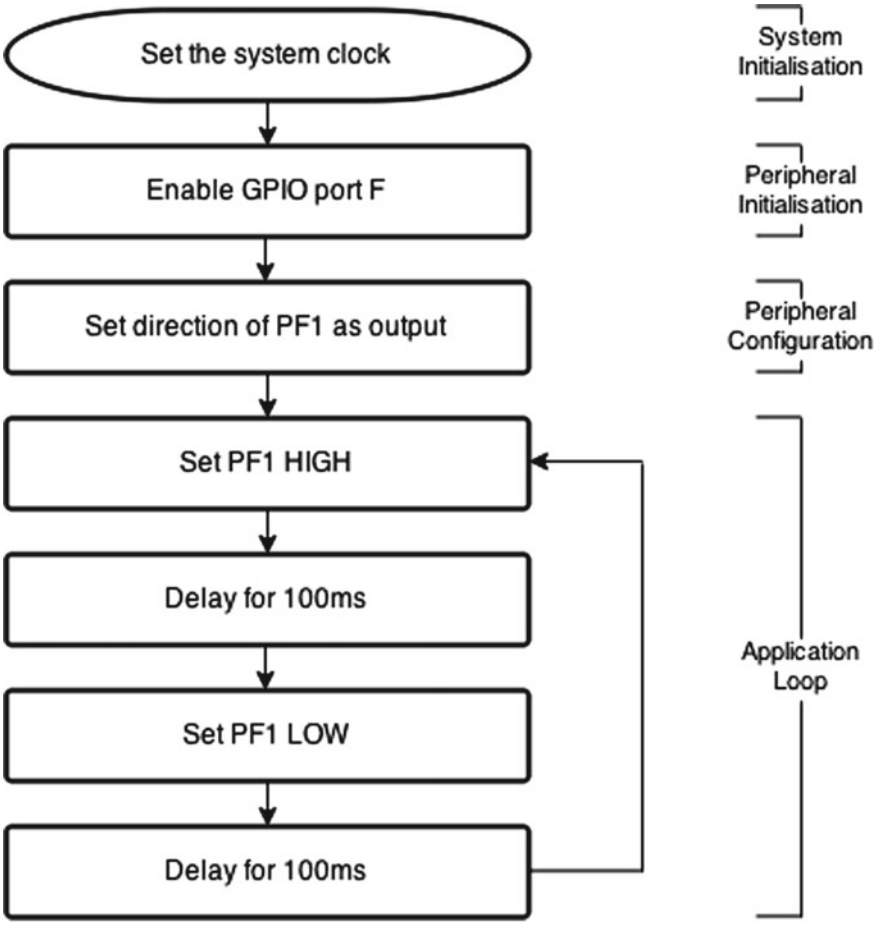

Fig. 2 Program flow to toggle LED

2.3 Program Flow

The algorithm to blink a LED is explained below.

1. Enable the system clock and the GPIO Port.
2. Configure the GPIO pin (from which LED is controlled) by setting its direction as output.
3. Set the state of the GPIO pin as high and low with appropriate delays in between to generate the required blinking effect.

Figure 2 shows the flowchart of program flow for the experiment.

2.4 Register Accesses

To use the register access model, include the header file "tm4c123gh6pm.h" which contains all the macros and definitions of each peripheral register corresponding to the microcontroller TM4C123GH6PM.

- **GPIO_PORTF_DIR_R**: This GPIO direction (GPIODIR) register is used to set the direction of the pin of the corresponding port as input or output. If the bit of the direction register is cleared then the corresponding pin of the port will be configured as input and if the bit is set then the pin will be configured as output.
- **GPIO_PORTF_DEN_R**: This is the GPIO digital enabled (GPIODEN) register. By default most of the pin are at tri-state after reset. They do not drive any logic value or allow the GPIO to read pin voltage. To use the pin as digital input or output the corresponding bit of GPIODEN register must be set.
- **GPIO_PORTF_AFSEL_R**: This is the GPIO alternate function select (GPIOAFSEL) register. If the bit is clear the corresponding pin is used as the GPIO but if the bit is set pin will be used by the associated peripheral.
- **GPIO_PORTF_DATA_R**: This is the GPIO data (GPIODATA) register. If the pin is configured as output whatever will the state of bit (logic "1" or "0") will be reflected on the pin. And, if the pin is configured as input then the state of pin (logic "1" or "0") will be stored as bit in the GPIODATA register.

2.5 Program Code

The complete C program of the experiment is given below. The program is divided into well commented segments.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

#include "inc/tm4c123gh6pm.h"

#define delay_value 135000

int main(void)
{
/* Delay Loop variable */
volatile unsigned long ulLoop;

/* Enable the GPIO port that is used for the onboard LED */
SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOF;

/* Do a dummy read to insert a few cycles after peripheral enable */
ulLoop = SYSCTL_RCGC2_R;

/* Enable the GPIO pin for the LED (PF1)
```

```
 * Set the direction as output, and enable the GPIO pin for
 * digital function. Care is taken to not disrupt the
 * operation of the JTAG pins on PC0-PC3 */
GPIO_PORTF_DIR_R |= 0x02;
GPIO_PORTF_DEN_R |= 0x02;
GPIO_PORTF_AFSEL_R = 0x00;

/* Loop forever */
while(1)
  {
  /* Turn on the LED */
  GPIO_PORTF_DATA_R |= 0x02;

  /* Delay for a 100ms */
  for(ulLoop = 0; ulLoop < delay_value; ulLoop++)
    {
    }

  /* Turn off the LED */
  GPIO_PORTF_DATA_R &= ~(0x02);

  /* Delay for a 100ms */
  for(ulLoop = 0; ulLoop < delay_value; ulLoop++)
    {
    }
  }
}
```

3 Experiment 2—Switchy

3.1 Objective

Mimic the state of a switch using a LED. The LED should light up whenever the switch is pressed.

3.2 Hardware Description

This experiment requires one tactile switch and one LED, both of which are present on the board of TIVA LaunchPad. The LED connected to PF1 will be used along with the onboard tactile switch connected to PF4 as shown in block diagram Fig. 3. For proper operation of the switch, its connection to PF4 must be pulled up to a logic high level. The onboard hardware does not include a pull-up connection. This means that the internal pull-up available on the Tiva microcontroller will have to be used on PF4. It may be noted that in this configuration, when switch is not pressed, logic high is read and when the switch is pressed logic low is read at GPIO pin PF4.

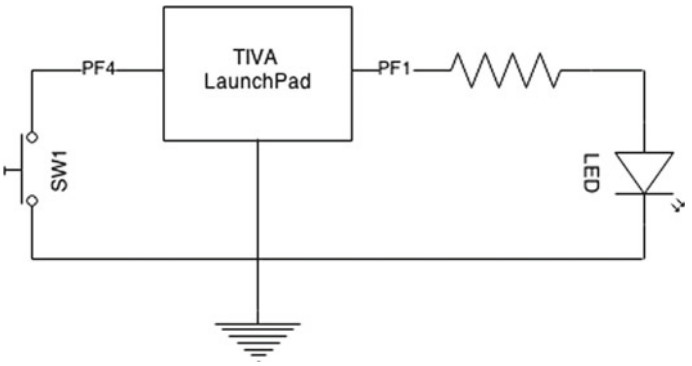

Fig. 3 Block diagram for LED and a switch

3.3 Program Flow

The algorithm to read a switch and reflect the state of switch through is enumerated below.

1. Enable the system clock and GPIO port F.
2. Set the direction of pin PF1 (pin which is controlling the LED) as output and pin PF4 (which is reading the state of switch) as input.
3. Enable the internal pull-ups on input pin, PF4.
4. Check the state of pin PF4.
5. If PF4 is at a logic low state, the switch has been pressed and the LED is turned on.
6. If PF4 is at a logic high state, it is at its default state due to the internal pull-up, implying that it is not pressed and hence the LED is turned off.

Figure 4 shows the flowchart of program flow for the experiment.

3.4 Register Accesses

It uses almost the same set of registers as the previous experiment. The difference is that earlier LED was blinking constantly but now state of LED is controlled by switch. So, two GPIO pins are used, one GPIO as input for switch and other one GPIO as output for LED.

- **GPIO_PORTF_PUR_R**: This is GPIO pull-up select (GPIOPUR) register. This register enables a pull-up resistor configuration at the GPIO pin corresponding to the GPIO port (in this experiment GPIO pin 4 of PORT F). When the switch is pressed logic low appears on the GPIO pin (configured as input) as other end of the switch is connected to ground (or logic low). And, when the switch is released,

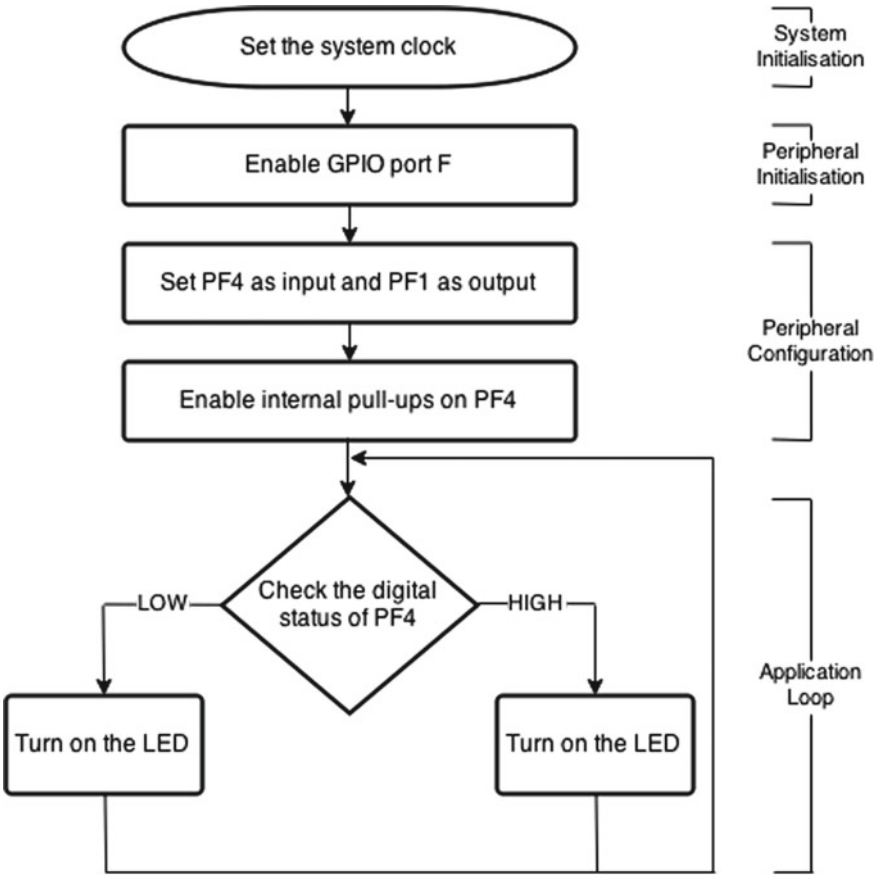

Fig. 4 Program flow for LED and a switch

logic high will be there on the GPIO pin (configured as input). This logic high is appearing because at that GPIO, internal pull-ups are enabled by using GPIO pull-up select register. Hence, there is no need for external pull-up resisters at the input pin.

The rest of registers which are used have been already discussed in the experiment 1.

3.5 Program Code

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>
```

```
#include "inc/tm4c123gh6pm.h"

int main(void)
{
/* Dummy variable */
volatile unsigned long ulLoop;

/* Enable the GPIO port that is used for the onboard LED */
SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOF;

/* Do a dummy read to insert a few cycles after peripheral enable */
ulLoop = SYSCTL_RCGC2_R;

/* Enable the GPIO pin for the LED (PF1)
 * Set the direction as output, and enable the GPIO pin for
 * digital function. Care is taken to not disrupt the
 * operation of the JTAG pins on PC0-PC3 */
GPIO_PORTF_DIR_R |= 0x02;
GPIO_PORTF_DEN_R |= 0x02;

/* Enable the GPIO pin for the Switch1(PF4)
 * Set the direction as input, and enable the GPIO pin for
 * digital function. Care is taken to not disrupt the
 * operation of the JTAG pins on PC0-PC3 */
GPIO_PORTF_DIR_R &= ~(0x10);
GPIO_PORTF_DEN_R |= 0x10;
GPIO_PORTF_AFSEL_R = 0x00;

/* Pulling up PF4 Pin */
GPIO_PORTF_PUR_R |= 0x10;

/* Loop forever */
while(1)
  {
  /* If the button is pressed, turn on LED */
  if(((GPIO_PORTF_DATA_R) & (0x10)) != (0X10))
    GPIO_PORTF_DATA_R |= 0x02;
  /* If button is not pressed, turn off LED */
  else
    GPIO_PORTF_DATA_R &= ~(0x02);
  }
}
```

Chapter 8
Application Programming Interface (API)

This chapter deals with the application programming interface (API) model of programming. This programming interface empowers the user for rapid development of applications for Tiva microcontrollers.

1 Peripheral Driver Library

Peripheral driver layer provides higher abstraction layer for the microcontroller registers. This powers the user to write application without going into depth for peripheral registers of the microcontroller. This helps in the rapid development of applications. In this, users are only exposed to the functions which require the configurable parameters as input. The actual configuration part at register level is covered in the peripheral driver libraries. Figure 1 shows the different layers of abstraction up till user application.

- **MCU Abstraction Layer**: This is the lower layer which deals with 32-bit microcontroller registers.
- **Peripheral Driver Abstraction Layer**: This abstraction layer is above the MCU abstraction layer. As, this layer comprises of functions which take configurable parameters as input. Inside the functions it uses the MCU abstraction layer. So, the layer above it will just need to call the functions of this layer. This would be helpful in developing complex applications involving several peripherals as they not require much depth of knowledge of the peripherals registers.
- **Middle Abstraction Layer**: This layer is the intermediate layer between the user application and peripheral driver abstraction layer. It may or may not be present. It contains the third-party libraries which are generally specific to certain features, like library for I2C-based humidity sensor. It will be using the peripheral driver abstraction layer inside and generating the feature-specific data.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_8

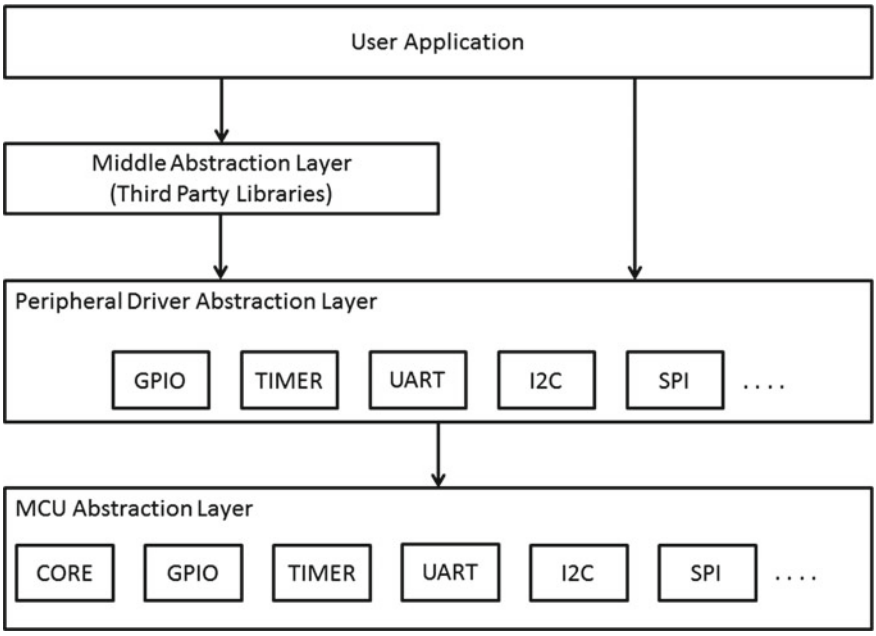

Fig. 1 Layers of abstraction

- **User Application**: It is program code specific to the application written by the developer. In this, peripheral abstraction layer can be accessed either directly or through the middle abstraction layer.

2 Programming Models

There are different programming models based on the abstraction layer. Each programming model has its own pros and cons. These programming models are elaborated in the subsequent sections.

2.1 Direct Register Access Model

In this model, user directly writes into the peripheral registers. Experiments 1 and 2 use direct register access model of programming. This kind of model develops a smaller and more efficient code as compared to the software driver model. But to implement this model, user needs to have a very good understanding of peripherals.

As, it is needed to control every bit of the peripheral registers, for developing complex application, using this model could be cumbersome.

To use this model, header file is required which contains all the peripheral register macros mapped to the respective address locations. These header files are in the "inc" directory.

2.2 Software Driver Model

The software driver model uses the API provided by the peripheral driver abstraction layer. Though due to inclusion of more header files related to peripheral drivers and calling more functions affect the size and efficiency of the code in degrading manner. But this kind of wrapper helps in the rapid development of complex applications as compared to direct register access model. Also, in this model, developer is insulated from the low-level details of the operation of peripherals.

All the further experiments are explained using this model only. The peripheral drivers are defined and declared as libraries in directory "driverlib" and common operation like performing trigonometric functions, software implementation of i2c and spi, etc. are defined and declared as libraries in directory "utils".

2.3 Using Both Models

Since, both the programming models have their own advantages and disadvantages, mixture of both the models can be used to develop the complex application which will be more efficient than the pure software driver model, but obviously less than pure direct register access model. But it will much easier to develop the complex applications as compared to developing applications by using direct register access model. It is more like balancing the efficiency and ease of design of application by using the mixture of both direct register access model and software driver model.

3 Useful API Function Calls

It is recommended to use the API or software driver model to develop the application. Therefore, all the further experiments are discussed in reference to API calls. So, in the below enumeration some commonly used API function calls are discussed.

1. **SysCtlClockSet** - It sets clock frequency of the device.
   Prototype: void SysCtlClockSet(uint32_t ui32Config)
   Parameters: ***ui32Config*** is the required configuration parameter of device clock frequency.

Table 1 System clock frequencies for possible system clock dividers

SYSDIV	Divisor	Frequency	Parameter
0 × 0	/1	reserved	SYSCTL_SYSDIV_1
0 × 1	/1	reserved	SYSCTL_SYSDIV_2
0 × 2	/1	66.67 MHz	SYSCTL_SYSDIV_3
0 × 3	/1	50 MHz	SYSCTL_SYSDIV_4
0 × 4	/1	40 MHz	SYSCTL_SYSDIV_5
0 × 5	/1	33.33 MHz	SYSCTL_SYSDIV_6
0 × 6	/1	28.57 MHz	SYSCTL_SYSDIV_7
0 × 7	/1	25 MHz	SYSCTL_SYSDIV_8
0 × 8	/1	22.22 MHz	SYSCTL_SYSDIV_9
0 × 9	/1	20 MHz	SYSCTL_SYSDIV_10
0 × A	/1	18.18 MHz	SYSCTL_SYSDIV_11
0 × B	/1	16.67 MHz	SYSCTL_SYSDIV_12
0 × C	/1	15.38 MHz	SYSCTL_SYSDIV_13
0 × D	/1	14.29 MHz	SYSCTL_SYSDIV_14
0 × E	/1	13.33 MHz	SYSCTL_SYSDIV_15
0 × F	/1	12.5 MHz	SYSCTL_SYSDIV_16

Description: This function configures clock frequency of the device. The input crystal frequency, oscillator to be used, use of PLL, and system clock divider are all configured with this function. The ui32Config parameter is the logical OR of several different values, many of which are grouped into sets where only one can be chosen. The system clock divider is chosen with one of the following values: SYSCTL_SYSDIV_1, SYSCTL_SYSDIV_2, SYSCTL_SYSDIV_3, ... SYSCTL_SYSDIV_64.
Corresponding to each divider value, there is a system clock value generated by the PLL. These values are mentioned in Table 1.
The use of the PLL is chosen with either SYSCTL_USE_PLL or SYSCTL_USE_OSC. The external crystal frequency is chosen with one of the following values:
SYSCTL_XTAL_1MHZ , SYSCTL_XTAL_1_84MHZ,
SYSCTL_XTAL_2MHZ , SYSCTL_XTAL_2_45MHZ,
SYSCTL_XTAL_3_57MHZ , SYSCTL_XTAL_3_68MHZ,
SYSCTL_XTAL_4MHZ , SYSCTL_XTAL_4_09MHZ,
SYSCTL_XTAL_4_91MHZ , SYSCTL_XTAL_5MHZ,
SYSCTL_XTAL_5_12MHZ , SYSCTL_XTAL_6MHZ,
SYSCTL_XTAL_6_14MHZ , SYSCTL_XTAL_7_37MHZ,
SYSCTL_XTAL_8MHZ , SYSCTL_XTAL_8_19MHZ,
SYSCTL_XTAL_10MHZ , SYSCTL_XTAL_12MHZ,
SYSCTL_XTAL_12_2MHZ , SYSCTL_XTAL_13_5MHZ,
SYSCTL_XTAL_14_3MHZ , SYSCTL_XTAL_16MHZ,
SYSCTL_XTAL_16_3MHZ , SYSCTL_XTAL_18MHZ,

SYSCTL_XTAL_20MHZ , SYSCTL_XTAL_24MHZ,
or SYSCTL_XTAL_25MHz.
There are values below which PLL operation does not work. These values vary with the different classes of devices. Example values below SYSCTL_XTAL_5MHZ are not valid when the PLL is in operation for Blizzard-class devices.

The oscillator source is chosen with one of the following values:

SYSCTL_OSC_MAIN , SYSCTL_OSC_INT,
SYSCTL_OSC_INT4 , SYSCTL_OSC_INT30,
or SYSCTL_OSC_EXT32.
To clock the system from an external source (such as an external crystal oscillator), use SYSCTL_USE_OSC | SYSCTL_OSC_MAIN. To clock the system from the main oscillator, use SYSCTL_USE_OSC | SYSCTL_OSC_MAIN. To clock the system from the PLL, use SYSCTL_USE_PLL | SYSCTL_OSC_MAIN, and select the appropriate crystal with one of the SYSCTL_XTAL_xxx values.
Returns: None.

2. **SysCtlClockGet** - this function returns the processor clock rate.
Prototype: uint32_t SysCtlClockGet(void)
Parameters: No parameters need to be passed.
Description: This function determines the clock rate of the processor clock, which is also the clock rate of the peripheral modules (with the exception of PWM, which has its own clock divider; other peripherals may have different clock frequency, see the device data sheet for details).[1]
Returns: The processor clock rate.
3. **SysCtlPeripheralEnable** - It enables the peripheral.
Prototype: void SysCtlPeripheralEnable(uint32_t ui32Peripheral)
Parameters: ***ui32Peripheral*** is the peripheral to enable.
Description: This function enables the peripheral. At power-up, all the peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes. The **ui32Peripheral** parameter is like below-mentioned values:
SYSCTL_PERIPH_ADC0 , SYSCTL_PERIPH_ADC1,
SYSCTL_PERIPH_COMP0 , SYSCTL_PERIPH_EEPROM0,
SYSCTL_PERIPH_I2C1 , SYSCTL_PERIPH_I2C2,
SYSCTL_PERIPH_PWM0 , SYSCTL_PERIPH_PWM1,
SYSCTL_PERIPH_SSI0 , SYSCTL_PERIPH_SSI1, etc.
It is possible that all the features may not be present in one microcontroller, so consult the datasheet and peripheral driver guide before using it.
Returns: None.

[1]This cannot return accurate results if SysCtlClockSet() has not been called to configure clock frequency of the device, or if the device is directly clocked from a crystal (or a clock source) that is not one of the supported crystal frequencies. In the latter case, this function should be modified to directly return the correct system clock rate.

4. **SysCtlDelay** - It provides a small delay.
   Prototype: void SysCtlDelay(uint32_t ui32Count)
   Parameters: ***ui32Count*** is the number of delay loop iterations to perform.
   Description: This function generates a constant length delay (maximum delay 2^{32}, as length of unsigned number is 32 bits). It is written in assembly to keep the delay consistent across tool chains, avoiding the need to tune the delay based on the tool chain used. The loop takes three cycles/loop.
   Returns: None.
5. **SysCtlReset** - It helps to reset the device.
   Prototype: void SysCtlReset(void)
   Parameters: No parameters need to be passed.
   Description: This function performs a software reset of the entire device. The processor and all peripherals are reset and all device registers are returned to their default values (with the exception of the reset cause register, which maintains its current value but has the software reset bit set as well).
   Returns: None.
6. **IntMasterEnable** - It enables the processor interrupt.
   Prototype: bool IntMasterEnable(void)
   Parameters: No parameters need to be passed.
   Description: This function allows the processor to respond to the interrupts. This function does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.
   Returns: Returns true if interrupts were disabled when the function was called or false if they were initially enabled.

Chapter 9
Digital Input/Output

This chapter will focus on using the TivaWare API functions instead of the register access model discussed in the previous chapter. To get acquainted with using the TivaWare API, some of the basic experiments that were performed in Chap. 7 will be again revisited. After getting comfortable with the TivaWare API, this chapter helps in exploring some of the features present on the PadmaBoard, such as using LED as light sensor, tone generation using a buzzer, and measure distance using ultrasonic module.

1 Experiment 3—API Blinky

1.1 Objective

Blink a LED on the Tiva LaunchPad with delay of 100 ms.

1.2 Hardware Description

For this experiment, one of the onboard LEDs on the Tiva LaunchPad is used. The LED will be controlled by one of the GPIO pins of the Tiva microcontroller. In order to achieve the required blinking rate, the GPIO pin controlling the LED will have to be toggled between logic high and logic low states with the required delay of 100 ms. Basic block diagram for the experiment setup is shown in Fig. 1.

The Tiva LaunchPad has an onboard RGB LED, controlled by GPIO pins PF1, PF2, and PF3. For this experiment, only one LED is required, the red LED is controlled by PF1. The schematic snippet of the Tiva LaunchPad (Fig. 2) shows the active high configuration of the onboard common anode RGB LED using NPN transistors. Such a configuration is necessary because the Tiva microcontroller is only capable

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_9

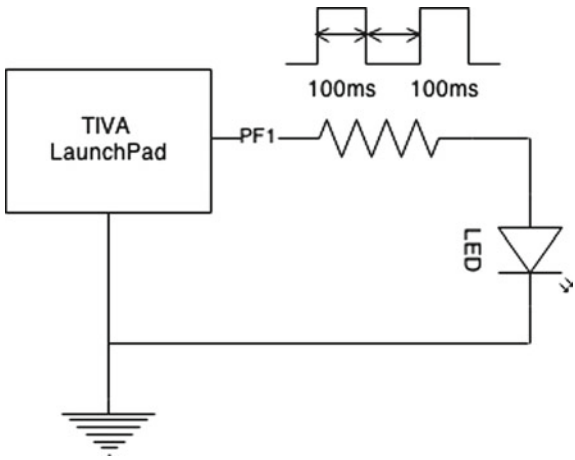

Fig. 1 Block diagram for blinky

of sinking up to 18 mA of current on a maximum on two pins located on one physical side of the device package.

This limitation of maximum current is overcome by the use of the NPN transistor configuration. In this configuration, logic high will turn on the LED and a logic low will turn it off.

1.3 Program Flow

The general algorithm to be followed for creating any blinking pattern is:

1. Enable the system clock and the GPIO port F.
2. Set the direction of GPIO pin, PF1. In this case, it will be an output pin.
3. Set the state of the GPIO pin, PF1 as high and low with appropriate delays in between to generate the required blinking pattern.

Figure 3 shows the flowchart of program flow for the experiment.

1.4 Useful API Function Calls

Unlike the register access method used previously, API function calls are used to configure and control the GPIO pins to perform the blinking function. The functions used to drive the GPIO port pins are contained in *driverlib/gpio.c*, with *driverlib/gpio.h* containing the API definitions for use by applications. This means that the header file *driverlib/gpio.h* must be included in the program code to use the necessary API functions. In this experiment, two basic API functions are used GPIOPinTypeGPIOOutput and GPIOPinWrite. As, from the name it can be guessed, these API

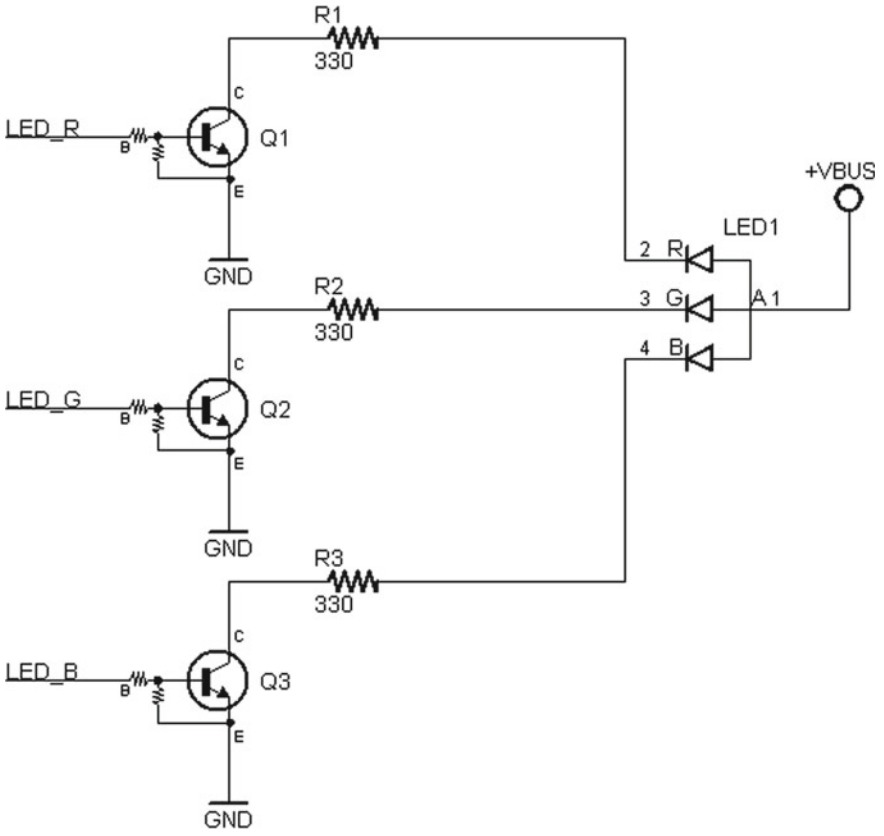

Fig. 2 Schematic of RGB LED on Tiva LaunchPad

functions are used to configure a GPIO pin as a digital output pin and to write logic high or low value to a particular GPIO pin. Let us briefly discuss the structure of these API function calls.

- **GPIOPinTypeGPIOOutput** - Configures the pin(s) for use as GPIO Outputs.
  Prototype: void GPIOPinTypeGPIOOutput(uint32_t ui32Port, uint8_t ui8Pins)
  Parameters: ***ui32Port*** is the base address of the GPIO Port. ***ui8Pins*** is the bit-packed representation of the pin(s).
  Description: This function sets the pins described by the parameter ui8Pins located on the GPIO port described in ui32Port as a general purpose digital output pin. This API function can be used to configure one or more pins located on a single GPIO port as an output pin. The pins to be set as output is specified in ui8Pins in a bit-packed fashion where the value of 1 on a particular bit indicates that that particular GPIO pin will be configured as a digital output pin.
  Returns: None.

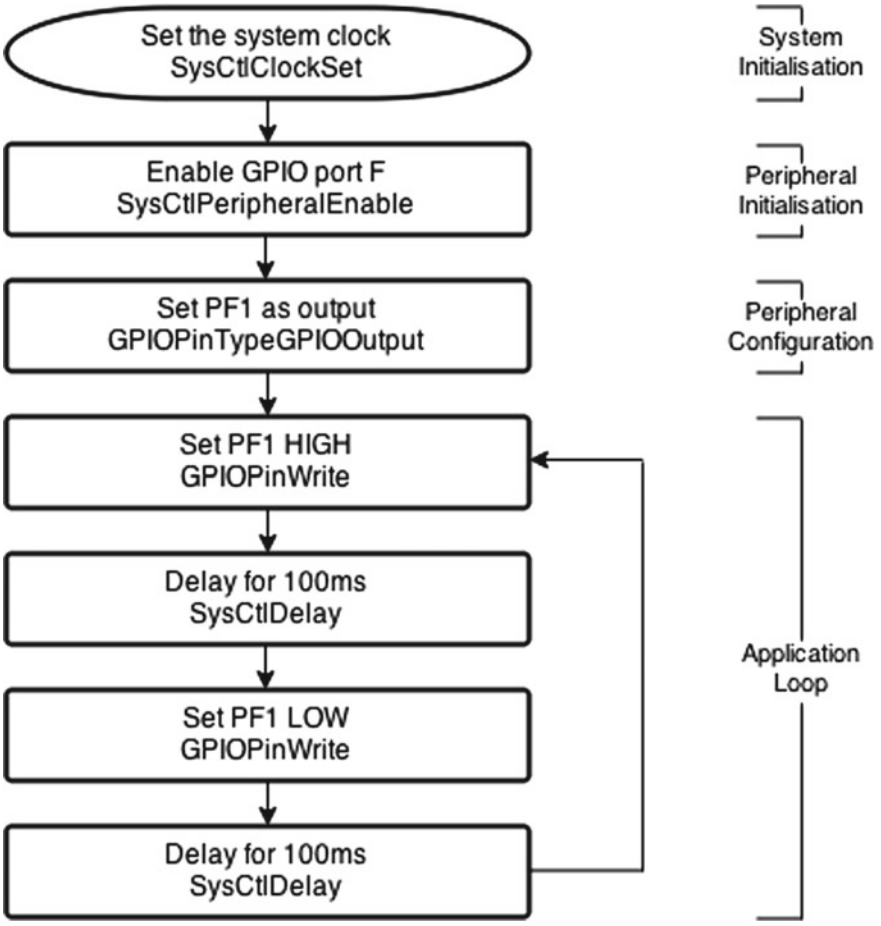

Fig. 3 Program flow for blinky

- **GPIOPinWrite** - Writes a value to the specified pin(s).
  Prototype: void GPIOPinWrite(uint32_t ui32Port, uint8_t ui8Pins, uint8_t ui8Val)
  Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s). ***ui8Val*** is the value to write to the pin(s).
  Description: This function writes the digital value (1 or 0) specified in ui8Val to the pins specified in ui8Pins located on the port described by ui32Port. The GPIOPinWrite function will be effective only on pins configured to be digital outputs. Using the function on an input pin will have no effect. Like the previous function, multiple pins on the same port can be written to using a single function call. The pin numbers and their corresponding values are given in a bit-packed form in the ui8Pins and ui8Val parameters.
  Returns: None.

1.5 Program Code

The complete C program code for the experiment is given below. The program has been broken up in to well-commented segments. This programming model may be used for all subsequent experiments.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

int main(void)
{

/* System Initialization Statements
 * Set the clock to directly run from the crystal at 16MHz */
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Peripheral Initialization Statement
 * Set the clock for the GPIO Port F */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

/* Peripheral Configuration Statement
 * Set the type of the GPIO Pin */
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1);

/* GPIO Pin 1 on PORT F initialized to 0, RED LED is off */
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1,0);

/* Application Loop */
while(1)
  {
  /* Make Pin High */
  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1,GPIO_PIN_1);

  /* Delay for 100ms */
  SysCtlDelay(SysCtlClockGet()/30);

  /* Make Pin Low */
  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1,0);

  /* Delay for 100ms */
  SysCtlDelay(SysCtlClockGet()/30);
```

```
    }
}
```

The GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1) function is used to configure pin 1 of Port F (i.e., PF1) as a digital output pin.

The GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1) function will set the output on PF1 to logic high and the GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1,0) function will set PF1 to logic low.

2 Experiment 4—API Switchy

2.1 Objective

Mimic the state of a switch on a LED. The LED should light up whenever the switch is pressed.

2.2 Hardware Description

This experiment will require one tactile switch and one LED, both of which are present on Tiva LaunchPad. The same LED connected to PF1 which is used in the previous experiment along with the onboard tactile switch connected to PF4 is used for this experiment. (Figure 4) One end of the tactile switch is connected to ground and the other end to the GPIO pin PF4. For proper operation of the switch, its connection to PF4 must be pulled up to a logic high level. The onboard hardware does not include a pull-up connection. This means that the internal pull-up available on the Tiva microcontroller will have to be used on PF4. It may be noted that in this configuration, when switch is not pressed, logic high is read at GPIO pin PF4 and when the switch is pressed logic low is read.

2.3 Program Flow

In this experiment, whenever switch connected to PF4 is pressed, LED connected to PF1 must be made to light up and when the switch is released the LED must turn off. The simplest method to check the state of a switch is using the polling method. In this method, the state of pin to which switch is connected, PF4 in this case, is checked continuously in order to determine whether it is at logic high or logic low. Depending on the logic state determined, the LED is either turned on or off. The basic outline of program will be as follows:

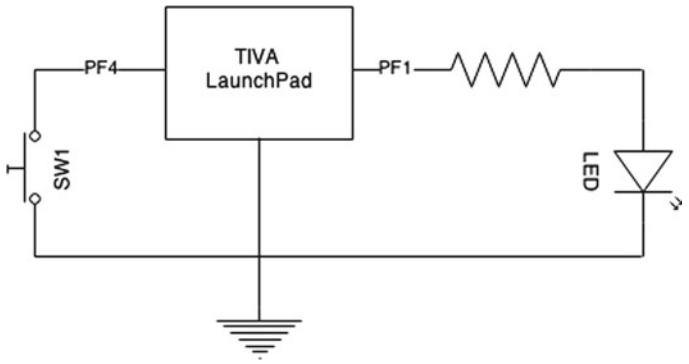

Fig. 4 Block diagram for switchy

1. Enable the system clock and GPIO port F.
2. Set the direction of pin PF1 as output and pin PF4 as input.
3. Enable the internal pull-ups on input pin, PF4.
4. Check the state of pin PF4.
5. If PF4 is at logic low state, then switch is pressed and LED is turned on.
6. If PF4 is at logic high state, which is the default state of the pin due to the internal pull-up, implying that switch is not pressed and hence, LED is turned off.

Figure 5 shows the flowchart of program flow for the experiment.

2.4 Useful API Functions Calls

In addition to the API functions used for configuring and writing digital output in the previous experiment, a few more API functions associated with GPIO input are required for this experiment. API functions listed below will be used to configure GPIO pin as digital input, to enable internal pull-ups present in the microcontroller as well as to read digital input state of GPIO pin.

- **GPIOPinTypeGPIOInput** - Configures the pin(s) as GPIO inputs.
  Prototype: void GPIOPinTypeGPIOInput(uint32_t ui32Port, uint8_t ui8Pins)
  Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s).
  Description: This function configures the pins defined by ui8Pins on the GPIO port ui32Port as general purpose input pins. This initial configuration is essential for the pins to function correctly as digital inputs.
  Returns: None.
- **GPIOPadConfigSet—**Sets the pad configuration for the specified pin(s).
  Prototype: void GPIOPadConfigSet(uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32Strength, uint32_t ui32PinType)

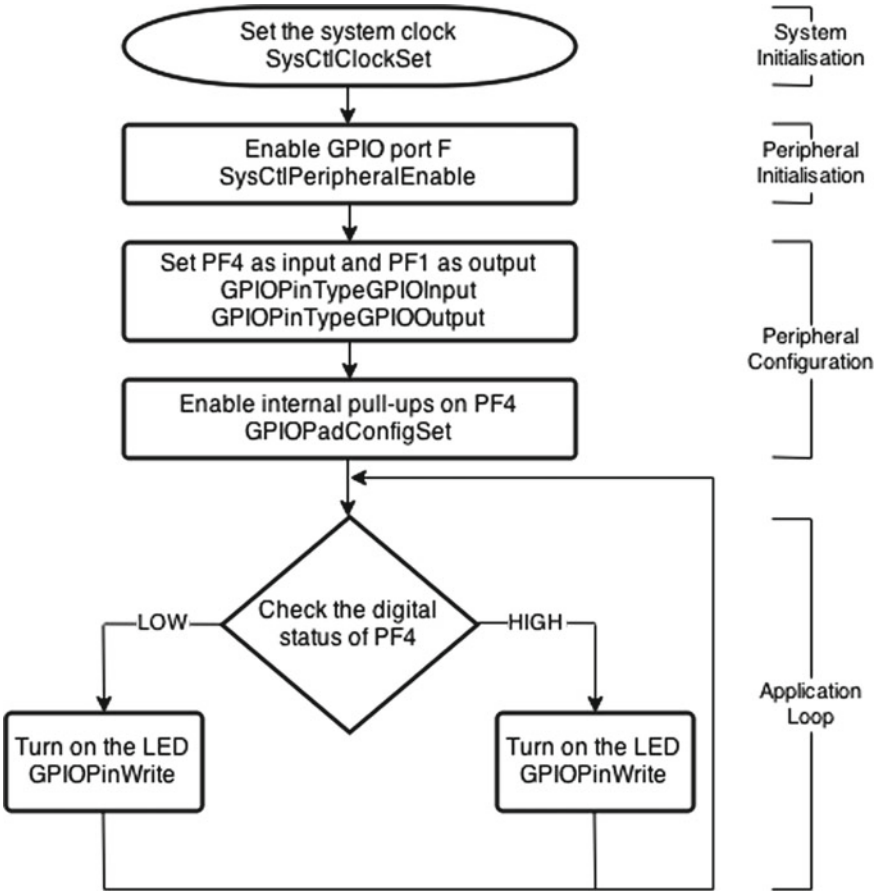

Fig. 5 Program flow for switchy

Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s). ***ui32Strength*** specifies the output drive strength. ***ui32PinType*** specifies the pin type.
Description: This function is used to control various parameters of the GPIO pins. For input pins, the pin can be configured as a standard push–pull or an open drain type with a weak pull-up or a weak pull-down as required by the application. The ui32PinType parameter of the API function is used to set up the required configuration. The possible values of the parameter ui32PinType are:
GPIO_PIN_TYPE_STD
GPIO_PIN_TYPE_STD_WPU
GPIO_PIN_TYPE_STD_WPD
GPIO_PIN_TYPE_OD
GPIO_PIN_TYPE_OD_WPU

GPIO_PIN_TYPE_OD_WPD
GPIO_PIN_TYPE_ANALOG
Here, STD indicated the standard push–pull configuration, OD denotes an open drain configuration and WPU and WPD denote the presence of a weak pull-up or a weak pull-down, respectively. ANALOG type is used on select pins to enable the analog input function of the pin.
For output pins, the ui32Strength parameter can be used to set the drive strength of the digital output pins. This setting has no effect on input pins. The drive strength configuration limits the maximum allowed current on output pins to the preset value. The parameter ui32Strength can be one of the following values:
GPIO_STRENGTH_2MA
GPIO_STRENGTH_4MA
GPIO_STRENGTH_8MA
GPIO_STRENGTH_8MA_SC
Where, **GPIO_STRENGTH_xMA** specifies either 2, 4, or 8 mA output drive strength,
and **GPIO_OUT_STRENGTH_8MA_SC** specifies 8 mA output drive with slew control.
Returns: None.

- **GPIOPinRead—**Reads the value present of the specified pin(s).
  Prototype: int32_t GPIOPinRead(uint32_t ui32Port, uint8_t ui8Pins)
  Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s).
  Description: This function is used to read the digital state of input pins specified in ui8Pins of the port specified in ui32Port. Only the state of the pins specified in ui8Pins is returned. Other bits will have a value of zero.
  Returns: Returns a bit-packed output providing the state of the specified pins, where bit 0 of represents GPIO port pin 0, and so on. Any bit that is not specified by ui8Pins is returned as a 0. Bits 31:8 should be ignored.

2.5 Program Code

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"
```

```
/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

int main(void)
{
/* System Initialization Statements
 * Set the clock to directly run from the crystal at 16MHz */
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Peripheral Initialization Statement
 * Set the clock for the GPIO Port F */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

/* Peripheral Configuration Statement
 * Set the type of the GPIO Pin, PF4 as input and PF1 as output */
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE,GPIO_PIN_4);
GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_4, GPIO_STRENGTH_2MA,
GPIO_PIN_TYPE_STD_WPU);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_1);

/* GPIO Pin 1 on PORT F initialized to 0, RED LED is off */
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,0);

/* Application Loop */
while(1)
  {
  /* Reading the state of input pin PF4
   * If read state is logic low implies switch is pressed
   * Turn on the LED at PF1 */
  if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4)==0)
    {
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
    }
  /* If read state is logic high implies switch is not pressed
   * Turn off the LED at PF1 */
  else
    {
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);
    }
  }
}
```

3 Experiment 5—Running LEDs

3.1 Objective

Produce a blinking LED pattern using the LEDs present on PadmaBoard or a colour pattern using the RGB LED on Tiva LaunchPad.

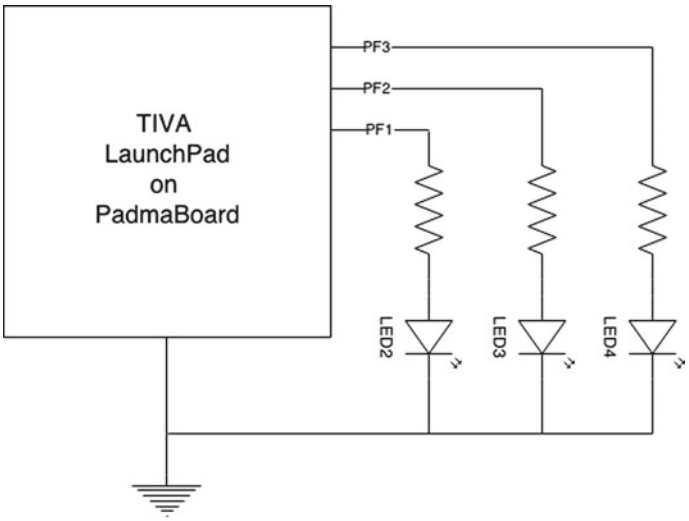

Fig. 6 Block diagram for running LEDs

3.2 Hardware Description

This experiment requires three LEDs for this experiment. Apart from RGB LED on Tiva LaunchPad, the PadmaBoard has three individual LEDs connected to the same GPIO pins. A blinking pattern can be generated using these three LEDs. Simultaneously, a sequence of colors will also be produced on the RGB LED present on Tiva LaunchPad as these are controlled by same GPIO pins PF1, PF2, and PF3. Figure 6 shows the block diagram for the experiment.

Similar to the configuration of RGB LED present on Tiva LaunchPad, the three LEDs on PadmaBoard are also connected using NPN transistors (as shown in Fig. 7) in order to provide sufficient amount of current flow into them.

3.3 Program Flow

This experiment requires to control the three LEDs as to blink in a particular pattern, unlike previous experiments where a single LED was blinking continuously. A simple blink pattern can be generated using the algorithm suggested below:

1. Enable the system clock and GPIO Port F.
2. Set data direction of Pins PF1, PF2 and PF3 as output.
3. Set PF1 to Logic High for 1 second.
4. Set PF1 Logic Low and PF2 to Logic High for next 1 second.
5. Set PF2 Logic Low and PF3 to Logic High for next 1 second.

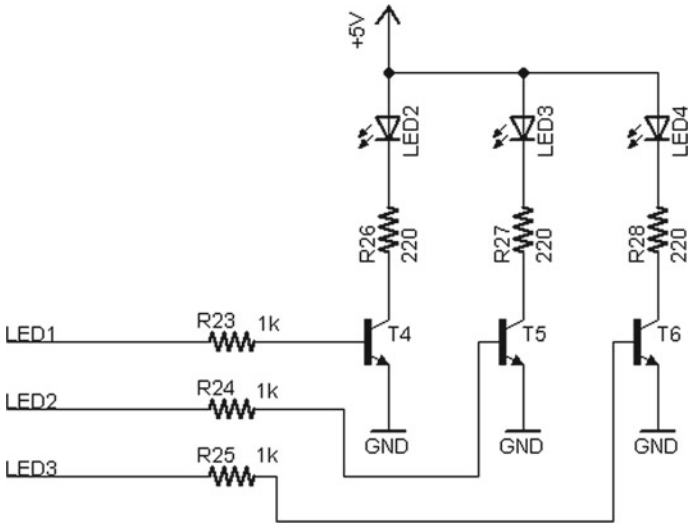

Fig. 7 Schematic of three LEDs on PadmaBoard

6. Repeat to generate a pattern.

A suggested program flow for the experiment is shown in Fig. 8. This program will not use any new API function calls. Only the functions for configuring GPIO pins as a digital output and setting its state to logic low or high will be used in the program. This example will illustrate how to control multiple digital output pins simultaneously using a single API function call.

4 Experiment 6—LED as Light Sensor

4.1 Objective

Blink a LED at a rate proportional to the intensity of light falling on it.

4.2 Hardware Description

Apart from the ability to emit light when subject to a forward biasing voltage, LEDs are capable of measuring the intensity of light falling on them. When subjected to a reverse biasing voltage, the LED acts as a small capacitor storing charge. When the reverse bias is removed, the rate at which the accumulated charge dissipates is proportional to the intensity of light falling on it. To enable this kind of operation,

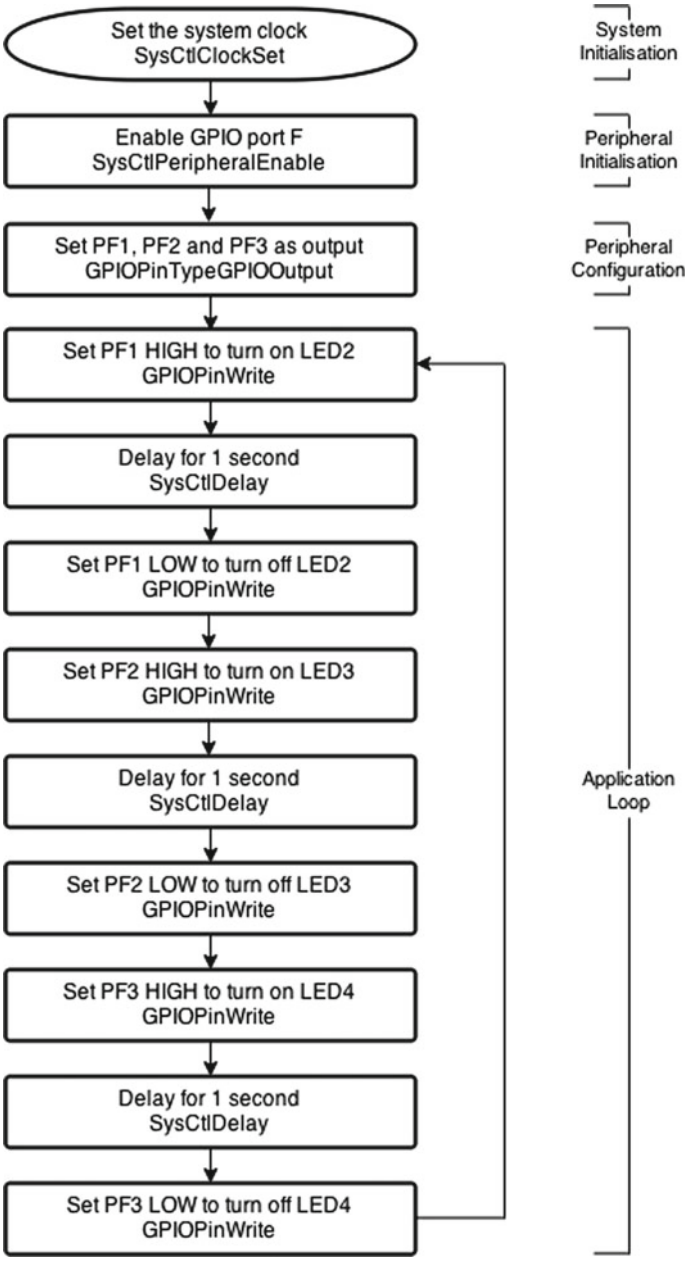

Fig. 8 Program flow for running LEDs

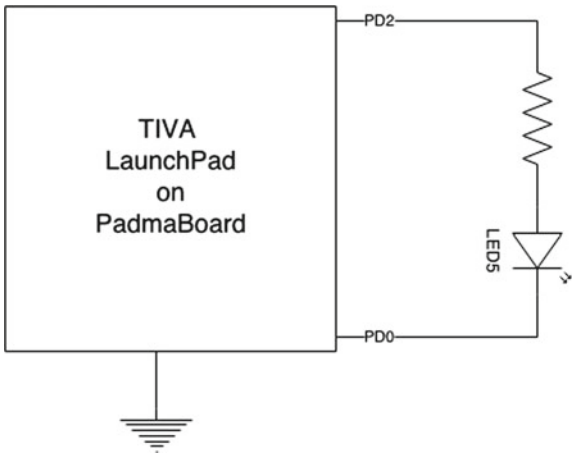

Fig. 9 Block diagram for LED as light sensor

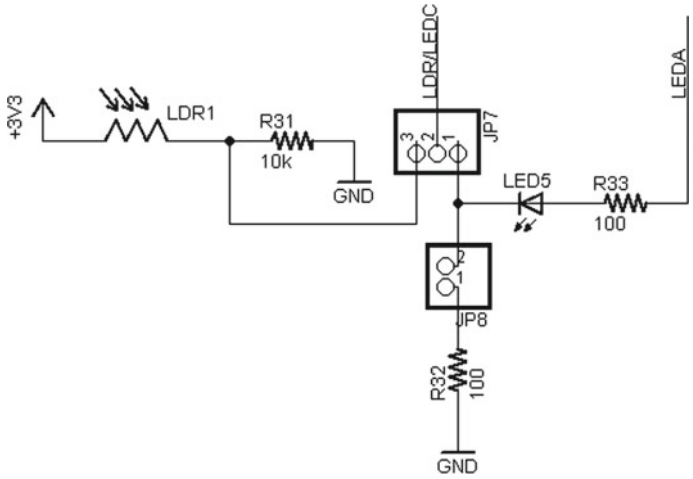

Fig. 10 Schematic of light sensor using LED and LDR on PadmaBoard

the biasing configuration of the LED must be switched between forward and reverse bias. This requires both the ends of the LED to be connected to GPIO pins of the microcontroller. LED5 present on the PadmaBoard is wired in this configuration using GPIO pins PD0 and PD2 as shown in Fig. 9.

On the PadmaBoard, the anode terminal of LED5 is connected to pin PD2 through 100 ohm current limiting resistor. The connection to the cathode terminal of LED5 is multiplexed with the LDR (Light-Dependent Resistor) present on the PadmaBoard using jumper JP7. To enable the use of LED5 as light sensor, the jumper must be

connected so as to join the cathode terminal of LED5 to pin PD0 and remove the jumper JP8 as shown in Fig. 10.

4.3 Program Flow

In order to measure the intensity of light using a LED, it is first set to reverse bias mode. After a short while, the reverse bias is removed and the voltage level of the cathode is read. The time taken for the cathode voltage to drop to logic low level is measured using a counter. Once the measurement is completed, the same LED is set to forward bias mode and made to blink at a rate proportional to the counter value.

An algorithm which can be followed for the experiment is:

1. Enable the system clock and GPIO port D.
2. Set direction of pins PD0 and PD2 as output.
3. Set PD0 High and PD2 Low. This puts the LED in reverse bias mode.
4. Change the direction of PD0 to input. This brings the LED out of reverse bias mode.
5. Start a counter and count until PD0 goes to logic low. The value of counter is proportional to the light intensity.
6. Now, change the direction PD0 back to output.
7. Set PD0 Low and PD2 High. This puts the LED in forward bias mode, hence turning it on.
8. Provide a delay proportional to the counter value.
9. Set PD0 and PD2 Low. This turns the LED off.
10. Provide a delay proportional to the counter value.
11. Repeat the Steps 3–10 to keep LED blinking proportionally to the light intensity.

A suggested program flow for the experiment is shown in Fig. 11.

5 Experiment 7—Switch Toggle

5.1 Objective

Toggle the state of LED, connected to PF1 using the switch, SW1. Example, when switch SW1 is pressed turn on the LED, and when switch SW1 is again pressed turn off the LED. Like this keep toggling the state of LED whenever the switch is pressed.

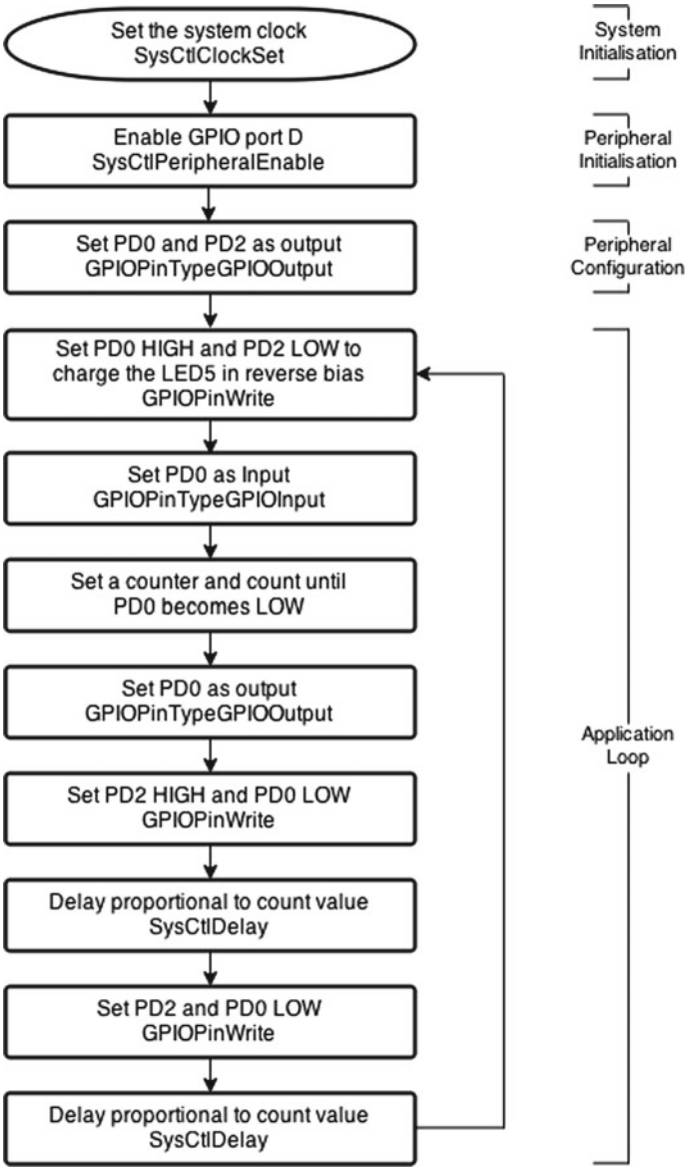

Fig. 11 Program flow for LED as light sensor

5.2 Hardware Description

This experiment will require only one LED, whose state will be toggled and one switch, SW1. The hardware of this experiment will be exactly the same as that of Experiment 4.

5.3 Experiment Tips

Initialize a flag to 0. When the switch is pressed, set the flag to 1 and give some small delay. When the switch is pressed again clear, the flag to zero and give some delay. Delay is required so that the switch is released within that time, if switch is not released within that it may be toggle the flag bit again. Corresponding to the flag bit, turn on or turn off the LED.

6 Experiment 8—Electronic Dice

6.1 Objective

Emulate the function of a dice by displaying a random number between 1 and 6 using 3 LEDs in binary form.

6.2 Hardware Description

A random number between 1 and 6 can be displayed in binary form using 3 LEDs. This experiment will use the three red LEDs available on the PadmaBoard. Example, a binary representation of number 5 is equivalent to a binary value of 101. This can be visualized as first and third LED of a set of the three LEDs being turned on and the second LED is turned off. As discussed previously, these LEDs are controlled by GPIO pins PF1, PF2, and PF3. The experiment will need a switch to trigger the random number generation and update corresponding number on LEDs. The switch SW1 on Tiva LaunchPad can be used for this purpose (Fig. 12).

6.3 Experiment Tips

Start a counter which counts from 1 to 6, when counter reaches value 6 reset is back to value 1. When the switch is pressed, capture that count value. It will result in a

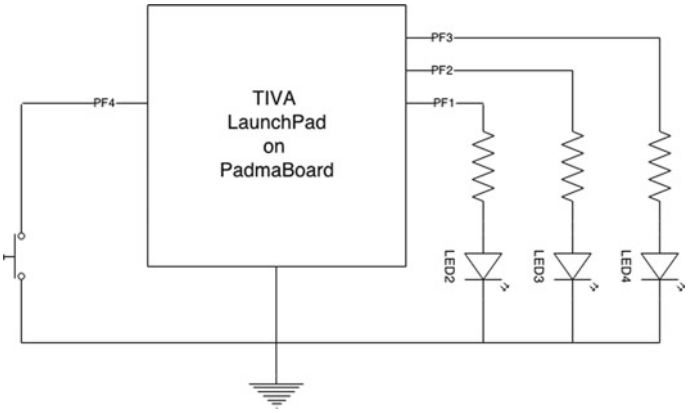

Fig. 12 Block diagram for electronic dice

reasonably random number generated between 1–6. This count can be displayed on the LEDs when the switch is released.

7 Experiment 9—Live Morse Generation

7.1 Objective

Generate the Morse code using two switches SW1 and SW2 on Tiva LaunchPad and buzzer on PadmaBoard. As Morse code is combination of dots and dashes which can be mapped on tones, lights, or clicks which carry coded form of text information. Use SW1 to generate dot and SW2 to generate dash, and at the same time, generate a tone corresponding to dot or dash on the buzzer.

7.2 Hardware Description

This experiment requires two switches, SW1 and SW2 which are connected to PF4 and PF0, respectively, to input Morse code, and a buzzer to generate tone corresponding to dot or dash. Buzzer is connected to PD7. Figure 13 shows the block diagram for the experiment.

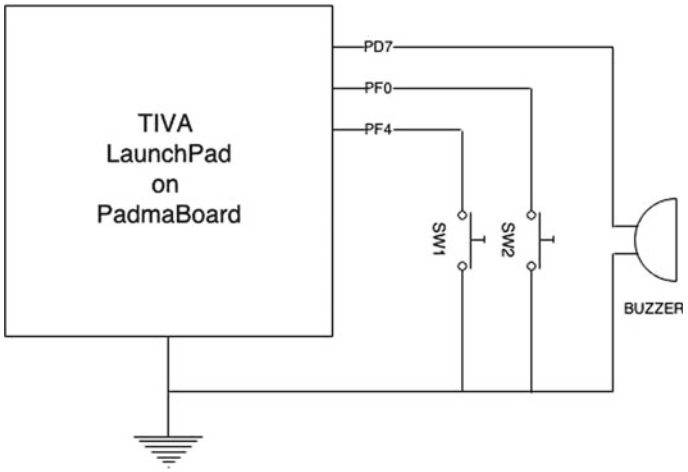

Fig. 13 Block diagram for Morse code generator

7.3 Experiment Tips

If tone for dot is a beep of duration t, then tone for dash should be a beep of duration 3*t, or 3 times the duration of tone for dot. Also, to use pins PF4 and PD7, they need to unlocked first. Give below is the part of code to unlock the PD7, which need to be added after enabling the GPIO Port D. GPIO are registers are define in header file "*inc/hw_gpio.h*", so it is also need to be included in the program code.

```
//
//Program Code to unlock PD7
//
HWREG(GPIO_PORTD_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;

//
// Set the commit register for PD7 to allow changing the function
//
HWREG(GPIO_PORTD_BASE + GPIO_O_CR) = 0xC0;

//
// Enable the alternate function for PD7 (U2TX)
//
HWREG(GPIO_PORTD_BASE + GPIO_O_AFSEL) |= 0x600;

//
// Turn on the digital enable for PD7
//
HWREG(GPIO_PORTD_BASE + GPIO_O_DEN) |= 0xC0;

//
// Relock the commit register
//
```

```
HWREG(GPIO_PORTD_BASE + GPIO_O_LOCK) = 0;
```

To generate a beep on buzzer, produce a square wave of equal high and low time (50% dutty cycle) for duration of time t. Mentioned square wave can be produced by toggling the GPIO with equal amount of off delay and on delay.

8 Experiment 10—Morse Recorder

8.1 Objective

Morse code is generated using switches SW1 and recorded in the LaunchPad, and then on pressing switch SW2, recorded Morse code is played through buzzer.

8.2 Hardware Description

Hardware description of this experiment is same as that of Experiment 9.

9 Experiment 11—Car Parking Sensor

9.1 Objective

Design a car parking sensor using ultrasonic module and buzzer, where the beep rate increases as the distance decreases.

9.2 Hardware Description

This will require ultrasonic module and a buzzer, on PadmaBoard there is a buzzer and a connector for ultrasonic module. Figure 14 shows the block diagram of ultrasonic module and buzzer.

The ultrasonic module requires two GPIO pins, trigger pin and echo pin. In PadmaBoard trigger pin of ultrasonic module connector is connected to PA6 through a jumper, JP3 and echo pin of ultrasonic module connector is connected to PA7 through a jumper, JP4.

So, while using ultrasonic module connector, place the jumpers JP3 and JP4 towards the side of ultrasonic module connector on PadmaBoard. Jumper selection

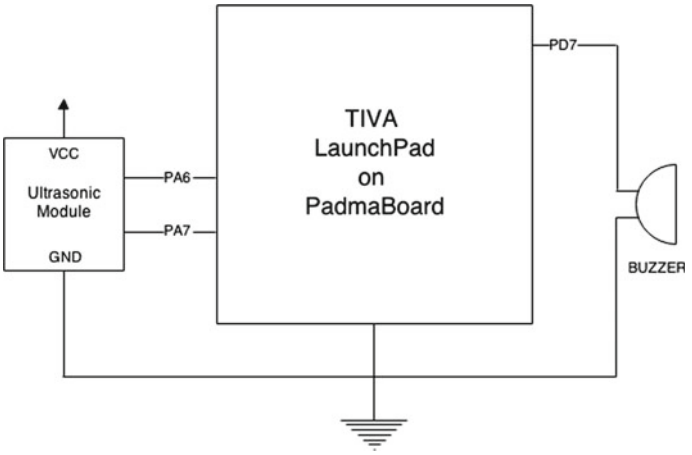

Fig. 14 Block diagram for car parking sensor

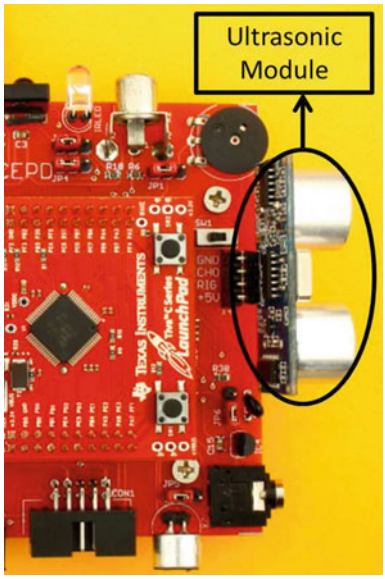

Fig. 15 Placement of ultrasonic module

on PadmaBoard is explained in Chap. 4 under sect. 5. Place the ultrasonic module on connector X4 as shown in Fig. 15.

9.3 Experiment Tips

While using Ultrasonic module first a short trigger pulse is sent on trigger pin, PA6 and then read the input pulse on the echo pin, PA7 whose width will be proportional to the distance of object in front of ultrasonic module. So sound the buzzer with the beep rate proportional to width of the pulse received at echo pin. Beep rate of the buzzer can be varied by varying the frequency of square wave given on buzzer pin, PD7. Since, buzzer is connected to PD7, unlock PD7 pin before using it as discussed in Experiment 9.

Chapter 10
Interrupts

This chapter deals with generation of interrupts on ARM Cortex M4 Tiva C Series microcontrollers. Whenever a switch is pressed, there are two ways to read its input, one is through polling and other is through interrupt. The algorithm mentioned in Chap. 9 in Sect. 2 is polling method. In polling, processor is always checking the state of the pin, which is actually wasting the clock cycles as there can be the situation at which the input is changing at much slower rate than the rate at which processor is checking the state of pin. These clock cycles could be used to perform some other task which can improve the efficiency of the processor. As when switch is pressed, state is changing from logic high to logic low, and this state change on the GPIO pin itself requests the processor to perform certain task. This will lead to development of efficient system and these requests to the processors are known as interrupts.

Similar to interrupts, there are other term exceptions. Interrupts are the asynchronous events triggered by the external events like serial ports, GPIOs, etc. By calling asynchronous events mean, that these interrupts are not in synchronization with the processor instruction execution. So, whenever the interrupt occurs first, processor will complete the instruction which it was executing during the occurrence of interrupt and then it will service the interrupt. But the exception are also caused by the events but these are in sync with the processor instruction like software reset, hardfaults, etc.

1 Exception Handling

The ARM Cortex M4 processor along with the Nested Vectored Interrupt Controller (NVIC) prioritizes and handles the exceptions. Whenever the processor services the exception, it pushes the current state of the processor in stack and after servicing the

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_10

exception it pops back the initial state from stack to resume back the ongoing task. The NVIC also supports tail-chaining[1] of interrupts.

1.1 Exception States

There are various states for the exception which are discussed below:

- **Inactive**: It implies that the exception is not active and since it is not active so it cannot be in pending state also.
- **Pending**: It implies the exception is waiting for the processor to get serviced. It can be an interrupt request from the external peripheral or from the software which has changed the state of corresponding interrupt to be pending.
- **Active**: It is an exception that is being serviced by the processor but has not completed.
- **Active and Pending**: The exception that is being serviced by the processor and there is a pending exception from the same source which is waiting to be serviced.

1.2 Exception Types

The exception that can occur are of various types mentioned below:

- **Reset**: Reset is treated as the special form of exception with highest priority. When reset is asserted, it stops the processor at any point of the instruction. When it is de-asserted, processor starts executing the instruction from the address provided by the reset entry in the vector table.
- **Non Maskable Interrupt (NMI)**: A NMI can be asserted by the peripheral or by the software. It is at second highest priority after reset. NMI can not be masked or preempted by the other exceptions other than Reset.
- **HardFault**: HardFault occurs due to the error during processing the exceptions or when the exceptions are not managed properly by the exception handling mechanism. These HardFaults are at higher priority than any other exceptions with configurable priority.
- **MemManage**: MemManage fault occurs because of a memory protection related fault. The memory protection constraints which are defined with the MPU (Memory Protection Unit) determines his fault for both instruction and data transactions.
- **BusFault**: BusFault occurs because of a memory related fault for instruction data memory transactions.

[1]Tail-Chaining mechanism speeds up the servicing of exceptions. As the new exception can occur during the servicing of exception. So on completion of exception, if there is a pending exception the stack pop which is required to resume back the processor state will be skipped and the control will transfer to the new exception hander.

Table 1 Properties of exception types

Exception type	Exception number	Priority	Vector address
Reset	1	–3, highest	0 × 00000004
NMI	2	–2	0 × 00000008
HardFault	3	–1	0 × 0000000C
MemManage	4	Configurable	0 × 00000010
BusFault	5	Configurable	0 × 00000014
UsageFault	6	Configurable	0 × 00000018
–	7–10 (Reserved)	–	–
SVCall	11	Configurable	0 × 0000002C
–	12–13 (Reserved)	–	–
PendSV	14	Configurable	0 × 00000038
SysTick	15	Configurable	0 × 0000003C
Interrupt (IRQ)	16	Configurable	0 × 00000040

- **UsageFault**: A UsageFault is an exception that occurs because of a fault related to instruction execution. This includes:
  - an undefined instruction
  - an illegal unaligned access
  - invalid state on instruction execution
  - an error on exception return
- **SVCall**: It is an exception triggered by the SVC (Supervisor Calls) instructions. These SVC are normally used to request privilege operations or to access OS kernel functions and device drivers.
- **PendSV**: PendSV is an interrupt-driven request for system-level service. In OS environment, PendSV is used for context switching[2] when no other exception is pending.
- **SysTick**: SysTick is a decrement counter, so it generates an exception on reaching zero value. In an OS environment, the processor can use this exception as system tick.
- **Interrupt (IRQ)**: Interrupts as discussed in previous section are asynchronous to the instruction execution and are generated by the peripherals like GPIOs, UART, I2C, SSI, ADC, Timers, etc.

The priority levels along with their vectored address of the exception types are mentioned in Table 1.

[2]Context switching is storing and restoring back the current state of the process so that it can be resumed from that point at a later time. This allows multiple processes to have access to processor and will lead to multitasking.

1.3 Exception Handler

The processor handles the various exceptions using the following handlers:

- **Interrupt Service Routines (ISRs)**: The IRQ interrupts are the exceptions handled by ISRs.
- **Fault Handlers**: HardFault, MemManage fault, UsageFault, and BusFault are fault exceptions handled by fault handlers.
- **System Handlers**: NMI, PendSV, SVCall SysTick, and the fault exceptions are all system exceptions that are handled by system handlers.

1.4 Exception Priorities

All the exception types have the priorities associated with them. Priorities help in managing the exceptions by the processor. All the exceptions have priority associated with them, the lower the priority value the higher will be the priority. Example reset with priority—3 is highest priority exception. These priorities are configurable for all exceptions except Reset, NMI, and HardFault.

If priorities of pending exceptions are same, then exception with lowest exception number will be processed first. When a processor is executing an exception handler and a new exception occurs with higher priority value then the current exception will be preempted by the new exception (one with higher priority value). If new exception occurred is of same or low priority value then current execution of exception handler will not be preempted even if the exception number of the newly occurred exception is low.

To increase the priority control in systems, NVIC supports group priority mechanism. In this mechanism, interrupt priority register will consist of two fields group priority and the sub-priority within the group. Only the group priority decides the preemption of the current executing handler. Only the interrupts of higher group priority are able to preempt the handler. If multiple pending interrupts have same group priority then sub-priority will decide the order in which they are processed. If multiple pending interrupts have same group priority and sub-priority, the interrupt with the lowest IRQ number is processed first.

2 Experiment 12—Interrupt Switchy

2.1 Objective

Generate an interrupt using a switch, SW1 and change the state of Blue LED on every 5th interrupt generation.

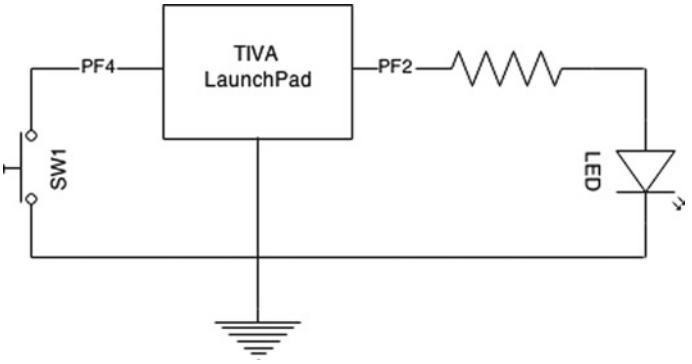

Fig. 1 Block diagram for interrupt switchy

2.2 Hardware Description

This experiment will require one tactile switch to generate interrupts and one LED. Both of these are present on the Tiva LaunchPad. Since, in Chap. 9, Sect. 2 discusses how to detect when switch, SW1 connected to PF4 is pressed. Also, LED connected PF4 which is connected in active high configuration is used in experiment. Figure 1 shows block diagram of hardware described.

2.3 Program Flow

The experiment requires to generate the external interrupt whenever the switch, SW1 is pressed. On every 5th interrupt generation toggle, the state of LED, connected to PF2. Algorithm to achieve the mentioned objective is described below:

1. Enable the system clock and GPIO Port F.
2. Set the direction of pin PF2 as output and PF1 as input.
3. Initialize the state of pin PF2 (to which LED is connected) as logic low.
4. Enable the internal pull up on PF4.
5. Register the interrupt service routine to Port F and configure the interrupt for falling edge[3] on the PF4.
6. Now clear the interrupt on PF4 and then enable the interrupt.

[3]Falling edge occurs when the state changes from logic high to logic low and rising edge occurs when the state changes from logic low to logic high. The switch connected is configured to be pulled up so the default state of switch is logic high. Since, the other end of switch is connected to ground, so when the switch is pressed it changes the state from logic high to logic low which is the falling edge, and when the switch is released the state changes from logic low to logic high which is rising edge.

7. In the interrupt service routine call, have a counter variable which increments based on the number of times interrupt occurred.
8. Whenever counter variable reaches 5, toggle the state of the LED and reset the counter value back to 0.

Figure 2 describes the program flow of the experiment in the flowchart manner.

2.4 Useful API Function Calls

To generate the interrupts mentioned below are the some necessary API function calls.

1. **GPIOIntRegister** - It registers the interrupt handler for a GPIO port.
   Prototype: void GPIOIntRegister(uint32_t ui32Port, void (*pfnIntHandler) (void))
   Parameters: ***ui32Port*** is the base address of the GPIO port. ***pfnIntHandler*** is a pointer to the GPIO port interrupt handling function.
   Description: This function ensures that the interrupt handler specified by pfnIntHandler is called when an interrupt is detected from the selected GPIO port. This function also enables the corresponding GPIO interrupt in the interrupt controller.
   Returns: None.
2. **GPIOIntTypeSet** - Sets the interrupt type for the specified pins.
   Prototype: void GPIOIntTypeSet(uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32IntType)
   Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s). ***ui32IntType*** specifies the type of interrupt trigger mechanism.
   Description: This function sets up the various interrupt trigger mechanisms for the specified pin(s) on the selected GPIO port. The parameter ***ui32IntType*** is an enumerated data type that can be one of the following values:
   GPIO_FALLING_EDGE , GPIO_RISING_EDGE
   GPIO_BOTH_EDGES , GPIO_LOW_LEVEL
   GPIO_HIGH_LEVEL , GPIO_DISCRETE_INT
   where the different values describe the interrupt detection mechanism (edge or level) and the particular triggering event (falling, rising, or both edges for edge detect, low or high for level detect).
   Some devices support discrete interrupts for each pin on a GPIO port, giving each pin a separate interrupt vector. To use this feature, the GPIO_DISCRETE_INT can be included to enable an interrupt per pin. The GPIO_DISCRETE_INT is not available on all devices or all GPIO ports, consult the data sheet to ensure that the device and the GPIO port supports discrete interrupts.
   Returns: None.

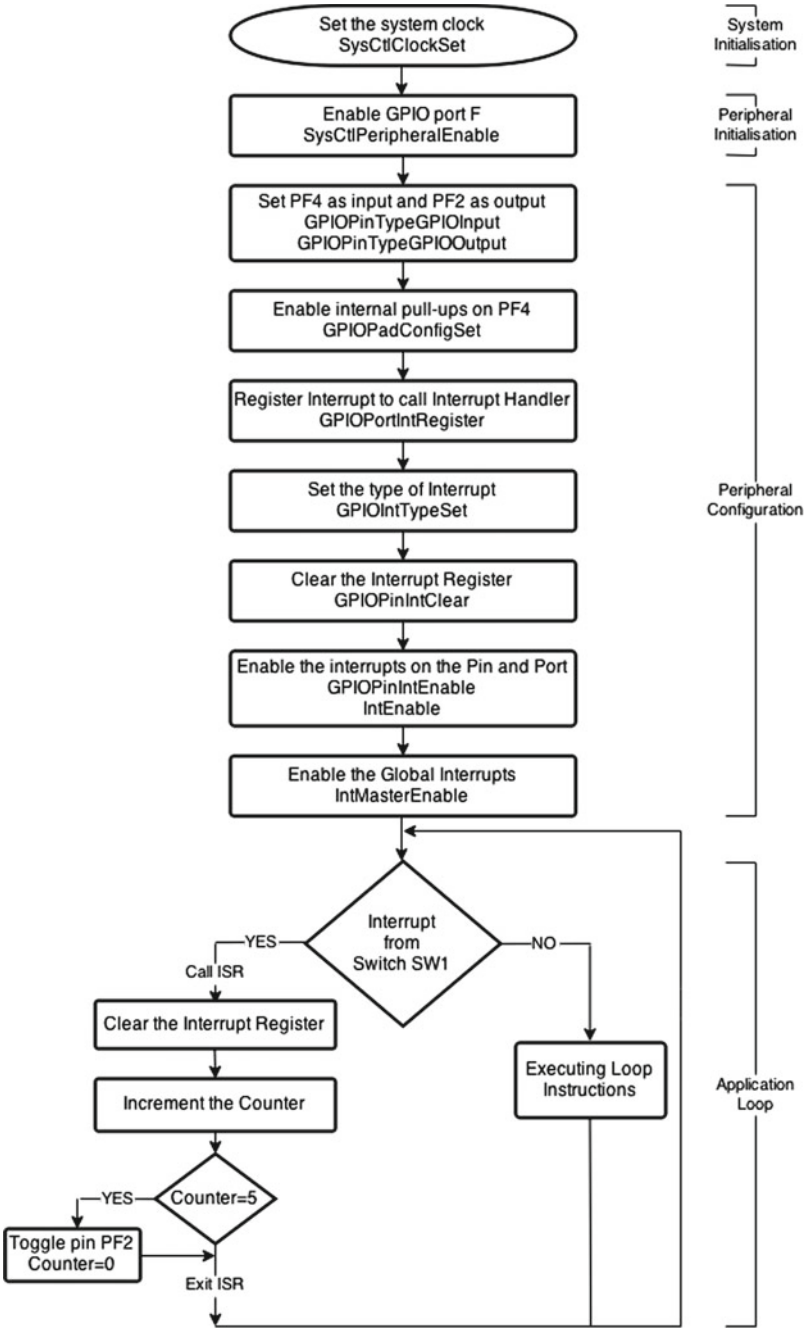

Fig. 2 Program flow for interrupt switchy

3. **GPIOIntClear** - It the clears interrupt for the specified pin.
   Prototype: void GPIOIntClear(uint32_t ui32Port, uint32_t ui32IntFlags)
   Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui32IntFlags*** is the bit mask of the interrupt sources to disable.
   Description: It clears the interrupts for the specified pin, so that the another interrupt on the same pin can be detected again. Usually, it takes few clock cycles to clear the interrupt after calling the function. Hence, it is recommended to call this function early in the interrupt handler. The parameter ***ui32IntFlags*** is the logical OR of the following values:
   GPIO_INT_PIN_0—interrupt due to activity on Pin 0.
   GPIO_INT_PIN_1— interrupt due to activity on Pin 1.
   GPIO_INT_PIN_2—interrupt due to activity on Pin 2.
   GPIO_INT_PIN_3—interrupt due to activity on Pin 3.
   GPIO_INT_PIN_4—interrupt due to activity on Pin 4.
   GPIO_INT_PIN_5—interrupt due to activity on Pin 5.
   GPIO_INT_PIN_6—interrupt due to activity on Pin 6.
   GPIO_INT_PIN_7—interrupt due to activity on Pin 7.
   GPIO_INT_DMA - interrupt due to DMA activity on this GPIO module.
   Returns: None.
4. **GPIOIntEnable** - It enables the interrupt for the specified pin.
   Prototype: void GPIOPinIntEnable(uint32_t ui32Port, uint32_t ui32IntFlags)
   Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui32IntFlags*** is the bit mask of the interrupt sources to disable.
   Description: It unmasks the interrupt on the specified pin or enables the interrupt on the particular pin or source.
   Returns: None.
5. **IntEnable** - It enables the interrupt from the source.
   Prototype: void IntEnable(uint32_t ui32Interrupt)
   Parameters: ***ui32Interrupt*** specifies the interrupt to be enabled.
   Description: The specified interrupt source is enabled in the interrupt controller. For example, to enable interrupts on PORTF, parameter is **INT_GPIOF**.
   Returns: None.

Chapter 11
Timer and Counters

This chapter deals with the programmable timers. These timers can be used to count or time the external events. ARM provides the timer along with the processor termed as SysTick timer, as a core peripheral. Then, while developing the microcontrollers IP companies like Texas Instruments, Freescale, etc., add the additional timer modules around the processor. These timer modules are generally added as they are multifunctionality timers like count up or down, input edge count,[1] daisy chaining of timers, and much more. With the help of timers, timed events can be generated. Also, with the help of timers, pulses of different duty cycles known as Pulse Width Modulation (PWM) can be generated. Generally, PWM is used to perform control operation like controlling the speed of motor, controlling the intensity of light, etc. Watchdog timer is also discussed in this chapter, this timer is generally used to monitor the system, and help the system to regain control from some erroneous halted state.

1 Systick Timer

ARM Cortex M4 processor has an integrated system timer, known as SysTick, which is a 24-bit down count timer. This counter decrements with every system clock.[2] Hence, it generally helps in determining the number of system clocks taken by the task to complete or to have a very speed alarm timer. This timer consists of three registers:

[1] Incrementing or decrementing the counter value at rising or falling edge of the external input.

[2] The clock source of SysTick timer can be selected between the processor clock and external clock through a SysTick control register (STCTRL).

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_11

- **SysTick Control and Status (STCTRL)**: It is a control and status register of the Systick timer. Through this register, timer can be enabled disabled, or it can enable the generation exceptions when count reaches a zero value. As part of configuration, the clock source for the timer can be the processor clock or the external clock. And, also it can be checked whether the timer has reached to value 0 since the last time SysTick counter value was read.
- **SysTick Reload Value (STRELOAD)**: As the Systick counter is 24-bit, so the 24 least significant bits of this register are the 24-bit reload value of the counter. When counter reaches zero, it restarts down counting from the reload value present in this register.
- **SysTick Current Value (STCURRENT)**: This register contains the current value of the SysTick counter. As the value of the counter can be a maximum of 24 bits, so the least significant 24 bits of this register specify the counter value. A write operation of any value to this register will clear this register. And, count flag bit (bit in the STCTRL which is set when the SysTick timer counted to 0 since last time this was read.) will be cleared to 0.

The correct order to implement the SysTick timer is mentioned below:

1. Program the reload value in the STRELOAD register of SysTick Timer.
2. Clear the STCURRENT register by writing any value to the register.
3. Configure the SysTick timer for required operation through register STCTRL.

2 General Purpose Timers

General Purpose Timers are the programmable timers used to count the events or to perform timed events on the Timer pins. Generally Tiva C Series family has 16/32-bit GPTMs (General Purpose Timer Module) and 32/64-bit GPTMs. 16/32-bit GPTM signifies that it has two 16-bit timers, Timer A and Timer B, which can be used independently, but these two timers can be concatenated to perform as 32-bit timer. Similarly, 32/64-bit GPTM has two 32-bit timers, Timer A and Timer B which can be used independently, and these two timers can also be concatenated to perform as 64-bit timer. Number of 16/32-bit GPTMs or 32/64-bit GPTM varies from microcontroller to microcontroller in the Tiva C Series family. The features for the GPTM are mentioned below:

- 16- or 32-bit programmable one shot timer for 16/32-bit GPTM and 32- or 64-bit programmable one shot timer for 32/64-bit GPTM.
- 16- or 32-bit programmable periodic timer for 16/32-bit GPTM and 32- or 64-bit programmable periodic timer for 32/64-bit GPTM.
- 32-bit real-time clock for 16/32-bit GPTM and 64-bit real-time clock when using an external 32.768 clock as input.

- 16-bit input edge count with 8-bit prescaler[3] for 16/32 bit GPTM or 32-bit input edge count with 16-bit prescaler for 32/64-bit GPTM.
- 16-bit PWM mode with 8-bit prescaler for 16/32 bit GPTM or 32-bit PWM mode with 8-bit prescaler for 32/64-bit GPTM.
- Can be configured as count up or down.
- ADC event trigger.

3 Watchdog Timer

The watchdog timer as the name suggests is used for system monitoring applications. The watchdog timer can generate reset, a non-maskable interrupt, or a regular interrupt on reaching a time-out value. The watchdog timer has the lock register (WDTLOCK) which helps to prevent the timer configuration to be altered unintentionally.

The watchdog timer is a 32-bit down counter with the programmable load value. The watchdog timer generates the time-out signal when reaching to a value 0. At time-out, watchdog timer will generate the interrupt which can be programmed to be a regular interrupt, a non-maskable interrupt or a reset through the control register of watchdog timer (WDTCTL). Enabling the watchdog interrupt will also enable the watchdog timer. The sequence to configure watchdog timer is:

1. Watchdog timer load register (WDTLOAD) is loaded with the desired timer load value.
2. If using watchdog module 1, WDT1, wait for WRC bit in WDTCTL register to be set.[4]
3. Select the interrupt to be generated at time-out using the bits INTTYPE and RESEN in ADTCTL register.
4. While using watchdog module 1, WDT1, wait for WRC bit in WDTCTL register to be set.
5. Set the INTEN bit in the WDTCTL register to enable the interrupts and watchdog timer.

The watchdog registers can be locked by writing any value to the WDTLOCK register. To unlock the watchdog timer, write value of 0x1ACCE551 to WDTLOCK register.

[3]Prescaler are the frequency dividers, used to lower the operational frequency by integer division.

[4]If WRC bit in WDTCTL register is set in watchdog module 1, WDT1 implies that WDT1 register can be read or written.

4 Experiment 13—Software PWM

4.1 Objective

Vary the intensity of light emitted by LED using delay function or software PWM.

4.2 Hardware Description

To perform this experiment, LED connected to PF1 is used. Hardware description for LED is explained earlier in Chap. 9 under Sect. 1.

4.3 Program Flow

To generate the PWM using delay functions is the easiest. PWM is square wave whose on (logic high) and off (logic low) time may be different. To achieve a PWM of 25% duty cycle, turn on the GPIO pin and give a delay of X cycles, then turn off the GPIO pin and give a delay of 3X cycles and repeat the procedure. This will generate a PWM of 25% on GPIO pin. So, to change the intensity of LED to 25%, generate a PWM of 25% duty cycle at the GPIO pin to which LED is connected. But the frequency of PWM should not be too low else flickering effect will be observed at the LED. Due to persistence of vision, human flicker fusion threshold is 16Hz. Therefore, frequency of PWM should be much greater than 16Hz for better intensity variation of LED. The duty cycle of the PWM can be varied by changing the logic high and logic low time of the GPIO pin. Program flow for the experiment is given below:

1. Enable the system clock and GPIO Port F.
2. Set the direction of pin PF1 as output.
3. Set the counter value, this value determines the number of steps to change intensity of light from minimum to maximum.
4. Set the state of pin PF1 as logic low and give a delay proportional to the counter value.
5. Set the state of pin PF1 as logic high and give a delay proportional to Initial value of counter - current Value of counter.
6. Decrement the counter value and repeat the Steps 4 and 5, till minimum counter value is reached. This will make the LED intensity increasing gradually.
7. Set the counter value, this value determines number of steps to change intensity of light from maximum to minimum.
8. Set the state of pin PF1 as logic low and give a delay proportional to Initial counter value—current counter value.

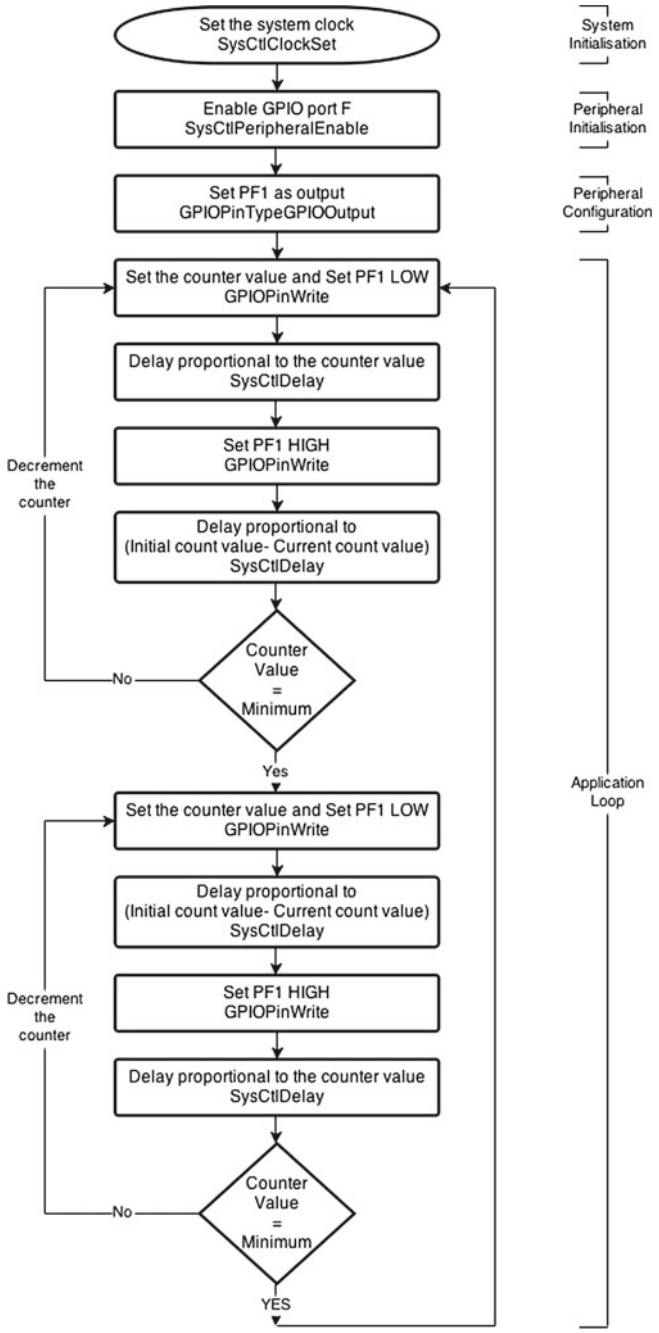

Fig. 1 Program flow for software PWM

9. Set the state of pin PF1 as logic high and give a delay proportional to current counter value.
10. Decrement the counter value and repeat the steps 9 and 10, till minimum counter value is reached. This will make the LED intensity decreasing gradually.
11. Repeat the above steps to keep increasing and decreasing LED intensity.

Figure 1 shows the program flow for the software PWM.

4.4 Program Code

The complete C program is given below. The code is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Generating 8-bit software PWM */
void pin_pwm(unsigned long ulPort, unsigned char ucPin)
{
volatile unsigned long count=255;
/* Intensity will increase */
for(count=255;count>=1;count--)
  {
  GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,0);
  SysCtlDelay(count*SysCtlClockGet()/100000);
  GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,GPIO_PIN_1);
  SysCtlDelay((256-count)*SysCtlClockGet()/100000);
  }

/* Intensity will decrease */
for(count=255;count>=1;count--)
  {
  GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,0);
  SysCtlDelay((256-count)*SysCtlClockGet()/100000);
  GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,GPIO_PIN_1);
  SysCtlDelay(count*SysCtlClockGet()/100000);
  }
}

int main(void)
```

```
{
/* Set the clock to directly run from the crystal at 16MHz */
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Set the clock for the GPIO Port F */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

/* Set the type of the GPIO Pin */
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1);

/* GPIO Pin 1 on PORT F initialized to 0 */
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1,0);

while(1)
  {
  pin_pwm(GPIO_PORTF_BASE, GPIO_PIN_1);
  }
}
```

5 Experiment 14—Hardware PWM

5.1 Objective

Vary the intensity of three colors of RGB LED[5] by using timers associated with the corresponding pins in PWM mode.

5.2 Hardware Description

This experiment requires a RGB LED, which is available on the Tiva LaunchPad. The red, blue, and green pins of the RGB LED are connected to PF1, PF2, and PF3 as shown in block diagram in Fig. 2. Timer B of 16/32-bit Timer 0 is mapped to pin PF1, Timer A of 16/32-bit Timer 1 is mapped to pin PF2 and Timer B of 16/32-bit Timer 1 is mapped to pin PF3. Hence, these timers will be used to generate the PWM.

5.3 Program Flow

As software PWM was very easy to generate as it was a combination of toggling GPIO and delays. But they are difficult to control, if there is a application in which there is

[5]If the intensity or duty cycles for all three colors of RGB LED are varied with same value, it appears as if intensity of white LED is varying, because combination of red, green, and blue light generates white light.

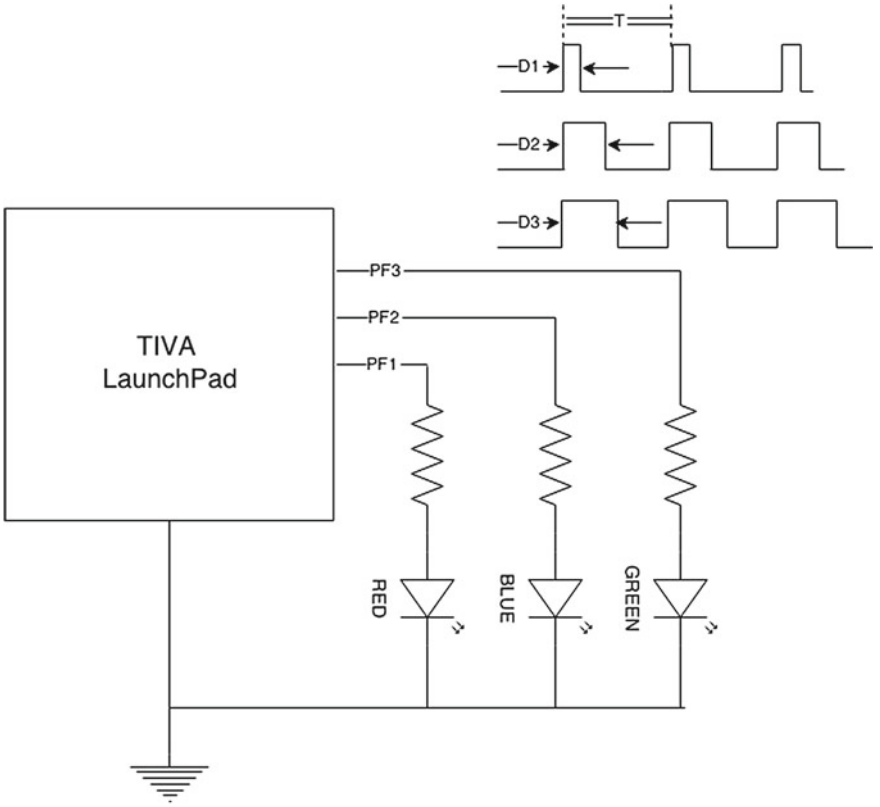

Fig. 2 Block diagram for hardware PWM

calculation task being performed along with the generation of PWM. Then, there will be additional delay occurred due to the other task performed. So, the duty cycle of software PWM can be varied with change in code. Hence, using software PWM is not recommendable for developing complex applications. For such applications, some hardware is required which needs to be configured and it will keep on generating the PWM of defined duty cycle irrespective of the task performed by the application. Such hardware is multifunctionality general purpose timer modules or there are separate PWM modules. PWM modules may or may not be available in every microcontroller but GPTMs will be available as GPTMs have much more functionality than PWM module so it is much preferred to include GPTM in low end microcontrollers than the PWM modules. The algorithm for the experiment is mentioned below:

1. Enable the system clock and GPIO Port F.
2. Enable the peripheral Timer 0 and Timer 1 of 16/32-bit GPTM.
3. Set the type of GPIO pin PF1, PF2, and PF3 as timer pins.
4. Configure the timers to be used as 16-bit and in PWM mode.

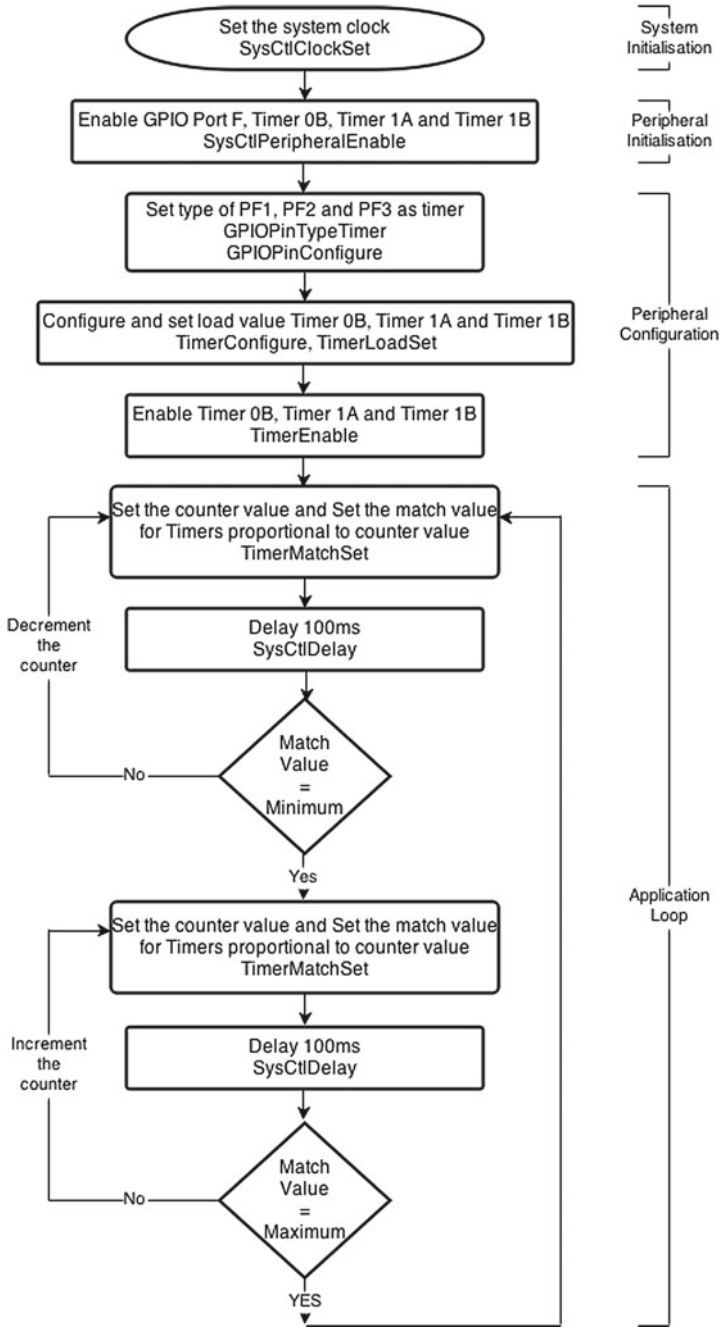

Fig. 3 Program flow for hardware PWM

5. Set the load value for the timers and enable them. This load value will determine the frequency of the PWM generated.
6. Set the counter value, this will be the number of steps in changing intensity of RGB Led from minimum to maximum.
7. Set the match value corresponding to the count value for all three timers, this match value will determine the duty cycle of PWM.
8. Decrement the count value and repeat the Step 7 till match value reaches to minimum value.
9. Set the counter value, this will be the number of steps in changing intensity of RGB Led from maximum to minimum.
10. Set the match value corresponding to the count value for all three timers, this match value will determine the duty cycle of PWM.
11. Increment the count value and repeat the Step 10 till match value reaches the maximum value.

Figure 3 shows the program flow for this experiment along with the API calls that will be used.

5.4 Useful API Function Calls

Mentioned below are some API calls necessary to perform this experiment.

1. **GPIOPinTypeTimer** - It configures the pin to use by the Timer peripheral.
   Prototype: void GPIOPinTypeTimer(uint32_t ui32Port, uint8_t ui8Pins)
   Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s).
   Description: This function configures the Capture Compare PWM (CCP) pins for the timer peripheral.
   Returns: None.
2. **GPIOPinConfigure** - It configures the alternate functionality of the GPIO Pin.
   Prototype: void GPIOPinConfigure(uint32_t ui32PinConfig)
   Parameters: ***ui32PinConfig*** is the pin configuration value, specified as only one of the GPIO_Pxx_xx values.
   Description: Since the GPIO pins are muxed with the multiple peripheral functions, but only one peripheral can access the GPIO at a time. So, that peripheral parameter is defined through this function. The available mappings for the multiple peripheral functions for each pin are defined in pin_map.h under the directory "driverlib".
   Returns: None.
3. **TimerConfigure** - It configures the timer.
   Prototype: void TimerConfigure(uint32_t ui32Base, uint32_t ui32Config)
   Parameters: ***ui32Base*** is the base address of the timer module. ***ui32Config*** is the configuration for the timer.
   Description: This function is used to configure the operating mode of timer. The

timer can be configured as full width timer, i.e., 32-bit in case of 16/32-bit GPTM by using ***ui32Config*** parameter as TIMER_CFG_xxx and can be configured as half width timer by using ***ui32Config*** parameter as TIMER_CFG_A_xxx. Commonly used configuration parameters are mentioned below:

- **TIMER_CFG_ONE_SHOT**: Full width one shot timer.
- **TIMER_CFG_ONE_PERIODIC**: Full width periodic timer.
- **TIMER_CFG_SPLIT_PAIR**: Splits the timer into two half width timers.
- **TIMER_CFG_A_ONE_SHOT**: Half width one shot timer.
- **TIMER_CFG_A_ONE_PERIODIC**: Half width periodic timer.
- **TIMER_CFG_A_CAP_COUNT**: Half width edge count capture.
- **TIMER_CFG_A_PWM**: Half width PWM output.

Returns: None.

4. **TimerLoadSet** - It sets the load value for the timer.
   Prototype: void TimerLoadSet(uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value)
   Parameters: ***ui32Base*** is the base address of the timer module. ***ui32Timer*** specify the timer to be configured like **TIMER_A, TIMER_B or TIMER_BOTH** for half width timer and **TIMER_A** for full width timer. ***ui32Value*** is the load value.
   Description: This function sets the load value for the timer. This function is valid for the both full width and half width timer for 16/32-bit GPTM and half width timer for 32/64-bit GPTM. Use TimerLoadSet64 to set the load value for full width timer of 32/64-bit GPTM.
   Returns: None.
5. **TimerLoadGet** - It returns the load value for the timer.
   Prototype: uint32_t TimerLoadGet(uint32_t ui32Base, uint32_t ui32Timer)
   Parameters: ***ui32Base*** is the base address of the timer module. ***ui32Timer*** specify the timer like either **TIMER_A or TIMER_B** for half width timers and **TIMER_A** for full width timers.
   Description: This function gets the load value of the specified timer.
   Returns: Load value of the specified timer.
6. **TimerMatchSet** - It sets the match value for the timer.
   Prototype: void TimerMatchSet(uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value)
   Parameters: ***ui32Base*** is the base address of the timer module. ***ui32Timer*** specify the timer to be configured like **TIMER_A, TIMER_B or TIMER_BOTH** for half width timer and **TIMER_A** for full width timer. ***ui32Value*** is the match value.
   Description: The match value is used in capture count modes to determine when to generate the interrupts or in PWM it is used to set the duty cycle of the output signal.
   Returns: None.

7. **TimerEnable** - It enables the timer.
   Prototype: void TimerEnable(uint32_t ui32Base, uint32_t ui32Timer)
   Parameters: ***ui32Base*** is the base address of the timer module. ***ui32Timer*** specifies the timer like **TIMER_A, TIMER_B or TIMER_BOTH** for half width timers and **TIMER_A** for full width timers.
   Description: This function enables the specified timer. The timer must be configured before enabling it.
   Returns: None.

5.5 Program Code

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for Timers */
#include "inc/hw_timer.h"
#include "driverlib/timer.h"

/* Defines of the interrupts */
#include "inc/hw_ints.h"

/* Defines and Macros for GPIO API */
#include "inc/hw_gpio.h"
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Defines the Port/Pin Mapping */
#include "driverlib/pin_map.h"

unsigned long i;
void low_to_high(void)
{
for(i=19;i>1;i--)
  {
  TimerMatchSet(TIMER1_BASE, TIMER_A, TimerLoadGet(TIMER0_BASE,
  TIMER_B)*i/20);
  TimerMatchSet(TIMER1_BASE, TIMER_B, TimerLoadGet(TIMER0_BASE,
  TIMER_B)*i/20);
  TimerMatchSet(TIMER0_BASE, TIMER_B, TimerLoadGet(TIMER0_BASE,
  TIMER_B)*i/20);
  TimerEnable(TIMER0_BASE, TIMER_B);
  TimerEnable(TIMER1_BASE, TIMER_A);
  TimerEnable(TIMER1_BASE, TIMER_B);
  SysCtlDelay(SysCtlClockGet()/50);
```

```
  }
}

void high_to_low(void)
{
for(i=1;i<20;i++)
  {
  TimerMatchSet(TIMER1_BASE, TIMER_A, TimerLoadGet(TIMER0_BASE,
  TIMER_B)*i/20);
  TimerMatchSet(TIMER1_BASE, TIMER_B, TimerLoadGet(TIMER0_BASE,
  TIMER_B)*i/20);
  TimerMatchSet(TIMER0_BASE, TIMER_B, TimerLoadGet(TIMER0_BASE,
  TIMER_B)*i/20);
  TimerEnable(TIMER0_BASE, TIMER_B);
  TimerEnable(TIMER1_BASE, TIMER_A);
  TimerEnable(TIMER1_BASE, TIMER_B);
  SysCtlDelay(SysCtlClockGet()/50);
  }
}

int main(void)
{
/* Set the clock to directly run from the crystal at 16MHz */
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Set the clock for the GPIO Port F */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

/* The TIMER0 and TIMER1 peripheral must be enabled for use.*/
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);

/* Configuring PORTF pin1,pin2,pin3 as timer pins */
GPIOPinTypeTimer(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
GPIOPinConfigure(GPIO_PF1_T0CCP1);
GPIOPinConfigure(GPIO_PF2_T1CCP0);
GPIOPinConfigure(GPIO_PF3_T1CCP1);

/* Configuring the TIMER0 and TIMER1 */
TimerConfigure(TIMER0_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_B_PWM);
TimerConfigure(TIMER1_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_B_PWM|
TIMER_CFG_A_PWM);

/* Set the Load value of the Timers */
TimerLoadSet(TIMER0_BASE, TIMER_B, 50000);
TimerLoadSet(TIMER1_BASE, TIMER_A, 50000);
TimerLoadSet(TIMER1_BASE, TIMER_B, 50000);

/* Enable the Timers */
TimerEnable(TIMER0_BASE, TIMER_B);
TimerEnable(TIMER1_BASE, TIMER_A);
TimerEnable(TIMER1_BASE, TIMER_B);

while(1)
  {
  low_to_high();
  high_to_low();
  }
}
```

6 Experiment 15—Systick Timer Blinky

6.1 Objective

Use the SysTick Timer to blink an LED on Tiva LaunchPad.

6.2 Hardware Description

In this experiment, SysTick timer is used to generate the interrupts to blink the LED on the Tiva LaunchPad. The hardware requirements to perform this experiment is same as that of blinky experiment discussed in Chap. 9 Sect. 1.

6.3 Program Flow

This experiment is similar to blinky experiment performed earlier. Only difference is in the way of generating the delay. Load the timer with load value and check for the value until it reaches zero. Hence, SysTick timer is running from load value to zero, then load value and frequency of system clock will help in determining the amount delay. The algorithm to perform this is mentioned below:

1. Enable the system clock, GPIO Port F and SysTick timer.
2. Configure the GPIO pin PF1 as output and initializes it to logic low.
3. Set the period of the SysTick Timer and enable the Systick timer.
4. Toggle the state of the LED when SysTick timer reaches value zero.
5. SysTick Timer will be reloaded again and starts down counting to zero and again toggles the state of LED when timer reaches zero.

A suggested program flow for the experiment is in Fig. 4.

6.4 Useful API Function Calls

Functions useful in driving SysTick timer are mentioned below:

1. **SysTickPeriodSet** - It sets the period of the SysTick counter.
   Prototype: void SysTickPeriodSet(uint32_t ui32Period)
   Parameters: ***ui32Period*** is the number of clock ticks in each period of the SysTick counter and must be between 1 and 16,777,216,[6] inclusive.

[6]The reason for the range of Systick timer Period is 1 to 16,777,216 because SysTick timer is 24-bit.

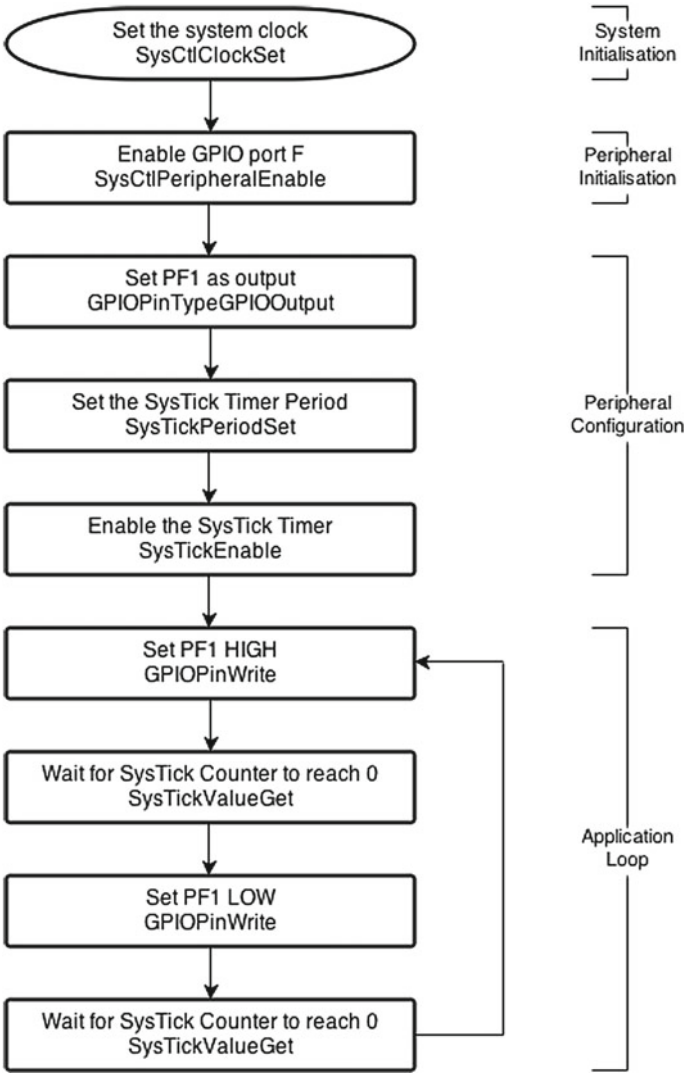

Fig. 4 Program flow for SysTick timer blinky

Description: This function sets the load value of the SysTick down counter, lower the value faster the Systick counter wraps to 0.
Returns: None.

2. **SysTickEnable** - It enables the SysTick counter.
   Prototype: void SysTickEnable(void)
   Parameters: None.
   Description: This function starts the Systick timer, if the SysTick interrupts are

enabled and their handlers are registered then it will the handlers at time-out conditions.
Returns: None.

3. **SysTickValueGet** - It returns the current value of the SysTick counter.
Prototype: uint32_t SysTickValueGet(void)
Parameters: None.
Description: This function returns the current value of the register which is between −1 and 0.
Returns: Current value of the SysTick counter.

7 Experiment 16—Obstacle Sensor

7.1 Objective

Detect the close range objects using IR LED and TSOP receiver and use the buzzer to indicate the presence of obstacle.

7.2 Hardware Description

This experiment requires IR LED, IR TSOP receiver and buzzer. This TSOP receiver is generally used in TV remotes (on signal detection side). IR LED is connected to PA7 through a jumper, JP4 and output of TSOP receiver is connected to PA6 through jumper JP3. Hence, make necessary jumper connections to connect IR LED and TSOP receiver. The block diagram for the experiment is shown in Fig. 5.

7.3 Experiment Tips

IR TSOP receiver is different, as it needs the incoming data to be modulated and it generates the trains of pulses as demodulated data. It generates a logic high when carrier frequency of received data (or IR Light) is not matching carrier frequency for TSOP is designed for. When 38 KHz (if TSOP receiver is of 38 KHz) of square pulses is generated at the IR transmitting end, it gives logic low. Since the environment around us has a lot more IR content depending on, so there is a need to develop a protocol to eliminate the noise due to IR content, so there are different pulse coding techniques. These pulse coding techniques are Bi Phase Coding, Pulse Distance Coding, and Pulse Length Coding. So to perform this experiment, use any of the above-mentioned coding techniques to send a fixed data let us say data is 12-bit (0×ABC). Now, the obstacle is placed in front of LED it will reflect the IR light and

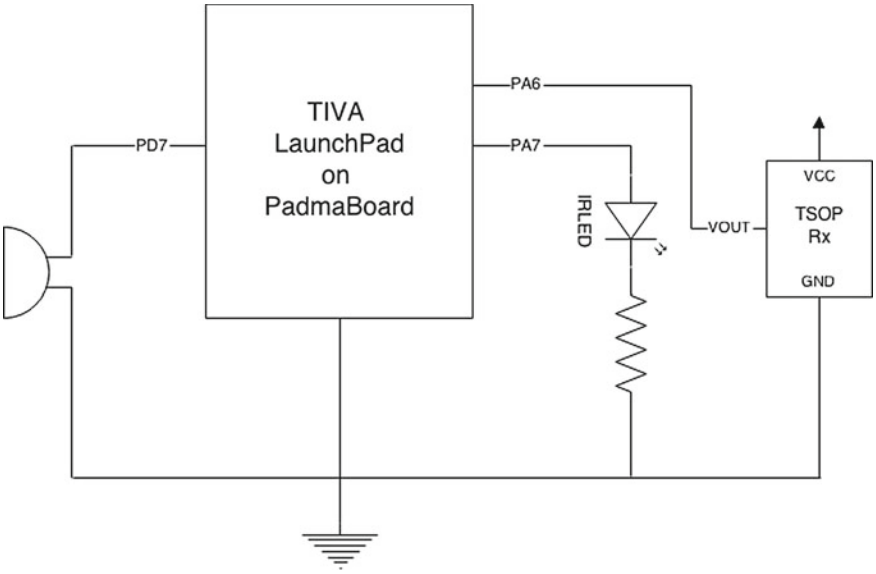

Fig. 5 Block diagram for obstacle sensor

will be received by the TSOP which is placed beside the IR LED. Observe what data it will receive. It must be same as the data transmitted by the IR LED. When the obstacle is detected, use buzzer to indicate the presence of the obstacle.

8 Experiment 17—Remote Control

8.1 Objective

Use the TV remote to toggle the state of LED on the Tiva LaunchPad or LED on PadmaBoard.

8.2 Hardware Description

Since the TV remote is the IR source, so it requires the TSOP receiver and the LED which is present on the Tiva LaunchPad. The block diagram for the experiment is explained in Fig. 6. TSOP (IR receiver) is connected to pin PA6 through a jumper JP3. And, LED is connected to PF1.

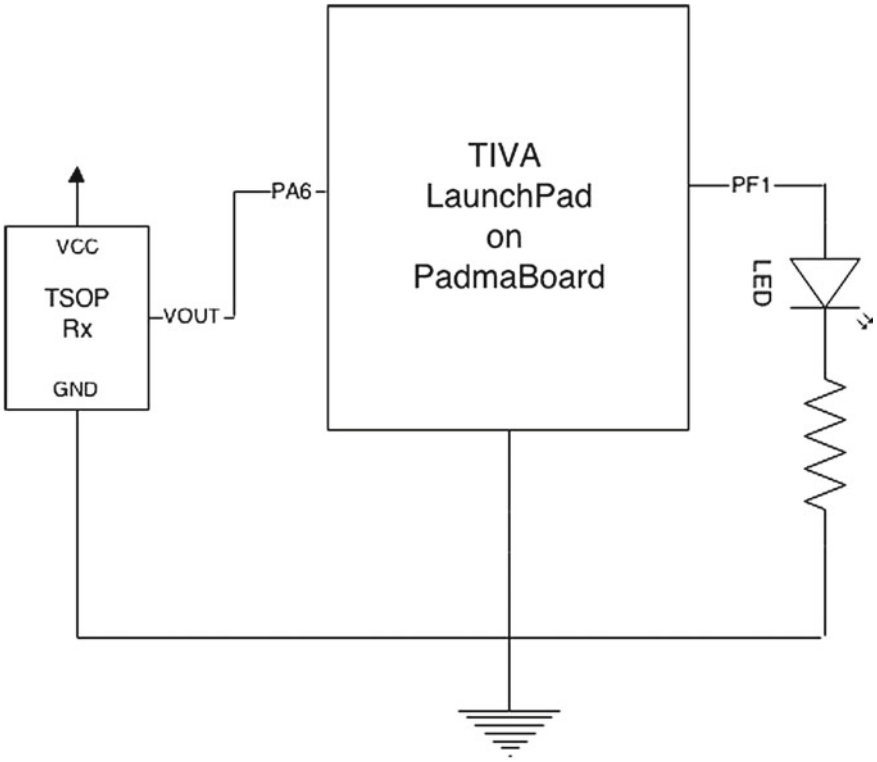

Fig. 6 Block diagram for remote control

8.3 Experiment Tips

Since in this experiment, only decoding needs to done as TV remote will generate the modulated pulses. Therefore, the decoding technique used will depend on the remote manufacturer.

9 Experiment 18—IrDA

9.1 Objective

Perform the wireless IR data transmission, between the two Tiva LaunchPads. Use the switch of LaunchPad 1 to toggle the state of LED on LaunchPad 2.

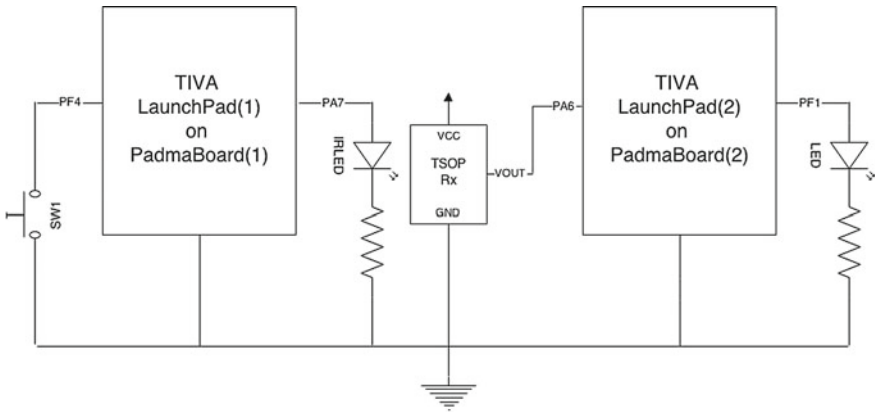

Fig. 7 Block diagram for IrDA

9.2 Hardware Description

To perform the experiment it would require two launchpads and one IR LED on the transmitting end and one TSOP on receiving end. The block diagram to perform such experiment is shown in Fig. 7. It also requires a switch on the transmitting end and a LED on the receiving end. Both switch and LED are available on the Tiva LaunchPad.

9.3 Experiment Tips

This communication can be only one directional that is both LaunchPads cannot transmit at the same time. If they transmit at the same time then it will corrupt the data received by the TSOP.

10 Experiment 19—Watchdog Timer

10.1 Objective

Configure the watchdog timer and reset the system when it overflows. Reset the watchdog timer and disable the interrupts when the switch is pressed turn on the RGB LED for visual indication.

10.2 Hardware Description

To perform this experiment, a switch and a RGB LED are required, which already have discussed in previous experiments. And watchdog timer is a internal peripheral of the microcontroller and no external hardware is required to implement it.

Chapter 12
Universal Asynchronous Receiver and Transmitter (UART)

Data transfer between two devices can be in parallel communication or serial communication. In parallel communication, multiple bits are transferred at each clock event whereas in serial communication, single bit is transferred at each clock event. In parallel communication, the number of lines determines the number of bit that can be transferred simultaneously. Hence, for an 8-bit parallel communication, there will be 8 data lines and few more line(s) for handshaking signals.[1] But in serial communication data is transferred bit by bit serially on the same line so the number of lines for data communication will be less as compared to the parallel communication. But speed of operation in serial communication will be less as compared to parallel operation. For communication between two boards, serial communication is preferred as it requires less number of data lines to connect from one board to another. There are several serial communication protocols like UART (Universal Asynchronous Receiver and Transmitter), SPI (Serial Peripheral Interface), or I2C (Inter-Integrated Circuit). This chapter will cover the UART peripheral. SPI and I2C will be covered later in the manual.

1 Modes of Serial Communication

The serial communication is further divided on the basis of the direction of flow of data on communication line.

1. **Simplex**: In this type of communication, direction of flow of data is in one direction only, i.e., data can be either transmitted or received. For example, like

[1]Handshaking signals are required to perform data communication with more synchronization. It generally contains signals like new data is available on bus, receiver is ready to receive the data, or acknowledgment when the data is received.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_12

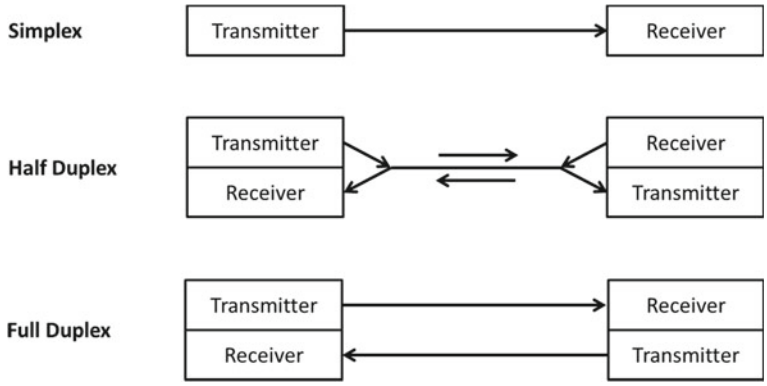

Fig. 1 Serial modes

communication between printer and host PC where data is transmitted from host PC to printer not in vice versa manner.

2. **Half Duplex**: In this type of communication, data can be either transmitted or received over the communication line. But at an instant of time data will either transmitting or receiving by the communication end. For example, walkie-talkies where data can be transmitted from any one walkie to another but at one time only can transmit and one can receive.
3. **Full Duplex**: In this type of communication, data can be simultaneously transmitted and received over the communication line. For example, like telephone. Figure 1 shows the above-mentioned modes of communication.

2 Functional Description

2.1 Data Framing

UART is the asynchronous serial communication. In asynchronous serial communication, start bit is sent prior to the data bits. This bit indicates the start of the data flow between the two communication ends. Hence, it indicates the receiving end to get prepared to receive the data. Along with data, parity bits can be transmitted which will confirm that the data has been transmitted correctly over the channel. Then, after transmitting data and parity bits, stop bit will be generated which will indicate the end of data transmission. Every data byte will be wrapped in such described manner. So, with every byte of data, some additional bits are also transmitted. This overhead

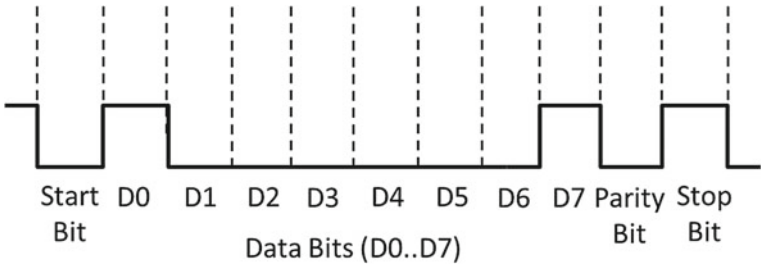

Fig. 2 Sample of data frame

transmission of bits is rectified when the data is transmitted through synchronous serial communication (Fig. 2).[2]

The start bit is always one bit whereas stop bit can be one or two bits. So, first, the start bit is sent which is logic low ("0").Then the data bits are serially streamed over the channel, LSB (Least Significant Bit) is first bit to be sent followed next LSB upto the MSB (Most Significant Bit). The data bits can be 5, 6, 7, or 8. After sending the data bits, optional parity bit (odd parity bit or even parity bit) can be sent. After transmitting parity bit, stop bit(s) are sent.

2.2 UART Peripheral Features

The TM4C123GH6PM controller which is used on the Tiva LaunchPad has eight UARTs with the below-listed features:

- Programmable baud rate generator
- Separate 16 × 8 transmit (TX) and receive (RX) FIFOs
- Standard asynchronous communication bits for start, stop, and parity
- Programmable serial interface characteristics
  - 5, 6, 7, or 8 data bits
  - Even, odd, stick, or no-parity bit generation/detection
  - 1 or 2 stop bit generation
- End-of-Transmission interrupts
- Modem flow control
- Efficient transfers using Micro-Direct Memory Access Controller (DMA)
- Support IrDA serial-IR (SIR) encoder and decoder functions

[2]In synchronous serial communication, along with the data, clock is also transmitted. The data is sampled by the receiving end in sync with the received clock. Hence, no extra start or stop bits need to be transmitted.

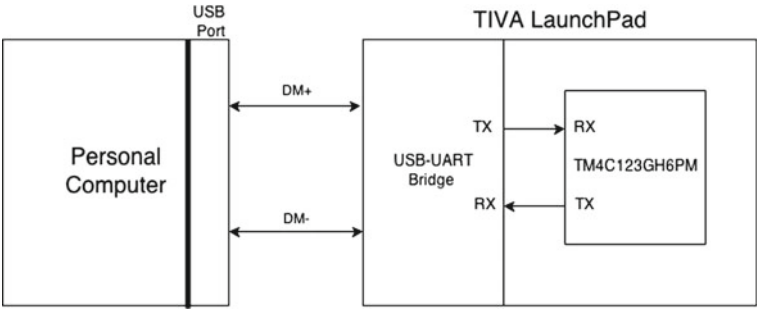

Fig. 3 Block diagram for UART Echo

3 Experiment 20—UART Echo

3.1 Objective

Receive a data byte over the UART, increment it by one and transmit it back. Configure the UART module at 115200 kbps baud rate, 8 data bits, none parity bits, and 1 stop bit.

3.2 Hardware Description

To communicate between personal computer and microcontroller through UART, USB to UART bridge is required (since USB are much common these days). USB to UART bridge converts the differential data of USB port to the UART data. As mentioned earlier on Tiva LaunchPad, there are two microcontroller target and debug microcontroller. On Tiva LaunchPad for UART module 0, the debugger microcontroller acts as USB to UART bridge. The output of UART can be observed on the personal computer on terminals like Putty, Tera Term, etc. There is a need to specify the UART configuration parameters same as specified in the program code for target microcontroller. Figure 3 shows the block diagram of experiment.

3.3 Program Flow

In experiment, UART data is transmitted from the terminal on host computer to target microcontroller. Microcontroller reads the UART data increment it by 1 and transfers it back to the terminal. So, if data “A” is sent from the terminal, the reply should be “B”. Program flow for the experiment is enumerated below:

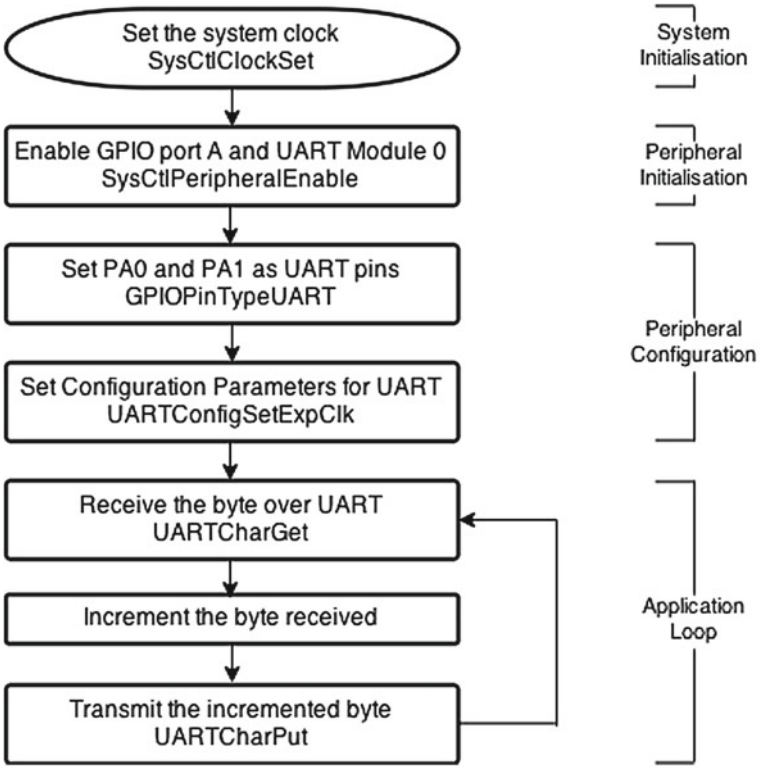

Fig. 4 Program flow for UART Echo

1. Enable the system clock and GPIO Port A.
2. Enable the UART Module 0.
3. Configure the UART Module 0 for 115200 baud rate, 8 data bits, none parity bits, and 1 stop bit.
4. Check whether the data byte is available at UART or not. If it is available then read UART data byte.
5. Increment the data byte by 1 and send it back to terminal. And, again wait for next data byte.

Figure 4 shows the program flow for the experiment.

3.4 Useful API Function Calls

Mentioned below are some API calls necessary to perform experiments involving UART peripheral.

1. **GPIOPinTypeUART** - It configures the pin(s) to use by the UART peripheral.
   Prototype: void GPIOPinTypeUART(uint32_t ui32Port, uint8_t ui8Pins)
   Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s).
   Description: This function configures pin(s) to be used as UART pins only if they are muxed with the UART peripheral.
   Returns: None.
2. **UARTConfigSetExpClk** - This function sets the configuration parameters for UART.
   Prototype: void UARTConfigSetExpClk(uint32_t ui32Base, uint32_t ui32UART Clk, uint32_t ui32Baud, uint32_t ui32Config)
   Parameters: ***ui32Base*** is the base address of the UART port. ***ui32UARTClk*** is the clock supplied to the UART. ***ui32Baud*** is the baud rate. ***ui32Config*** is the data format(number of data bits, number of stop bits and parity).
   Description: This function configures the UART module specified by the parameter ***ui32Base***. The baud rate is specified by the parameter ***ui32Baud*** and configuration parameters by ***ui32Config***.
   The ***ui32Config*** parameter is the logical OR of three values: the number of data bits, the number of stop bits, and the parity. **UART_CONFIG_WLEN_8, UART_CONFIG_WLEN_7**, **UART_CONFIG_WLEN_6,** and **UART_CONFIG_WLEN_5** select from eight to five data bits per byte (respectively). **UART_CONFIG_STOP_ONE** and **UART_CONFIG_STOP_TWO** select one or two stop bits (respectively). **UART_CONFIG_PAR_NONE, UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD, UART_CONFIG_PAR_ONE,** and **UART_CONFIG_PAR_ZERO** select the parity mode (no-parity bit, even parity bit, odd parity bit, parity bit always one, and parity bit always zero, respectively).
   Returns: None.
3. **UARTCharsAvail** - Determines whether any character is available in the UART buffer.
   Prototype: bool UARTCharsAvail(uint32_t ui32Base)
   Parameters: ***ui32Base*** is the base address of the UART port.
   Description: This function returns whether the data is available on the UART buffer or not.
   Returns: Returns **True** if data is available and returns **false** if data is not available.
4. **UARTCharGet** - Reads the character from the specified UART port.
   Prototype: int32_t UARTCharGet(uint32_t ui32Base)
   Parameters: ***ui32Base*** is the base address of the UART port.
   Description: This function reads the character from the specified port if the character is not available in receive buffer of UART it will wait until character is received before returning.
   Returns: The character reads from the specified UART port, cast as int32_t.
5. **UARTCharPut** - This function sends a character to the specified UART port.
   Prototype: void UARTCharPut(uint32_t ui32Base, unsigned char ucData)
   Parameters: ***ui32Base*** is the base address of the UART port. ***ucData*** is the character to be transmitted.

Description: This function sends the character ***ucData*** to the transmit buffer of the UART. If buffer is already full it will wait until buffer is empty and then it places the data in the buffer before returning.
Returns: None.

3.5 Program Code

The complete C program is given below. The code is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Defines and Macros for UART API */
#include "driverlib/uart.h"

int main(void)
{
/* Set the clock to directly run from the crystal at 16MHz */
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/*Enable the GPIO Port A*/
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

/*Enable the peripheral UART Module 0*/
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

/* Make the UART pins be peripheral controlled. */
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

/* Sets the configuration of a UART. */
UARTConfigSetExpClk(UART0_BASE, 16000000, 115200,
(UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

while(1)
  {
  if(UARTCharsAvail(UART0_BASE))
    {
    /* Transmit the incremented received unsigned character */
    UARTCharPut(UART0_BASE,(unsigned char)(UARTCharGet(UART0_BASE)+1));
```

```
        }
    }
}
```

4 Experiment 21—Bluetooth Control

4.1 Objective

Interface Tiva LaunchPad with Bluetooth module and control the state of LED on Tiva LaunchPad using mobile phone (or laptop) through bluetooth.

4.2 Hardware Description

This experiment requires Bluetooth Module and a LED. On PadmaBoard, there is a connector for Bluetooth Module, Fig. 5 shows the placement of the Bluetooth module on the PadmaBoard.

Since Bluetooth Module is based on UART serial communication, UART Module 1 with UART pins PB0 and PB1 are connected to the Bluetooth Module connector. This connector is fully compatible with Bluetooth Module HC-05. By default module's Bluetooth name is "HC-05" and password is "1234", it transfers the data with baud rate of 9600 kbps. These parameters can be reconfigured by using the KEY pin of the bluetooth. This KEY has been connected to PD6. The red LED (connected to PF1) of the RGB LED is controlled using bluetooth. Most importantly, a bluetooth device with serial terminal is required to control the state of LED. If LED is controlled with Laptop using bluetooth, use the terminals same as that used for UART interfacing with the configuration setting matched with the Bluetooth Module. For mobile phones, bluetooth terminals are easily available. Configure those terminals with settings same as Bluetooth Module. Figure 6 shows the block diagram for the experiment.

4.3 Program Flow

In this experiment, LED is controlled through Bluetooth device. By default state of LED is "on" when the character "0" (ASCII value is 48) is sent from bluetooth device to Tiva LaunchPad, the LED will turn "OFF". And Tiva LaunchPad transmits "OFF" string over the bluetooth to bluetooth device. If any other character is sent from bluetooth device to Tiva LaunchPad then, LED will turn "ON" and Tiva LaunchPad transmits string "ON" over the bluetooth to bluetooth device.

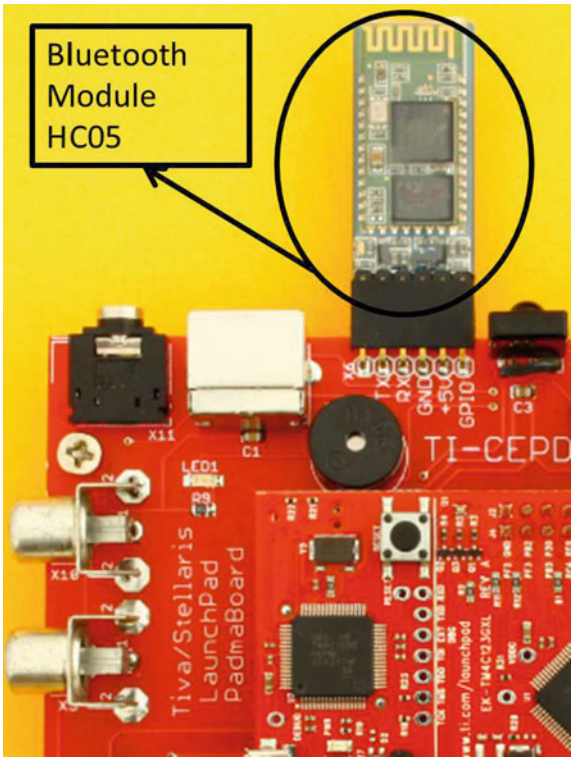

Fig. 5 Placement of UART module

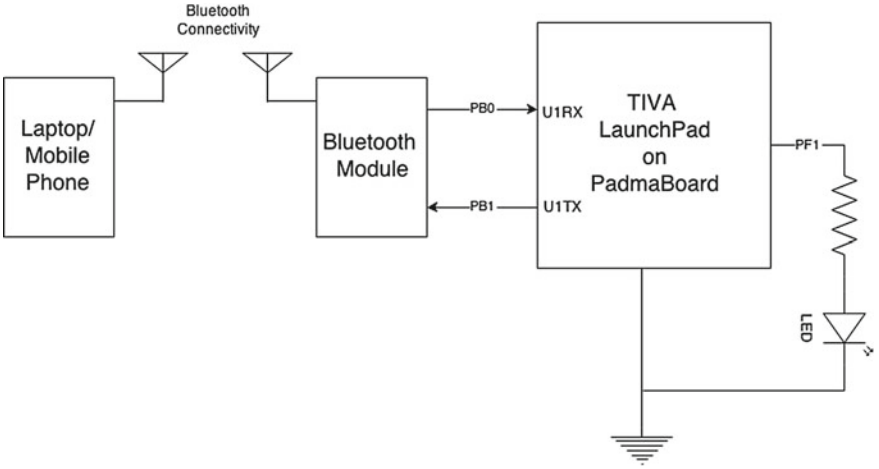

Fig. 6 Block diagram for Bluetooth control

1. Enable the System Clock, GPIO Port B and GPIO Port F.
2. Enable the UART Module 1.
3. Configure the pins PB0 and PB1 as UART pins.
4. Configure PF1 as output and initialize PF1 to logic high.
5. Pull up the UART pins PB0 and PB1.
6. Configure the UART Module 1 with parameters matching to Bluetooth Module.
7. Wait till data is available on the UART buffer.
8. Read the data byte from the UART Buffer. If its "0" turn off the LED and echo back "OFF" string (indicating the current state of LED on the controlling terminal).
9. For any other character, turn on the LED and echo back "ON". And again wait for the character in UART buffer.

Figure 7 shows the program flow of the experiment.

4.4 Program Code

The complete C program is given below. The code is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"
#include "inc/hw_gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Defines and Macros for UART API */
#include "driverlib/uart.h"

/* Defines the Port/Pin Mapping */
#include "driverlib/pin_map.h"

unsigned char a;
int main(void)
{
/* Set the clock to directly run from the crystal at 16MHz */
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Enable the GPIO Port B, GPIO Port F and UART Module 1 */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);

/* Configure PB0 and PB1 as UART RX/TX pins */
GPIOPinConfigure(GPIO_PB0_U1RX);
GPIOPinConfigure(GPIO_PB1_U1TX);

/* Make the UART pins be peripheral controlled */
GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);

/* Configure PF1 as output and initialize it to logic high */
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_1);
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,GPIO_PIN_1);

/* Pull up the PB0 and PB1 */
GPIOPadConfigSet(GPIO_PORTB_BASE, GPIO_PIN_0,
GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);

/* Sets the configuration of a UART */
UARTConfigSetExpClk(UART1_BASE,16000000, 9600,
(UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

while(1)
  {
  while(!UARTCharsAvail(UART1_BASE));
  a=UARTCharGet(UART1_BASE);
  if(a == 48)
    {
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,0);
    UARTCharPut(UART1_BASE,79);//O
    UARTCharPut(UART1_BASE,70);//F
    UARTCharPut(UART1_BASE,70);//F
    UARTCharPut(UART1_BASE,9);//horizontal tab
    }
  else
    {
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,GPIO_PIN_1);
    UARTCharPut(UART1_BASE,79);//O
    UARTCharPut(UART1_BASE,78);//N
    UARTCharPut(UART1_BASE,9);//horizontal tab
    }
  }
}
```

5 Experiment 22—UART Intensity Control

5.1 Objective

Control the intensity of LED, connected to PF1 through the parameter input from UART using software PWM. Input parameter may vary from 0 to 9, with 0 as input LED is at minimum intensity and if input is 9, LED is at maximum intensity.

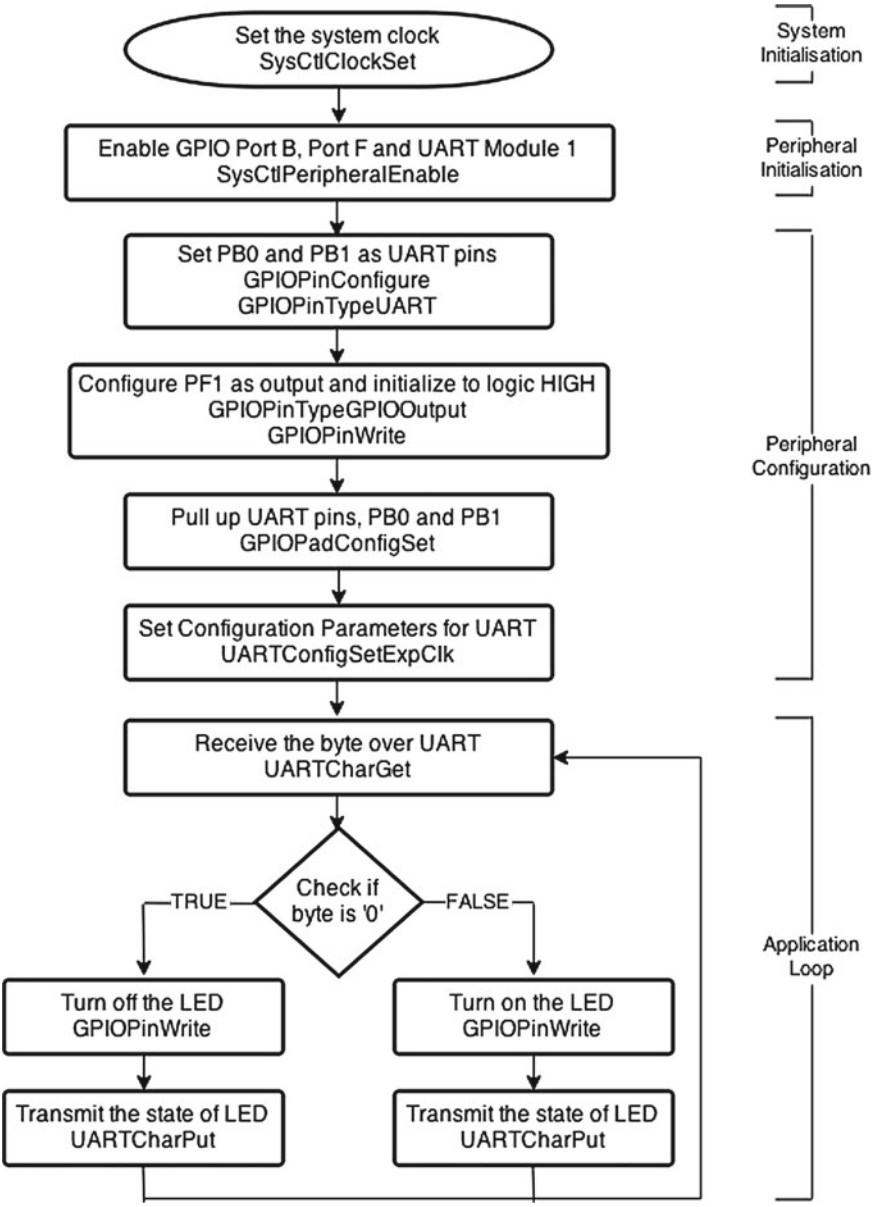

Fig. 7 Block diagram for Bluetooth control

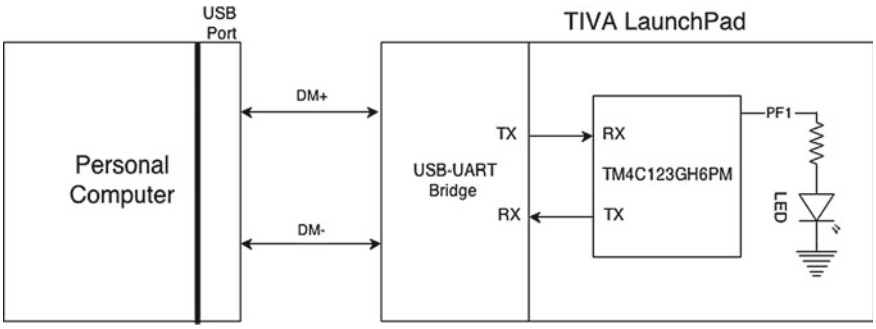

Fig. 8 Block diagram for UART intensity control

5.2 Hardware Description

This experiment requires UART terminal on the personal computer to input intensity value and LED which is there on Tiva LaunchPad. The block diagram for the experiment is shown in Fig. 8.

5.3 Program Flow

In this experiment, intensity of LED or the duty cycle of software PWM at PF1 is controlled through parameter input over UART. Below is enumerated program flow for the experiment:

1. Enable the system clock, GPIO Port A and GPIO Port F.
2. Enable the UART Module 0.
3. Set the directions for GPIO Port A and GPIO Port F.
4. Configure the UART Module 0 for 115200 baud rate, 8 data bits, none parity bits, and 1 stop bit.
5. Receive the parameter over UART.
6. Depending on the magnitude of the input parameter over UART, adjust the duty cycle of the software PWM at PF1.
7. Continue until a new parameter is used.

The program flow for the experiment is explained in Fig. 9.

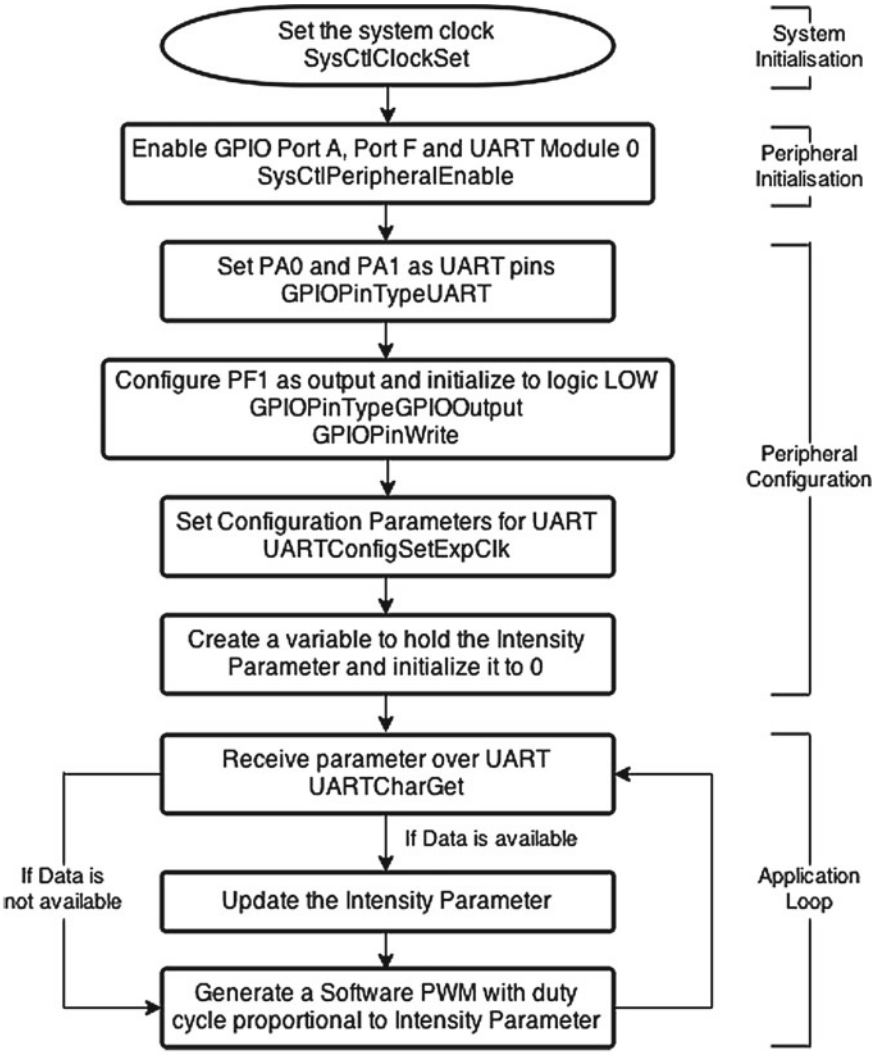

Fig. 9 Program flow for UART intensity control

6 Experiment 23—Color Generator

6.1 Objective

Generate the color using the RGB LED on the Tiva LaunchPad with red, green, and blue color component values are taken from personal computer over UART.

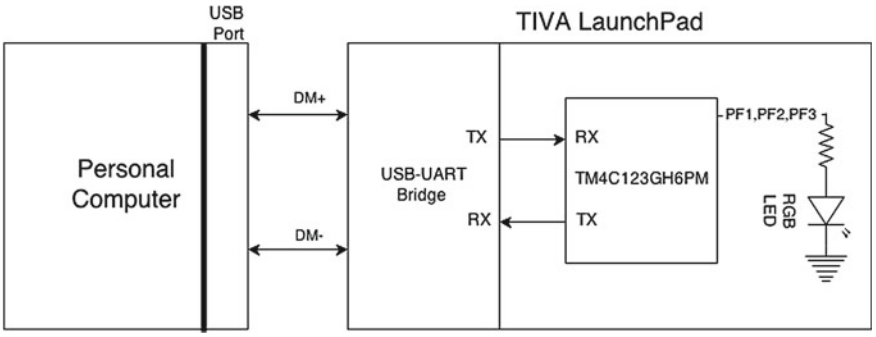

Fig. 10 Block diagram for color generator

6.2 Hardware Description

This experiment requires a RGB LED and terminal on personal computer to input values of red, green, and blue color components. Instead terminal Graphical User Interface can be developed, which allows the user to enter the color components and then transfer them to Tiva LaunchPad over UART. Block diagram for the experiment is shown in Fig. 10.

6.3 Experiment Tips

Instead of using software PWM to control the three LEDs, use hardware PWM. Generate 8-bit PWM for each of the three LEDs, so the number of colors that can be generated is (256*256*256) 16777216. But it may not be possible to observe each color separately with naked eye.

7 Experiment 24—Ultrasonic Ranger

7.1 Objective

Measure the distance of the object using Ultrasonic Module and display it on personal computer over UART terminal.

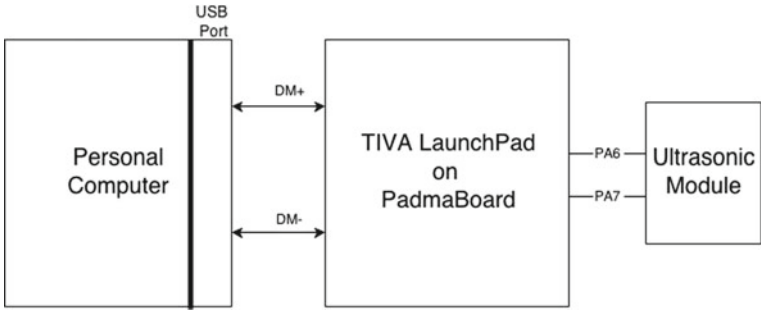

Fig. 11 Block diagram for ultrasonic ranger

7.2 Hardware Description

This experiment requires Ultrasonic Module and UART terminal on personal computer. The block diagram for experiment is shown in Fig. 11. Place the appropriate jumpers on PadmaBoard to select the connection between Ultrasonic Module Connector and Tiva LaunchPad. Jumper selection on PadmaBoard is explained in Chap. 4 under Sect. 5

8 Experiment 25—RS232 Communication

8.1 Objective

Perform the RS232 communication by echoing data between two Tiva LaunchPads.

8.2 Hardware Description

Connect RS232 module on the connector X6 of PadmaBoard, which has UART pins. Similarly, connect RS232 module on other PadmaBoard. Now connect these two RS232 connectors through an RS232 cable.

9 Experiment 26—RS485 Communication

9.1 Objective

Perform the RS485 communication by echoing data between multiple Tiva Launch-Pads.

9.2 Hardware Description

Connect RS485 modules on connector X6 (which has UART pins) of multiple PadmaBoards. Connect these RS485 ports between multiple PadmaBoards through wires.

Chapter 13
System Control and Power Management

This chapter deals with power management operations using System Control module. System Control module provides system control and implementation information. Power management is one of the main aspects of design. The product should be energy efficient and must consume minimal power. Example in smoke detector, controller may not need to be active all the time, it should be active only when smoke is detected. Hence, controller should consume very little power in its inactive state. Such tasks can be achieved through power management operations in Tiva C series family.

1 System Control

System Control configures the overall operation of device. It provides information about the device and includes configurable features like reset control, clock control, power control, and low-power modes. The system control has below enumerated capabilities:

1. Device Identification
2. Reset Control, Power Control, and Clock Control
3. Modes of Operation (Run, Sleep, Deep Sleep, and Hibernate modes)

These capabilities mentioned above are described in subsequent sections.

1.1 Device Identification

There are several read-only registers which provide the information for the device identification like version, part number, memory sizes and peripherals present on the

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_13

device. There are Devide Identification registers (DID0 and DID1) which provide information about the device version, part number, pin count, package (0×1 for LQFP package and 0×2 BGA package), temperature range in which the device will work and family of the device. The information about the peripherals present or how many modules of each type is present, is available in Peripheral Present registers like Watchdog Timer Peripheral Present (PPWD) register. The information about properties for each of the peripheral, is present in the peripheral properties register like for module GPTM, the information about the properties will be available in GPTM Peripheral Properties (GPTMPP) register.

1.2 Reset Control

There are various sources of reset for a device. The reset can be both hardware or software in nature.

- **Power-On Reset (POR)**: The internal circuitry for Power-On reset monitors the supply voltage (or V_{DD} signal). When this voltage reaches a certain threshold value. This circuitry generates a reset signal for the device.
- **External Reset Pin**: This pin has internal filter and it requires a minimum pulse width of T_{MIN} which can be determined from the controller datasheet. The filter is present to remove high-frequency noise at external reset pin. The reset is active low configuration if internal POR circuitry is present. Hence, an external pull up resistor is present at reset pin, and this pin is driven to logic low through button or a switch.
- **Brown-Out Reset**: Microcontroller provides the brown-out detection circuit, this circuitry triggers reset when the voltage supply falls below the brown-out detection voltage. The cause for the reset is stored in Reset Cause (RESC) register. So the application running on microcontroller can detect the occurrence of brown-out event from reading the register RESC.
- **Software Reset**: Software reset can be used to reset a particular peripheral or the entire microcontroller.
- **Watchdog Timer Reset**: Watchdog Timer prevent the system to hang and generates a reset when counter hits the zero value. If system is not hanged up watchdog timer can be reloaded before it hits zero.

1.3 Clock Control

The System Control also influences the clock control of the device. There are various clock sources available for microcontroller. Among those, two major clock sources are discussed below:

- **Precision Internal Oscillator (PIOSC)**: It is an on-chip clock source. It does not require external components and provide a clock of $16MHz +/- 1\%$ at room temperature and $+/-3\%$ across temperature range. Even if the PIOSC is not configured to be used for system clock it can be used as source for ADC, UART, and SSI clocks.
- **Main Oscillator (MOSC)**: This is the external clock source which can be single ended clock source connected to OSC0 pin or can be external crystal connected between OSC0 and OSC1. If PLL[1] is used crystal value should range from 5 to 25 MHz and if PLL is not used crystal value ranges from 4 to 25 MHz.

1.4 Modes of Operation

Tiva C series family of microcontrollers can be operated in various modes as enumerated below:

- **Run Mode**: In Run Mode, microcontroller performs the normal execution of code. In this, all peripherals that are enabled and processor performs the normal operation. The clock source can be any of available clock source including PLL.
- **Sleep Mode**: In Sleep Mode, clock frequency for the active peripherals remains same but memory and processor are not clocked. Hence, no longer code runs in the sleep mode. The microcontroller can be brought back to the Run Mode with the help properly configured interrupt.
- **Deep Sleep Mode**: In Deep Sleep Mode, clock frequency of the active peripherals may change depending upon the Run Mode clock configuration, in addition to memory and processor clock being stopped. Again, the microcontroller can be brought back to the Run Mode with the help of properly configured interrupt.
- **Hibernate Mode**: In hibernation, power supplies to the main part of microcontroller is turned off, only the Hibernation Module circuitry is powered. Hence, reducing the power consumption to minimal. The power can be restored based on the external signal or using built in real time clock (RTC). The Hibernation module can be powered independently, by using external battery or auxiliary power supply.

[1]Phase Locked Loop (PLL) is used to generate multiple clock frequencies using single clock frequency which is used as a reference by PLL module. In this, the output signal is divided down using frequency divider by the multiplication factor, then the divided signal compared with the reference clock signal and a voltage is generated corresponding to the phase difference between the two. The voltage is used to drive the Voltage Control Oscillator (VCO) which increases or decreases the frequency of output signal as per voltage input. If PLL operation is observed from the view of control systems, frequency divider will be present in the feedback path and VCO will be present in forward path.

2 Experiment 27—PLL

2.1 Objective

Use PLL functionality of microcontroller to generate a system clock of 80 MHz. And display the system clock on UART terminal.

2.2 Hardware Description

Since, PLL module is present inside the microcontroller, so no external device is required to perform the PLL operation. UART Terminal is required to display the system clock frequency. The block diagram of experiment is same as described in Experiment20-(UART Echo).

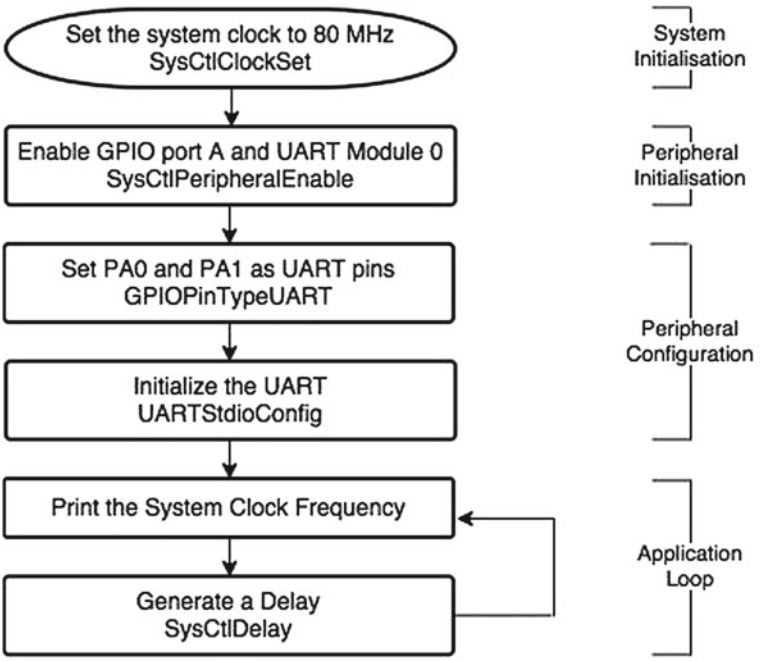

Fig. 1 Program flow for PLL

2.3 Program Flow

Configure the system clock to 80 MHz and print the same on the UART Terminal. In this experiment UART library from "utils" folder is used. This library is much more user friendly like instead of printing byte by byte on the terminal it has function "printf" to transmit string of characters over UART to terminal on host PC. If using Code Composer Studio, then copy uartstdio.c from "C:\ti\TivaWare_C_Series-2.1.1.71\utils" to the project folder in workspace area. The program flow is shown in Fig. 1 and also enumerated below:

1. Set the system clock to 80 MHz using PLL operation.
2. Configure the UART.
3. Keep printing the system clock value on the terminal with a delay of 1 second.

2.4 Program Code

The complete C program is given below. The code is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Defines and Macros for UART API */
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

void InitConsole(void)
{
/* Initializing GPIO Port A and UART Module 0 */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

/* Make the UART pins be peripheral controlled */
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

/* Initialize the UART for console I/O */
UARTStdioConfig(0, 115200, SysCtlClockGet());
}

int main(void)
{
/* Set the clock to run at 80Mhz from the crystal of 16MHz using PLL */
```

```
SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

/* UART configuration */
InitConsole();

while(1)
  {
  UARTprintf("Clock Frequency::");
  UARTprintf("%04d\n", SysCtlClockGet());
  SysCtlDelay(SysCtlClockGet()/3);
  }
}
```

3 Experiment 28—Runtime PLL

3.1 Objective

Modify the system clock frequency during the runtime of code. Start system clock with 66.66 MHz. Use switch SW1 connected to PF4 to change the system clock to 50 MHz and use the SW2 connected to PF0 to change the system clock to 80 MHz. And keep displaying system clock on UART Terminal present on host PC.

3.2 Hardware Description

To perform this experiment, two switches are required which are available on Tiva LaunchPad. These switches are connected to PF4 and PF0. Also, UART Terminal is required on host PC. The block diagram for experiment is illustrated in Fig. 2.

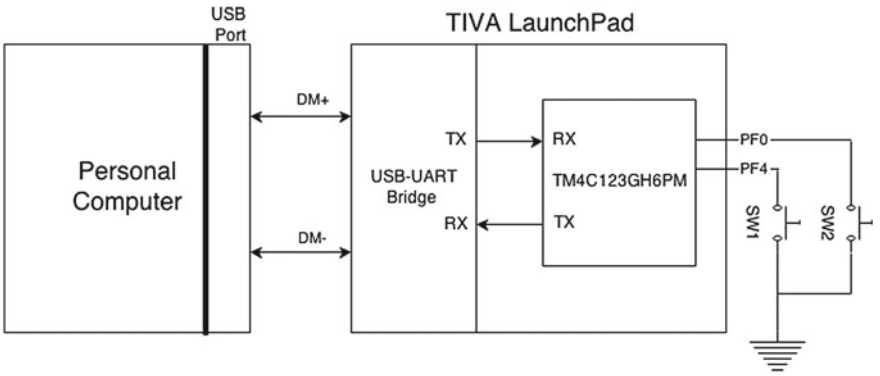

Fig. 2 Block diagram for Runtime PLL

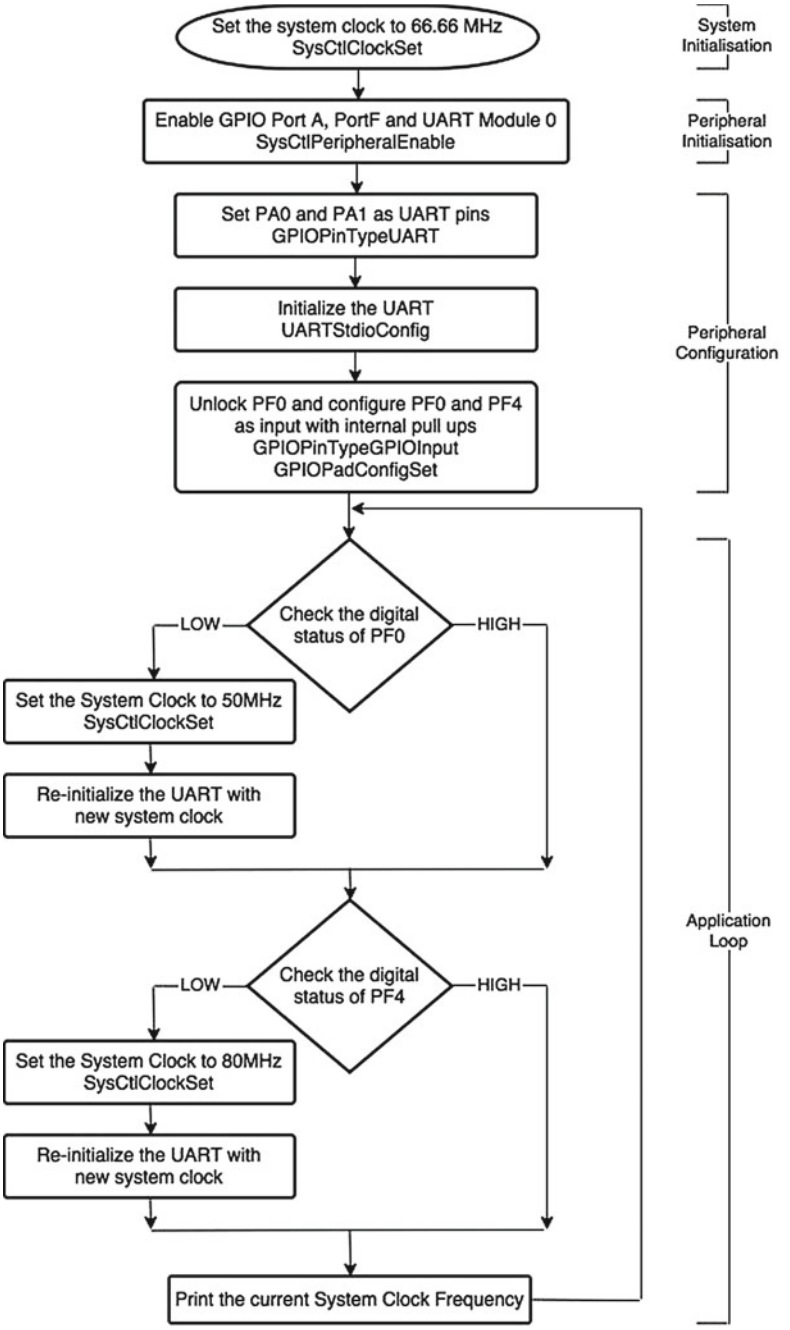

Fig. 3 Program flow for Runtime PLL

3.3 Program Flow

To perform this experiment, PF0 pin need to be unlocked as it is a NMI pin. In the beginning of program configure the system clock with 66.66 MHz, then while running the application if switch SW1 is pressed change the system clock to 50 MHz and if switch SW2 is pressed change the system clock to 80 MHz. Keep displaying the current system clock to make experiment look alive.

1. Set the system clock to 66.66 MHz.
2. Enable the GPIO Port F.
3. Unlock the NMI Pin PF0 and configure PF0 and PF4 as input pins.
4. Enable and configure the UART Module 0.
5. Check whether SW2 is pressed, if pressed change the system clock to 80 MHz.
6. Check whether SW1 is pressed, if pressed change the system clock to 50 MHz.
7. Print the current system clock. Repeat from step 5.

Program flow for the experiment is shown in Fig. 3.

3.4 Program Code

Mentioned below is the complete C program. The code is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "inc/hw_gpio.h"
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Defines and Macros for UART API */
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

#include "inc/tm4c123gh6pm.h"

void InitConsole(void)
{
/* Initializing GPIO Port A and UART Module 0 */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

/* Make the UART pins be peripheral controlled */
```

```
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

/* Initialize the UART for console I/O */
UARTStdioConfig(0, 115200, SysCtlClockGet());
}

int main(void)
{
/* Set the clock to run at 66.66Mhz from the crystal of 16MHz using PLL */
SysCtlClockSet(SYSCTL_SYSDIV_3 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Set the clock for the GPIO Port F */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

/* Unlocking the NMI Pin PF0 */
HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
HWREG(GPIO_PORTF_BASE + GPIO_O_CR) |= 0x01;
HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = 0;

/* Set the type of the GPIO Pin */
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_4|GPIO_PIN_0);
/* GPIO Pins 1 on PORT F initialized to 0 */
GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_4|GPIO_PIN_0,
GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);

/* UART config */
InitConsole();

while(1)
  {
  while(GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_0)==0)
    {
    SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL |
    SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
    InitConsole();
    }

  while(GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_4)==0)
    {
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
    SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
    InitConsole();
    }
  UARTprintf("\nClock Frequency::%04d\n",SysCtlClockGet());
  }
}
```

4 Experiment 29—Sleep Mode and Deep Sleep Mode

4.1 Objective

Perform various modes of operation by putting the processor into Sleep and Deep Sleep Modes.

4.2 Hardware Description

To perform this experiment, Tiva LaunchPad and a switch present on it is required. This switch is required to put the processor in sleep modes or deep sleep modes and also to bring back the processor to run mode. Block diagram of the experiment is similar to that of experiment 4 - Switchy.

4.3 Program Flow

In this experiment switch is used to generate the interrupt that put the processor in sleep and deep sleep modes. And, the same switch is used to generate the interrupt to put the processor back to run mode. The algorithm for the experiment is enumerated below:

1. Enable the system Clock and GPIO Port F Module.
2. Configure the PF4 as input with internal pull ups.
3. When switch is pressed, interrupt is generated. In ISR, disable the clock gating to Port F and enable the port for operation[2] in Sleep or Deep Sleep Mode. Put the processor in Sleep or Deep Sleep Mode.
4. When the switch is pressed again, software reset is generated to put the processor back in Run Mode. Figure 4 shows the block diagram of the experiment.

4.4 Useful API Function Calls

Mentioned below are the useful API Calls needed to perform the experiment.

1. **SysCtlPeripheralClockGating -** It controls peripheral clock gating in sleep and deep sleep mode.
   Prototype: void SysCtlPeripheralClockGating(bool bEnable)
   Parameters: *bEnable* is a boolean that is true if the sleep and deep sleep peripheral configuration should be used and false otherwise.
   Description: This function controls the clock of the peripherals when the processor is in Sleep or Deep Sleep Mode. By default, peripherals are clocked same as in run mode.
   Returns: None.
2. **SysCtlPeripheralSleepEnable -** It enables the peripheral in Sleep Mode.
   Prototype: void SysCtlPeripheralSleepEnable(uint32_t ui32Peripheral)
   Parameters: *ui32Peripheral* is the peripheral to enable in Sleep Mode.
   Description: This function allows the peripheral to continue operating when the

[2]Port F is enabled because the interrupt to bring back the processor will be generated from Port F only.

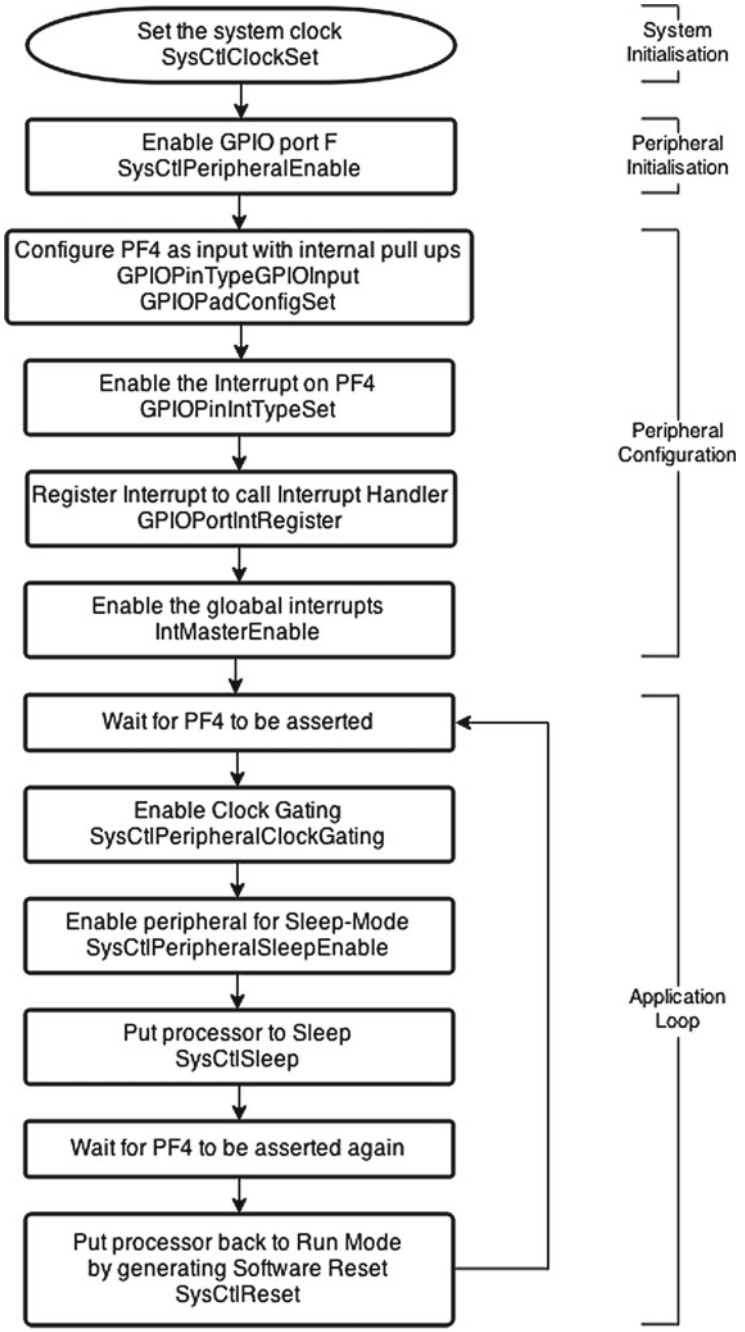

Fig. 4 Program flow for sleep mode experiment

processor goes into Sleep Mode.
Returns: None.

3. **SysCtlSleep -** It puts the processor in Sleep Mode.
   Prototype: void SysCtlSleep(void)
   Parameters: None.
   Description: This function puts the processor into Sleep Mode. It does not return until the processor returns to run mode. The peripherals that are enabled using SysCtlPeripheralSleepEnable() continue to operate and can wake up the processor.
   Returns: None.
4. **SysCtlPeripheralDeepSleepEnable -** It enables the peripheral in Deep Sleep Mode.
   Prototype: void SysCtlPeripheralDeepSleepEnable(uint32_t ui32Peripheral)
   Parameters: *ui32Peripheral* is the peripheral to enable in Deep Sleep Mode.
   Description: This function allows the peripheral to continue operating when the processor goes into Deep Sleep Mode.
   Returns: None.
5. **SysCtlDeepSleep -** It puts the processor in Deep Sleep Mode.
   Prototype: void SysCtlDeepSleep(void)
   Parameters: None.
   Description: This function puts the processor into Deep Sleep Mode. It does not return until the processor returns to run mode. The peripherals that are enabled using SysCtlPeripheralDeepSleepEnable() continue to operate and can wake up the processor.
   Returns: None.
6. **SysCtlReset -** It reset the device.
   Prototype: void SysCtlReset(void)
   Parameters: None.
   Description: This function generates the software reset which resets all the peripherals and the processor. However it does not reset the Reset Cause (RESC) register, which on later stages help us to determine the cause for the earlier occurred reset.
   Returns: None.

5 Experiment 30—RTOS

5.1 Objective

Understand the free rtos demo application, and create tasks and allocate priorities to them.

5.2 Experiment Tips

Free RTOS files can be found under "${TivaWare_C_Series-xxx}\third_party", go through them and try to create own tasks and assign priorities to them.

Chapter 14
Analog to Digital Converter (ADC)

This chapter covers the processing of analog data by microcontroller by using analog-to-digital converters (ADCs). The environment present around the microcontroller is analog but the processing of data in the microcontroller is done on the digital data. So, ADCs are required to convert continuous analog voltage to discrete digital number which can be realised in digital form. The microcontroller has built in ADC modules. Since the conversion of analog to digital data is done on the basis of quantization, so some error will be introduced in the conversion due to quantization. This error cannot be eliminated though it can be decreased by increasing the number of bits of digital output.

1 Introduction

On the PadmaBoard, there are various analog peripherals like temperature sensor using LM35 or thermistor, light sensor using LDR (Light Dependent Resistor), Hall effect sensor to sense the magnetic field and audio sensing through microphone or 3.5 mm audio jack. Generally, analog sensors are cheaper and widely available than the digital sensors. As digital sensors will sense the analog data and then change the data in digital form which can be transmitted in parallel or serial format to the master device. Hence, they have additional digital circuitry in addition to analog sensing part. That is why they are costlier than analog sensors. Microcontroller on Tiva LaunchPad, TM4C123GH6PM has two 12-bit ADC modules with various features as mentioned below:

- 12 analog input channels shared among both ADC modules.
- Single-ended and differential-input configurations.
- On-chip internal temperature sensor.
- Maximum sample rate of one million samples per second.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_14

- Four programmable sample conversion sequencers from one to eight entries long, with corresponding conversion result FIFOs.
- Flexibility in triggering the analog to digital conversion : Controller, Timers, Analog Comparators, PWM, and GPIO.
- Hardware average of up to 64 samples.
- Power and ground for the analog circuitry are separate from digital power and ground.

Since the controller has two ADC modules which share 12 Analog input channels. These modules use separate sequencers, generate different triggers or interrupts. Also, both ADC modules can be triggered at same time, so that two analog channels can be sampled at same time. This type of functionality is generally used for IQ sampling in communication signals.

2 Functional Description

Each ADC module has four sample sequencers. The sequencers allow sampling of 1, 4, 4, and 8 analog sources to be sampled. Note that there are two 4-step sequencers. Each sequencer has its own set of configuration registers. All steps in the sequencers can be mapped to analog channels (including internal temperature sensor), as single-ended or differential mode sampling. Different sequencers can be triggered by different sources. Sequencers have configurable priority when multiple sequencers are triggered simultaneously. Sequencer 0 has 8 steps, Sequencer 1 and Sequencer 2 has 4 steps, and Sequencer 3 has 1 step. On completing the sample sequence, conversion outputs are stored in FIFO for each sequencer. These FIFO can be retrieved by processor on the later stages.

Figure 1 shows the example of configuration of Sample Sequencer 1 and Sample Sequencer 3. In the example, to Sequencer 1, three analog channels (AIN0, AIN1, and AIN2) are mapped to the first three step of it. And, internal temperature sensor is mapped to Sample Sequencer 3, which has only 1 step. Sample Sequencer 1 is triggered by timer where as Sample Sequencer 3 is triggered by processor. As Sample Sequencer 1 still has one step remaining, if there is a need to trigger the internal temperature sensor with timer (instead of processor) it can be connected to the Sample Sequencer 1 (instead of Sample Sequencer 3).

3 Experiment 31—Thumbwheel

3.1 Objective

Read analog voltage variation through thumbwheel potentiometer on PadmaBoard and display the 12-bit corresponding ADC value on UART Terminal.

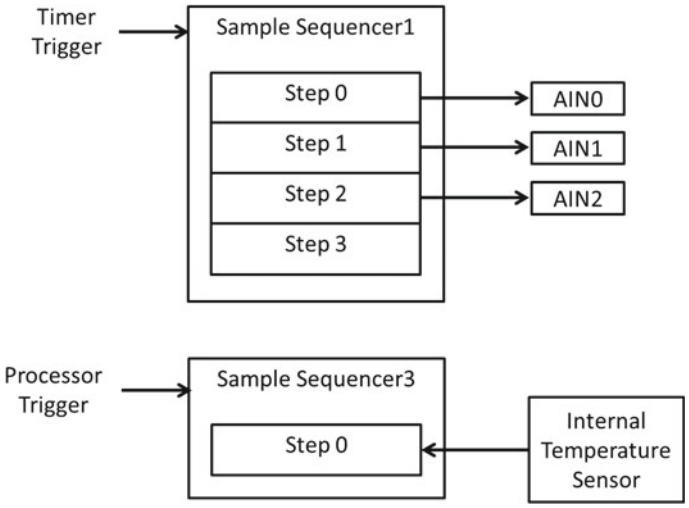

Fig. 1 Example of sample sequencer configuration

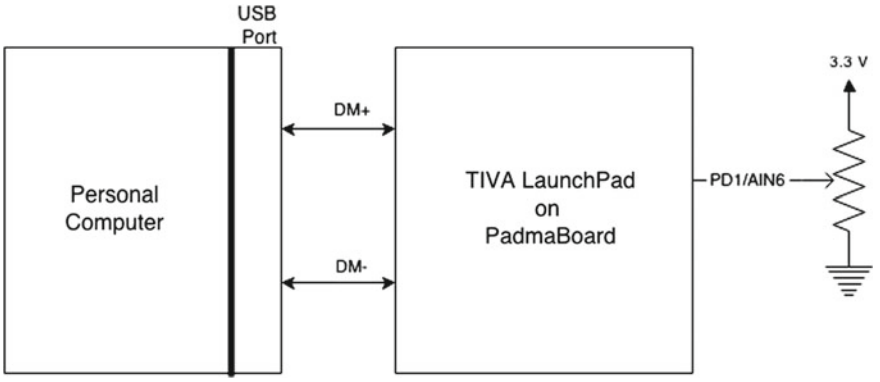

Fig. 2 Block diagram for thumbwheel

3.2 Hardware Description

This experiment requires a thumbwheel potentiometer and UART Terminal on personal computer to display the ADC values. The thumbwheel potentiometer is connected to Analog Channel 6 (AIN6) which is multiplexed with GPIO PD1. The connection of thumbwheel potentiometer to PD1 is through the jumper, JP1. For proper placement of jumper refer to Chap. 4 under Sect. 5. Figure 2 shows the block diagram of experiment.

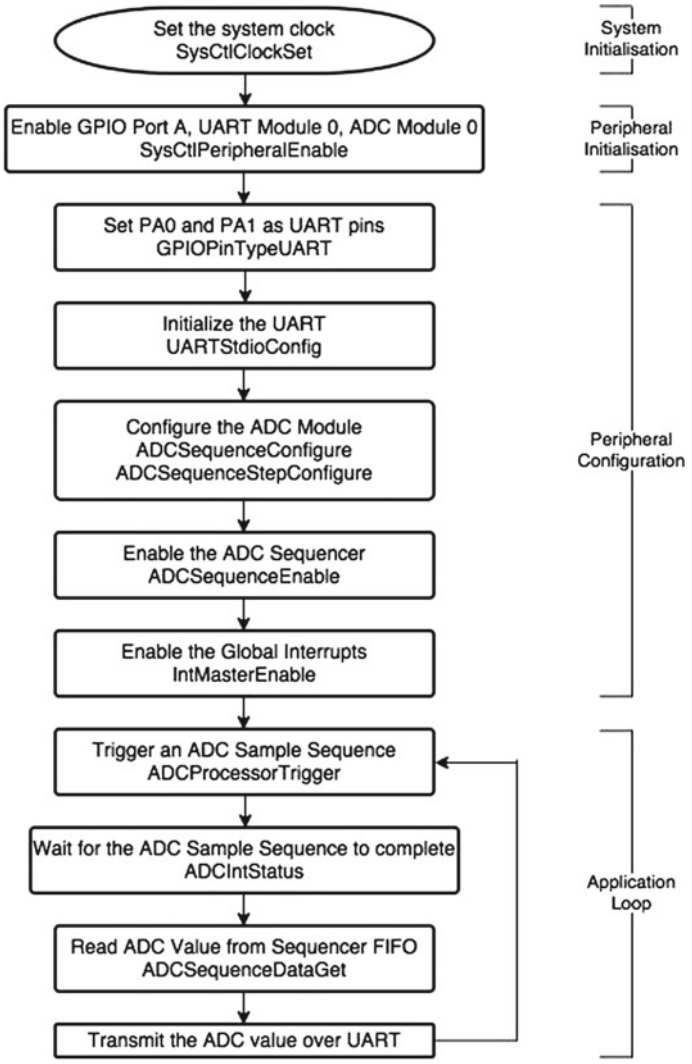

Fig. 3 Program flow for thumbwheel

3.3 Program Flow

Since experiment requires one analog channel, so Sample Sequencer 3 can be used which has only one step. The algorithm to perform the experiment is as follows:

1. Enable the system clock, GPIO Port A, UART Module 0, and ADC Module 0.
2. Configure the UART and ADC modules with appropriate configuration parameters.

3. Read the ADC value from ADC Sequencer FIFO and display it on UART Terminal.
4. Move the thumbwheel potentiometer and observe the changes on ADC values on UART Terminal.

Figure 3 explains the program flow of experiment with appropriate API function calls.

3.4 Useful API Function Calls

Below are API Function Calls needed to perform ADC related operation with Tiva C Series family of microcontrollers.

1. **ADCSequenceConfigure -** It configures the trigger source and priority for the sequencer.
   Prototype: void ADCSequenceConfigure(uint32_t ui32Base, uint32_t ui32 SequenceNum, uint32_t ui32Trigger, uint32_t ui32Priority)
   Parameters: *ui32Base* is the base address of the ADC module. ***ui32Sequence Num*** is sample sequence number. ***ui32Trigger*** defines the trigger source for analog to digital conversion. ***ui32Priority*** defines the priority of the sample sequencer among the other sample sequencer.
   Description: This function configures the initiation criteria for a sample sequence. Valid sample sequencers range from zero to three; sequencer zero captures up to eight samples, sequencers one and two capture up to four samples, and sequencer three captures a single sample.
   The ***ui32Trigger*** parameter can take on the following values:
   ADC_TRIGGER_PROCESSOR , ADC_TRIGGER_COMP0,
   ADC_TRIGGER_COMP1 , ADC_TRIGGER_COMP2,
   ADC_TRIGGER_EXTERNAL , ADC_TRIGGER_TIMER,
   ADC_TRIGGER_PWM0 , ADC_TRIGGER_PWM1,
   ADC_TRIGGER_PWM2 , ADC_TRIGGER_PWM3,
   ADC_TRIGGER_ALWAYS.
   The ***ui32Priority*** parameter is a value between 0 and 3, where 0 represents the highest priority and 3 the lowest.
   Returns: None.
2. **ADCSequenceStepConfigure -** It configures the step of the sample sequencer.
   Prototype: void ADCSequenceStepConfigure(uint32_t ui32Base, uint32_t ui32 SequenceNum, uint32_t ui32Step, uint32_t ui32Config)
   Parameters: *ui32Base* is the base address of the ADC module. ***ui32SequenceNum*** is sample sequence number. ***ui32Step*** is the step of the sequencer to be configured. ***ui32Config*** is logical OR of the configuration parameters of the step like **ADC_CTL_TS, ADC_CTL_IE, ADC_CTL_END, ADC_CTL_D,** analog input channels **ADC_CTL_CHx** and digital comparator selects **ADC_CTL_CMPx**.

Description: This function configures the step of the sample sequencer. The ADC can be configure for differential operation with the help of **ADC_CTL_D** bit. The ADC channel to be sampled can be selected using parameter **ADC_CTL_CHx**, where x is ADC channel number. Use **ADC_CTL_TS** to sample the internal temperature sensor value. Also, use **ADC_CTL_END** to make the step defined as last step in sequencer. To enable the step when the sequence is completed use **ADC_CTL_IE**. Generally, this parameter is used with the last step of the sequencer so all data in FIFO register can be read then.
Returns: None.

3. **ADCSequenceEnable -** It enables the sample sequencer.
   Prototype: void ADCSequenceEnable(uint32_t ui32Base, uint32_t ui32 SequenceNum)
   Parameters: *ui32Base* is the base address of the ADC module. ***ui32Sequence Num*** is sample sequence number.
   Description: This function enables the sample sequencer to capture when trigger occurs. A sample sequencer must be configured before it is enabled.
   Returns: None.
4. **ADCSequenceDisable -** It enables the sample sequencer.
   Prototype: void ADCSequenceDisable(uint32_t ui32Base, uint32_t ui32Sequence Num)
   Parameters: *ui32Base* is the base address of the ADC module. ***ui32Sequence Num*** is sample sequence number.
   Description: This function disables the sample sequencer to capture when trigger occurs. A sample sequencer must be disabled before it is configured.
   Returns: None.
5. **ADCIntClear -** It clears sample sequence interrupt source.
   Prototype: void ADCIntClear(uint32_t ui32Base, uint32_t ui32 SequenceNum)
   Parameters: *ui32Base* is the base address of the ADC module. ***ui32Sequence Num*** is sample sequence number.
   Description: This function clears the interrupt for the specified sample sequence.
   Returns: None.
6. **ADCProcessorTrigger -** It causes the processor trigger for a sample sequence.
   Prototype: void ADCProcessorTrigger(uint32_t ui32Base, uint32_t ui32
   SequenceNum)
   Parameters: *ui32Base* is the base address of the ADC module. ***ui32SequenceNum*** is sample sequence number.
   Description: This function triggers a processor initiated sample sequence if the sample sequence trigger is configured to **ADC_TRIGGER_PROCESSOR**.
   Returns: None.
7. **ADCSequenceDataGet -** It gets the captured data in FIFO of sample sequence.
   Prototype: int32_t ADCSequenceDataGet(uint32_t ui32Base, uint32_t ui32Sequence
   Num, uint32_t *pui32Buffer)

Parameters: *ui32Base* is the base address of the ADC module. ***ui32Sequence Num*** is sample sequence number.***pui32Buffer*** is the address where data is stored.
Description: This function copies data from the specified sample sequencer output FIFO to a memory resident buffer. The number of samples available in the hardware FIFO is copied into the buffer, which is assumed to be large enough to hold that many samples.
Returns: It returns the number of samples copied to the buffer.

3.5 Program Code

The complete C program is given below. The code is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Defines and Macros for ADC API */
#include "driverlib/adc.h"

/* Defines and Macros for UART API */
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

/* Defines and Macros for Interrupt API */
#include "driverlib/interrupt.h"

void InitConsole(void)
{
/* Initializing GPIO Port A and UART Module 0 */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

/* Make the UART pins be peripheral controlled */
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

/* Initialize the UART for console I/O */
UARTStdioConfig(0, 115200, SysCtlClockGet());
}
```

```
int main(void)
{
unsigned long temp[1];


/* Set the clock to 80MHz from the crystal of 16MHz using PLL */
SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Enable ADC Peripheral */
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

/* UART configuration */
InitConsole();

/* Configure ADC Peripheral */
/* Before Configuring ADC Sequencer 3, it should be OFF */
ADCSequenceDisable(ADC0_BASE, 3);

/* Configure ADC Sequence */
ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
/* Sequencer Step 0: Samples Analog Channel 6 (Potentiometer) */
ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH6| ADC_CTL_IE |
ADC_CTL_END);

/* Enable ADC sequence */
ADCSequenceEnable(ADC0_BASE, 3);

/* Clear ADC Interrupt */
ADCIntClear(ADC0_BASE, 3);
IntMasterEnable();
while(1)
  {
  ADCProcessorTrigger(ADC0_BASE, 3);
  while(!ADCIntStatus(ADC0_BASE, 3, false))
    {
    }
  ADCIntClear(ADC0_BASE, 3);
  ADCSequenceDataGet(ADC0_BASE, 3, temp);

  UARTprintf("POT:");
  UARTprintf("%04d\n", temp[0]);
  SysCtlDelay(SysCtlClockGet()/50);
  }
}
```

4 Experiment 32—Controlled Temperature Sensor

4.1 Objective

Use LM35 temperature sensor to sense ambient temperature and send it to host PC only when thumbwheel potentiometer value exceeds 1.65 V.

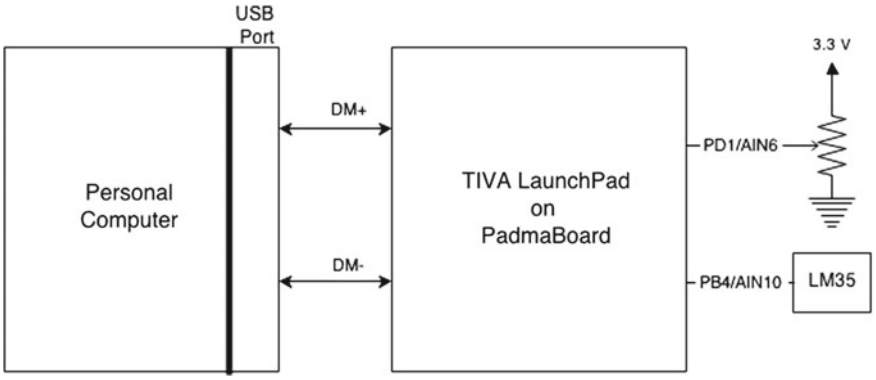

Fig. 4 Block diagram for controlled temperature sensor

4.2 Hardware Description

This experiment requires LM35 temperature sensor, thumbwheel potentiometer and UART Terminal on host PC to sense and display ambient temperature. LM35 generates voltage output proportional to the surrounding temperature. LM35 is connected to Analog Channel 10 (AIN10) which is multiplexed with GPIO PB4. Connection of LM35 to pin PB4 is through a jumper, JP6. As discussed in previous experiment thumbwheel potentiometer is connected to Analog Channel 6 (AIN6) through a jumper, JP1. For appropriate placement of jumpers refer to Chap. 4, Sect. 5. Figure 4 shows the block diagram of experiment.

4.3 Program Flow

The main aim of experiment is to read multiple analog channels. As, experiment requires two analog channels, one for temperature sensor and one for thumbwheel potentiometer. So, the sample Sequencer 1 is used which have four steps. Map the analog channels connected to LM35 and potentiometer as input to Step 0 and Step 1, respectively, of sample Sequencer 1. Also, for 1.65 V the corresponding 12-bit output of ADC (having reference value as 3.3V) is 2048 in decimal. The algorithm for program flow of experiment is as follows:

1. Enable the system clock, GPIO Port A, UART Module 0, and ADC Module 0.
2. Configure the UART module with appropriate configuration parameter.
3. Configure the ADC module using the sequencer as described above.
4. Read the Sample Sequencer data, if thumbwheel potentiometer value is greater than 2048, display the temperature value on UART Terminal. Figure 5 shows the program flow of the experiment with appropriate API function calls.

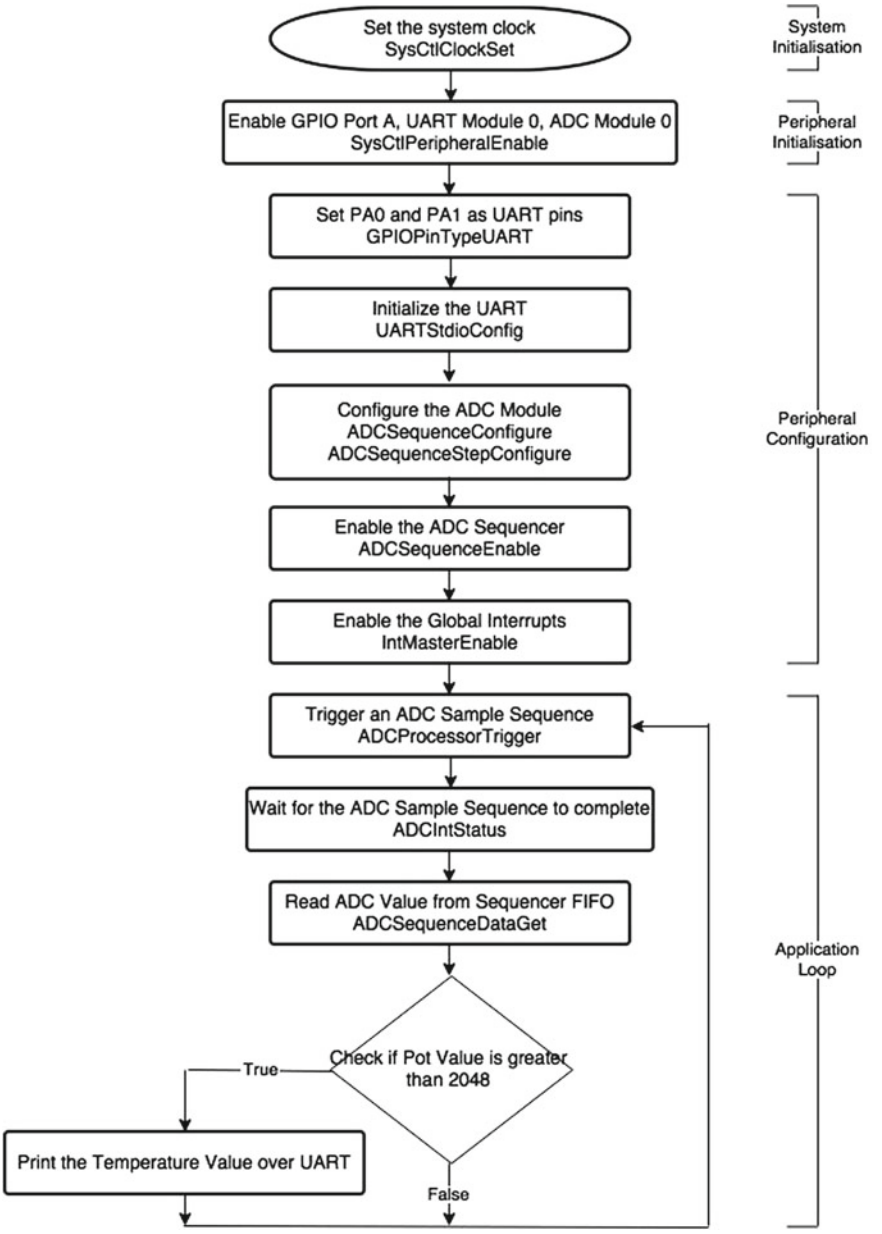

Fig. 5 Program flow for controlled temperature sensor

4.4 Program Code

The complete C program is given below. The code is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Defines and Macros for ADC API */
#include "driverlib/adc.h"

/* Defines and Macros for UART API */
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

/* Defines and Macros for Interrupt API */
#include "driverlib/interrupt.h"

void InitConsole(void)
{
/* Initializing GPIO Port A and UART Module 0 */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

/* Make the UART pins be peripheral controlled */
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

/* Initialize the UART for console I/O */
UARTStdioConfig(0, 115200, SysCtlClockGet());
}


int main(void)
{
unsigned long temp[4], celsius;
/* Set the clock to 80MHz from the crystal of 16MHz using PLL */
SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Enable ADC Peripheral */
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

/* UART configuration */
InitConsole();
```

```
/* Configure ADC Peripheral */
/* Before Configuring ADC Sequencer 1, it should be OFF */
ADCSequenceDisable(ADC0_BASE, 1);

/* Configure ADC Sequence */
ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
/* Sequencer Step 0: Samples Analog Channel 10 (Temperature Sensor) */
ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH10);
/* Sequencer Step 1: Samples Analog Channel 6 (Potentiometer Sensor) */
ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_CH6| ADC_CTL_IE |
ADC_CTL_END);

/* Enable ADC sequence */
ADCSequenceEnable(ADC0_BASE, 1);

/* Clear ADC Interrupt */
ADCIntClear(ADC0_BASE, 1);
IntMasterEnable();

while(1)
  {
  ADCProcessorTrigger(ADC0_BASE, 1);
  while(!ADCIntStatus(ADC0_BASE, 1, false))
    {
    }
  ADCIntClear(ADC0_BASE, 1);
  ADCSequenceDataGet(ADC0_BASE, 1, temp);

  if(temp[1]>2048)
    {
    celsius=temp[0]*330/4096;
    UARTprintf("Temperature :");
    UARTprintf("%04d\n", celsius);
    SysCtlDelay(SysCtlClockGet()/50);
    }
  }
}
```

5 Experiment 33—Thumbwheel Intensity Control

5.1 Objective

Use thumbwheel potentiometer to control the intensity of RGB LED.

5.2 Hardware Description

This experiment requires a RGB LED and thumbwheel potentiometer. Tiva Launch-Pad has the RGB LED connected to PF1, PF2, and PF3. PadmaBoard has thumbwheel

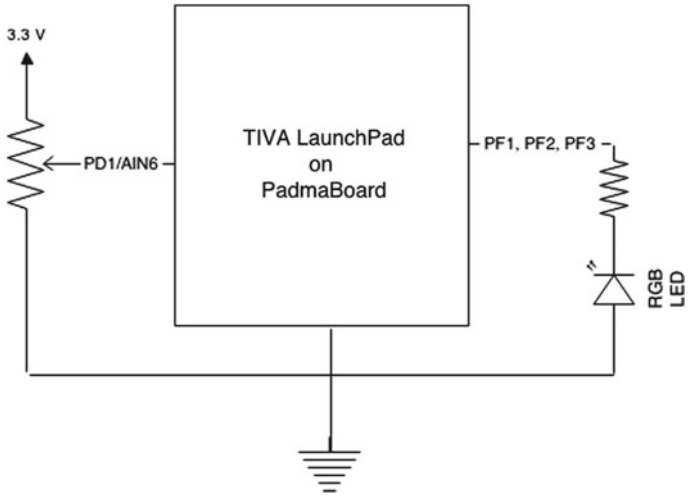

Fig. 6 Block diagram for thumbwheel intensity control

potentiometer connected to analog channel 6 (AIN6) through a jumper, JP1. Figure 6 shows the block diagram of experiment.

5.3 Program Flow

In this experiment value ADC value of thumbwheel potentiometer is simulated on the RGB LED, by vaying its intensity. Hence, the duty cycle of the PWM of RGB LED will depend on the ADC value of thumbwheel potentiometer. The PWM generated can be software or hardware, but it is preferred to be hardware PWM. The algorithm to perform the experiment is shown below:

1. Enable the system clock, GPIO Port F and ADC Module 0.
2. Configure the ADC modules with appropriate configuration parameters.
3. Configure the PF1, PF2, and PF3 according to Software PWM or Hardware PWM.
4. Read the ADC value from ADC Sequencer FIFO and used use it to vary the duty cycle of PWM at RGB LED.
5. Move the thumbwheel potentiometer and observe the change in intensity of RGB LED.

Figure 7 shows the program flow for the experiment.

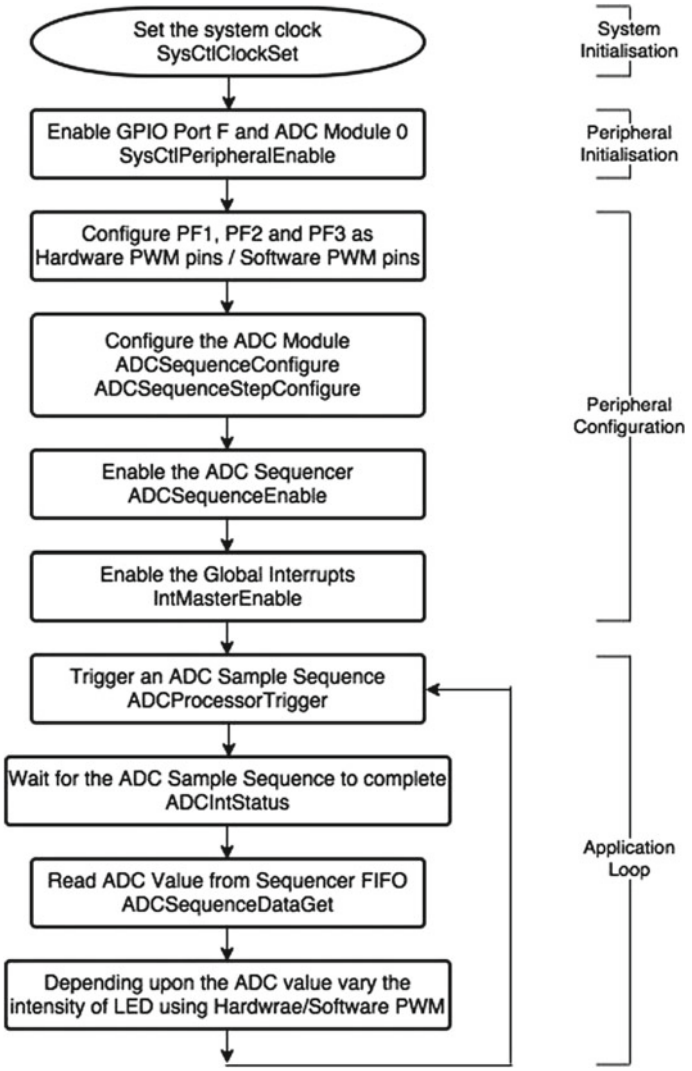

Fig. 7 Program Flow for thumbwheel intensity control

6 Experiment 34—Mini VU Meter

6.1 Objective

Sample the sound using Microphone and plot the sound level on three LEDs on PadmaBoard.

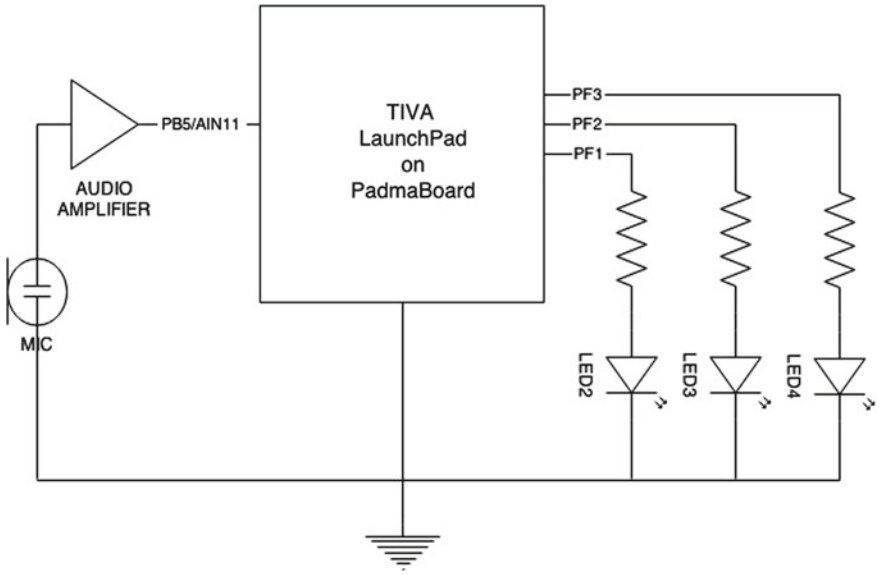

Fig. 8 Block diagram for mini VU meter

6.2 Hardware Description

This experiment requires microphone for audio input and three LEDs. Three LEDs on PadmaBoard are connected to PF1, PF2, and PF3 (same GPIOs to which RGB LED is connected). Microphone is connected to Analog Channel 11 (AIN11) multiplexed with GPIO PB5. Microphone is connected to pin PB5 through a jumper, JP5. Refer to Chap. 4, Sect. 5 for proper placement of jumpers. Figure 8 shows block diagram of the experiment.

6.3 Program Flow

In this experiment, one analog channel is need to be sampled to which microphone is connected, AIN11. The magnitude of sound can be plotted on the three LEDs on PadmaBoard like VU meter. Like, if magnitude of sound is high then turn on all three LEDs, if it is low turn on one LED, LED2 (connected to PF1) and if it is medium turn on two LEDs, LED2, and LED3 (connected to PF1 and PF2). The algorithm to perform the experiment is discussed below:

1. Enable the system clock, GPIO Port F and ADC Module 0.
2. Configure the ADC modules with appropriate configuration parameters.
3. Configure the PF1, PF2, and PF3 as output.

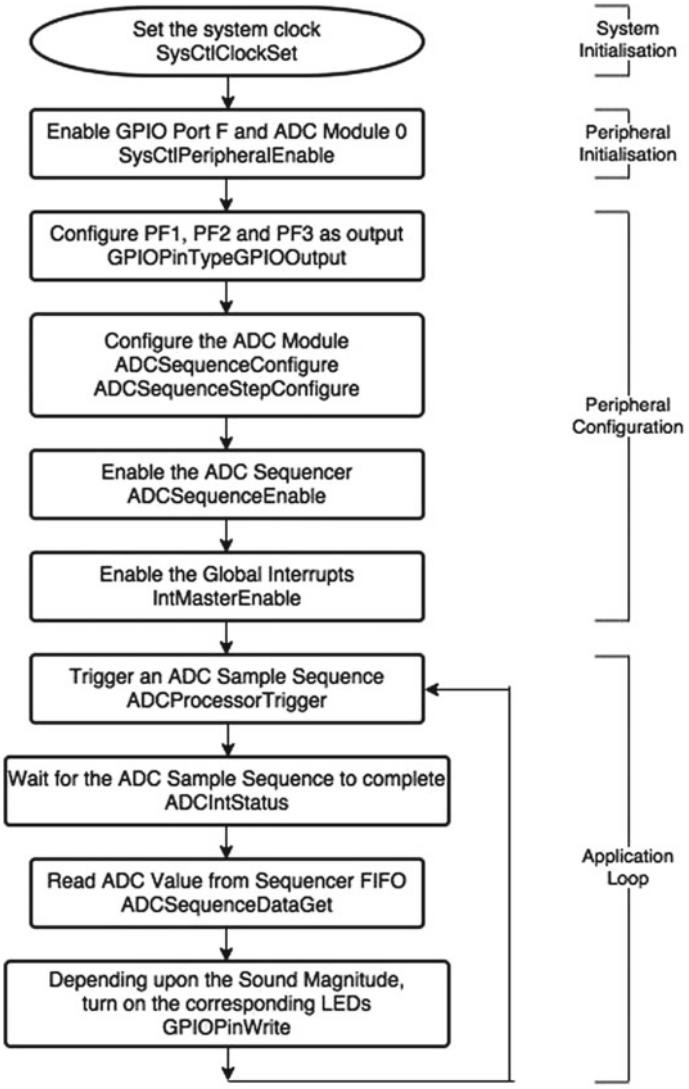

Fig. 9 Program flow for mini VU meter

4. Read the ADC value from ADC Sequencer FIFO and turn on the corresponding LEDs.

Figure 9 shows the program flow for the experiment.

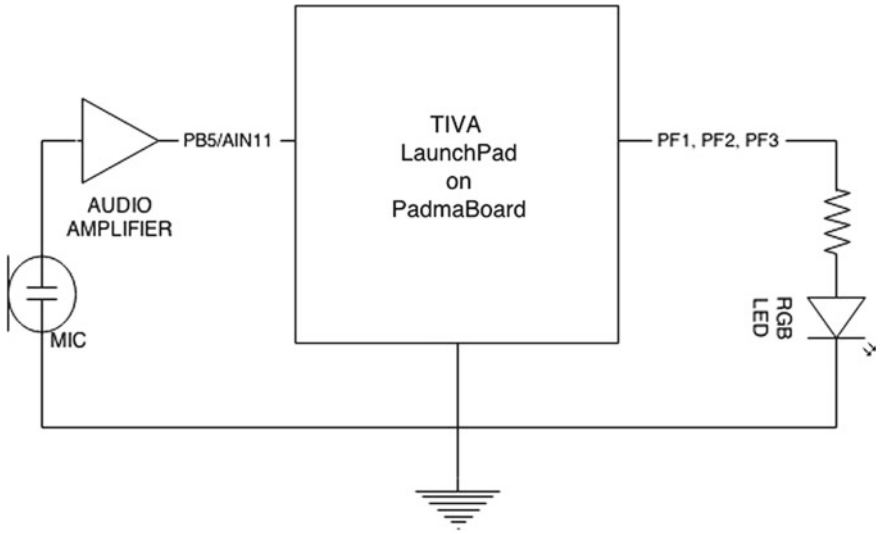

Fig. 10 Block diagram for sound control

7 Experiment 35—Sound Control

7.1 Objective

Use the audio input to control the intensity of RGB LED. To vary intensity of RGB LED use either Hardware PWM or Software PWM. Increase the intensity of RGB LED with increase in magnitude of sound.

7.2 Hardware Description

This experiment requires microphone for audio input and RGB LED. The block diagram for experiment is shown in Fig. 10.

7.3 Experiment Tips

Audio Input can be provided through either microphone or 3.5 mm Audio Jack through a jumper, JP5. The audio input is biased to a particular voltage and with audio as input, output voltage oscillates about the bias voltage.

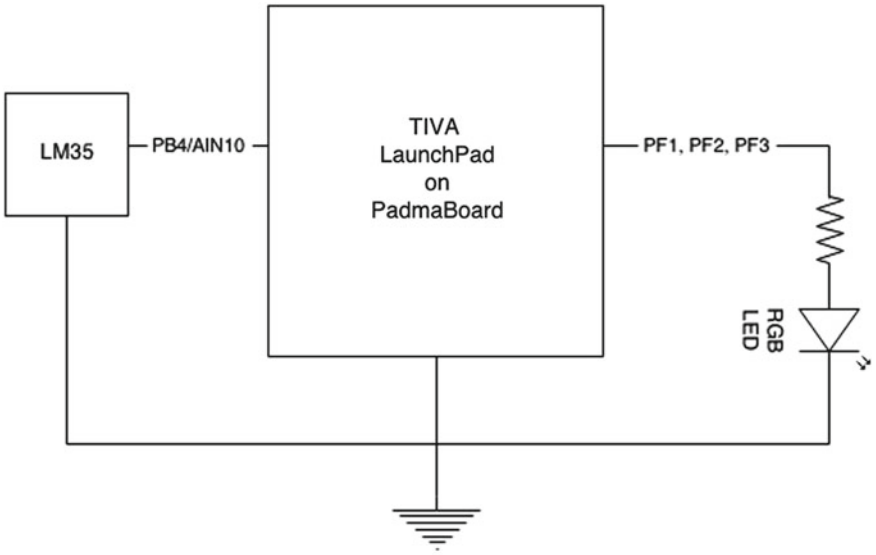

Fig. 11 Block diagram for temperature on RGB

8 Experiment 36—Temperature on RGB

8.1 Objective

Sense the temperature using LM35, and map temperature value on RGB LED. Mapping of temperature value should be such that, when temperature is below 10°C, blue LED will turn on and intensity of LED will keep on increasing as temperature drops further. When the temperature is between 10 and 35°C, green LED will turn on with its intensity maximum at 35°C and minimum at 10°C. When the temperature increases above 35°C, red LED will turn on, and its intensity will keep on increasing as temperature increases further.

8.2 Hardware Description

To perform this experiment RGB LED and temperature sensor LM35 is required. Both these components have been discussed in previous experiments. Figure 11 shows the block diagram of the experiment.

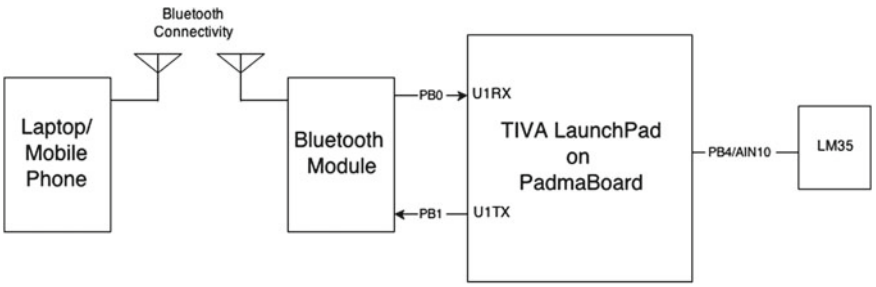

Fig. 12 Block diagram for temperature over Bluetooth

9 Experiment 37—Temperature Over Bluetooth

9.1 Objective

Temperature is sensed using LM35 and is transmitted to another device wirelessly using bluetooth.

9.2 Hardware Description

To perform this experiment bluetooth module and LM35 temperature sensor is required. Also, another bluetooth device which can be personal computer with bluetooth or mobile phone with bluetooth terminal to which data can be received. Block diagram for the experiment is shown in Fig. 12.

10 Experiment 38—Improved Ultasonic Ranger

10.1 Objective

Develop a contact less distance measurement system using ultrasonic module which is temperature compensated. Display the measured distance on UART terminal of host PC.

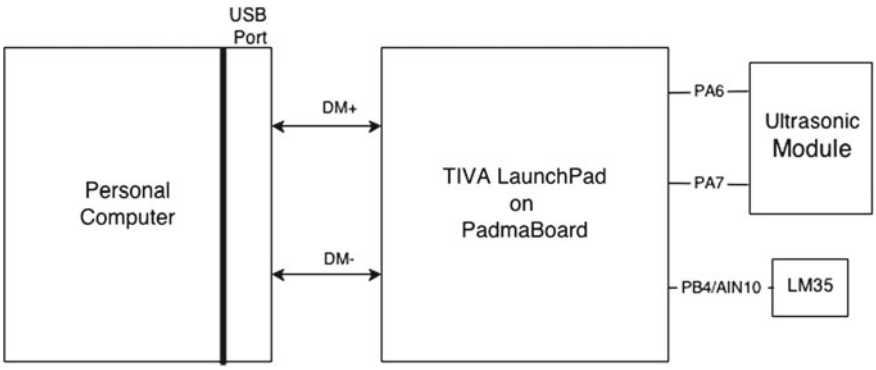

Fig. 13 Block diagram for improved ulrasonic ranger

10.2 Hardware Description

To perform this experiment ultrasonic module and temperature sensor, LM35 is required. Since, ultrasonic module emits ultrasonic sound wave, this wave reflects backs from the obstacle and reaches back to ultrasonic module. Distance will be determined from the time taken by ultrasonic sound wave to reach back to ultrasonic module. The speed of sound wave varies with surrounding temperature, hence temperature compensation is needed to include in distance measured by ultrasonic module. Since, connections to ultrasonic module connector are multiplexed IR transmitter and receiver on PadmaBoard by jumpers JP3 and JP4. Place the jumpers and ultrasonic module properly as discussed in previous experiments. Also, host PC with UART terminal to display the distance with temperature error correction using ultrasonic module. Figure 13 shows the block diagram of experiment.

11 Experiment 39—Temperature Alarm

11.1 Objective

Develop the alarm system which starts buzzing when the temperature exceeds 40°C, and the frequency of beeping sound should increase with increase in temperature.

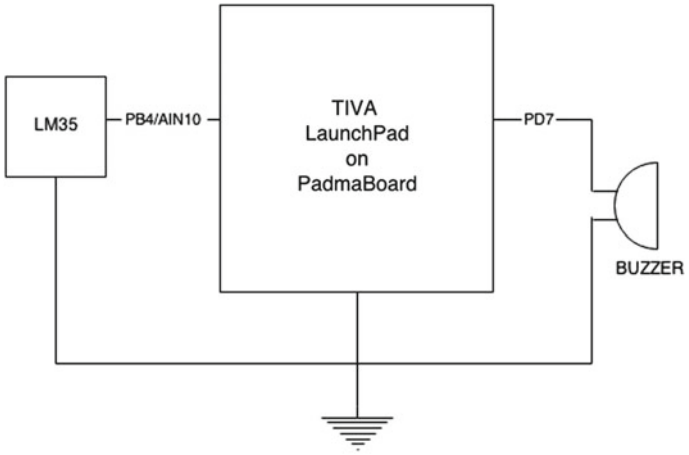

Fig. 14 Block diagram for temperature alarm

11.2 Hardware Description

This experiment requires a temperature sensor (LM35) and a buzzer to generate buzz sound when the temperature reaches above the certain temperature. Figure 14 shows the block diagram of the experiment.

12 Experiment 40—Thermistor Linearization

12.1 Objective

Use thermistor to sense temperature using the linear approximation method, and display the temperature on host PC.

12.2 Hardware Description

This experiment requires thermistor to sense the temperature,and UART terminal to display the temperature. The input from thermistor circuitry is multiplexed with input from LM35 through a jumper, JP6. Place the jumpers appropriately as discussed in Chap. 4 under Sect. 5. Block diagram for the experiment is shown in Fig. 15.

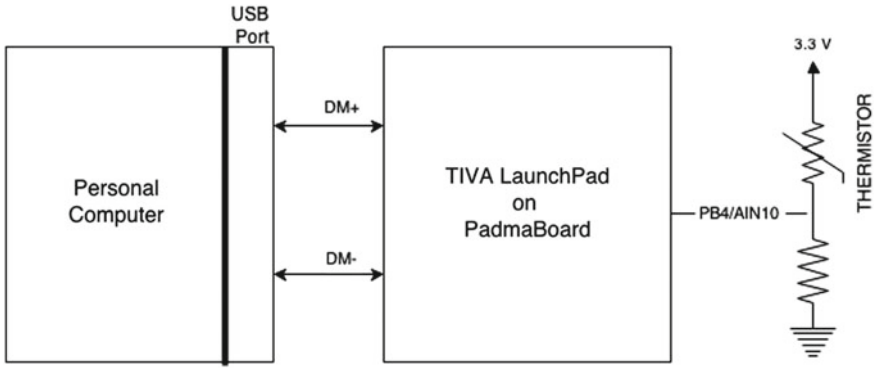

Fig. 15 Block diagram for thermistor linearization

13 Experiment 41—Thermistor Lookup

13.1 Objective

Use thermistor to measure temperature by using look up table method, and display the temperature on host PC.

13.2 Hardware Description

The hardware requirement for this experiment is same as the previous experiment.

14 Experiment 42—Hall Effect Sensor

14.1 Objective

Use Hall effect sensor to sense change in magnetic field with the help of magnet moving around the sensor. And, display the corresponding analog value on host PC.

14.2 Hardware Description

The Hall effect sensor used is A1302 which generates the analog output voltage linearly varying with the magnetic field. The Hall effect sensor is connected to analog

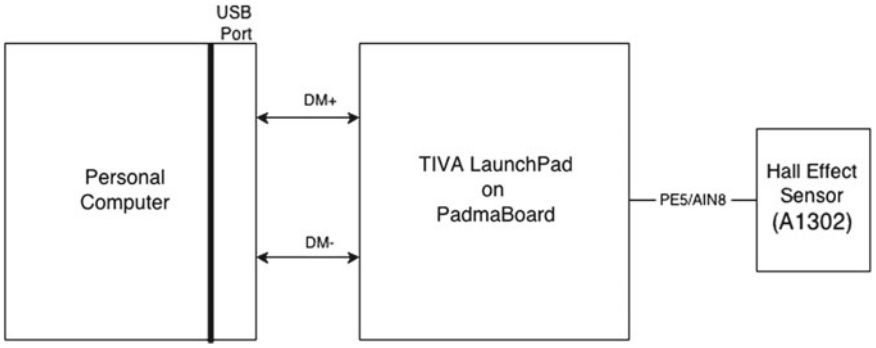

Fig. 16 Block diagram for hall effect sensor

channel 8 which is multiplexed with pin PE5. Hall effect sensor is connected to pin PE5 through a jumper, JP2. Place the jumper properly to connect the sensor to pin PE5. Block diagram for the experiment is shown in Fig. 16.

15 Experiment 43—Speedometer

15.1 Objective

Use the Hall effect sensor to sense the speed of rotating object. And display the speed on host PC.

15.2 Hardware Description

Hardware description of experiment is same as that of previous experiment.

15.3 Experiment Tips

In this experiment glue a magnet to the rotating body, and place the sensor at fix position near the rotating body. Whenever, magnet crosses that fixed position (where sensor is placed) Hall effect sensor will detect the magnet. Just count the number of times magnet crosses that fixed position in one second, which will give the number of rotations per second. If rotating object is wheel, which is having a translational motion also. By knowing the diameter of the wheel and rotations per second speed can be calculated.

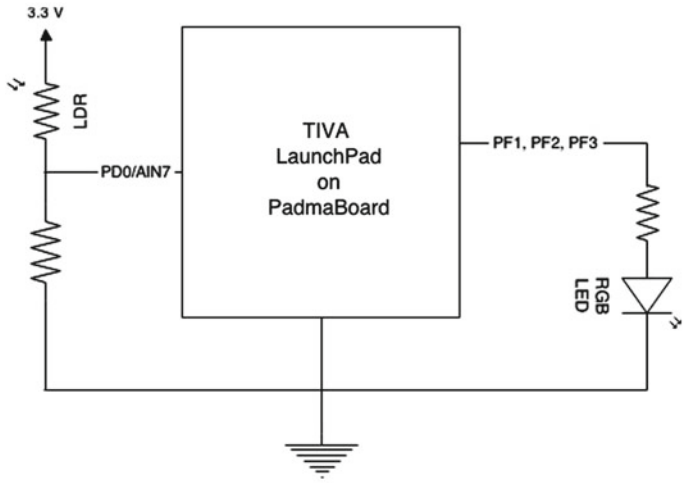

Fig. 17 Block diagram for automatic night lamp

16 Experiment 44—Automatic Night Lamp

16.1 Objective

Detect the surrounding light intensity using Light Dependent Resistor (LDR) and control the intensity of LED according to it. If ambient light intensity decreases, increase the intensity of LED and vice-versa.

16.2 Hardware Description

To perform this experiment a Light Dependent Resistor (LDR) and LED is required. Variation in intensity of LED can be performed using software PWM or hardware PWM. The resistance of LDR changes with change in intensity of light falling on it. So this change in resistance can be measured by using another fixed value series resistance, as resistance of LDR changes voltage drop across LDR will also change. The LDR is connected to analog channel 7 which multiplexed with GPIO Pin PD0. LDR is connected to PD0 through the jumper, JP7. The block diagram of the hardware setup is shown in Fig. 17.

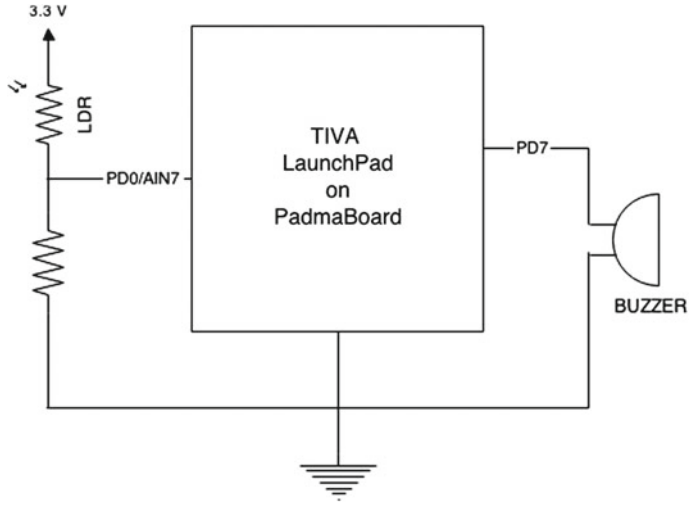

Fig. 18 Block diagram for light alarm

17 Experiment 45—Light Alarm

17.1 Objective

Sense ambient light using LDR when it falls below a certain limit trigger the alarm using buzzer.

17.2 Hardware Description

To perform this experiment LDR and buzzer is required. Buzzer is connected is connected to PD7, which is locked due to NMI functionality, hence it needs to be unlocked. Figure 18 shows the block diagram for the experiment.

18 Experiment 46—Sound Measurement

18.1 Objective

Measure the sound levels using either microphone or 3.5 mm audio jack and plot the waveform on application running on host computer.

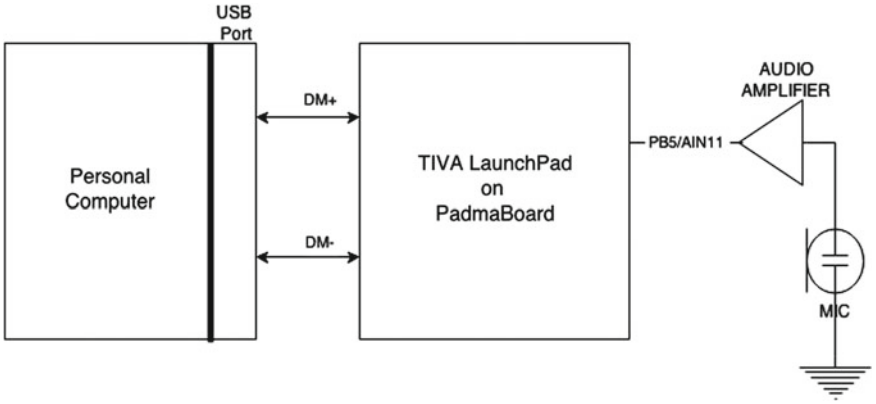

Fig. 19 Block diagram for sound measurement

18.2 Hardware Description

To perform this experiment audio input either through microphone or 3.5 mm audio jack is required. Block diagram for the experiment is shown in Fig. 19. Though in the block diagram microphone is depicted as audio input source but it can be 3.5 mm audio input also.

18.3 Experiment Tips

The 12-bit ADC value of audio input is transferred to host PC over UART. On host PC, there will be application running which can be designed using python or processing. This PC application will read the data received over UART and plot it. To test the application a sine wave can be applied on ADC channel instead of audio input and same should get plotted on host PC. The UART baud rate may become bottleneck for input sine wave frequency.

19 Experiment 47—Digital Filters Implementation

19.1 Objective

Implement the digital filters, low pass, band pass, and high pass filter on the audio input. Depict the output of three filters by varying the intensity of three LEDs (LED2, LED3, and LED4) connected to PF1, PF2, and PF3, each LED corresponding to each filter.

19.2 Hardware Description

Hardware requirement is same as that described in experiment Mini VU Meter, earlier.

19.3 Experiment Tips

Implement FIR filters, and store the coefficients of filter in flash as microcontroller have more flash memory available than SRAM. Coefficients for FIR filter will depend on the order of filter implemented. Try to implement higher order filters to have better response. Use hardware PWM to vary the intensity of LEDs.

Chapter 15
Serial Communication: SPI and I2C

This chapter deals with two types of serial communication Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI). Already, one serial communication protocol UART is discussed earlier in Chap. 12. These two peripherals give access to control various features. The features that are covered in this chapter are Real Time Clock (RTC) which can be used for time stamping applications, digital-to-analog converter (DAC) to generate analog signal, microSD card for mass data storage, TV interfacing using composite channel for display. Lots of interesting projects can be developed by combination of these peripherals. As for transceivers like RS232, MAX485 transmission and receiving range is around 2–120 m depending on baud rate and type of transceiver used. But I2C and SPI serial communication cannot be used for such distant communication, generally used to interface devices on same PCB.

1 Inter-Integrated Circuit (I2C)

1.1 Introduction

Inter-Integrated Circuit (I2C) also known as two wire interface. As this serial communication requires two lines, data line (SDA) and clock (SCL). On PadmaBoard, real time clock (RTC) and two 12-bit DAC to generate analog signals are present. Both of these features are based on I2C protocol. There is I2C bus connector on the PadmaBoard[1] to which other breakout boards having different features based on I2C can be connected. Microcontroller on Tiva LaunchPad, TM4C123GH6PM has four such I2C modules with various features as mentioned below:

[1] As the footprint of this I2C bus connector is not universal so, refer to schematics and layout before connecting external breakout board to this bus connector.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_15

- Devices on I2C bus can be configured as master or slave.
- Supports four modes: Master Transmit, Master Receive, Slave Transmit and Slave Receive.
- Supports four transmission speeds: 100 Kbps, 400 Kbps, 1 Mbps, and 3.33 Mbps.
- Various end or begin of transmission based interrupts in both master and slave mode.
- Supports multiple masters and 7-bit addressing mode.

To operate the I2C at Fast Mode (i.e., 400 Kbps), or Fast Mode Plus (i.e., 1 Mbps) or at High-Speed Mode (i.e., 3.33 Mbps), there is minimum amount of system clock requirement by microcontroller for each mode. For more details on minimum system clock requirement refer to microcontroller datasheet.

1.2 Functional Description

I2C has two open-drain lines, data line (SDA), and clock (SCL). These lines are pulled up to VCC via pull up resistors. Advantage of I2C is that multiple number of devices with different address can be connected on the same I2C bus. Among the devices there will masters and slaves.[2] And depending on request, data transfer will take place either from master to slave or from slave to master. Hence, SDA line will be bi-directional. So, this protocol empowers the user to control various devices (with different address) by using only two lines (SDA and SCL).

Similar, to UART serial communication, I2C has start and stop bits. To generate start bit SDA need to be pulled low by keeping SCL high and to generate the stop bit SDA will be pulled high by keeping SCL high. And all the other bits related to data, address of slave, acknowledgment of data, and direction of flow of data (master to slave or slave to master) will be signaled between the start and stop bits. There are two frames, address frame, and data frame. These frames will be signaled between the start and stop bits.

1. **Address Frame** - This frame needs to be sent first to initiate transactions on I2C bus. This frame comprises of 9-bits. Among them first 7-bits are for the slave address. The $8th$ bit of the frame will be indicated by the direction of flow of data (data will be transmitted from master to slave or data will be transmitted from slave to master). These 8-bits are driven by the master, after that master releases the SDA line. And the last bit, $9th$ bit will be acknowledgment bit driven by the slave. If slave of corresponding address which is transacted in beginning of the frame is present then that slave will pull the acknowledgment bit low else if it is not present the bit will remain high.
2. **Data Frame** - This frame is transacted after the address frame. The first 8-bits of this frame are data bits which will transmitted from master to slave or slave to

[2]Clock line on I2C bus is always initiated by the Master.

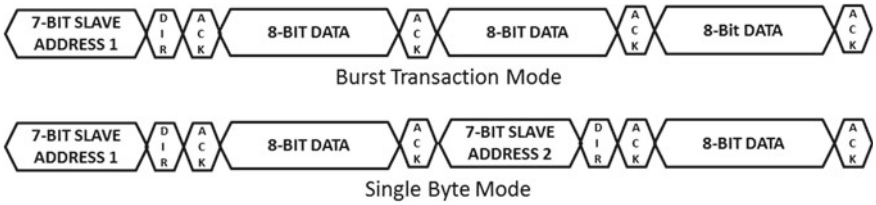

Fig. 1 Transaction modes of I2C

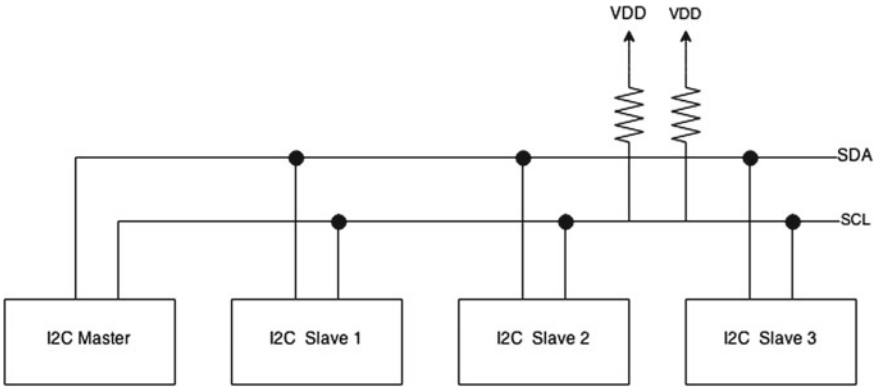

Fig. 2 Block diagram for I2C connection

master depending on the direction bit in address frame. The last bit, $9th$ bit will be acknowledgment bit transmitted by receiving end (it can be master or slave).

If there is an application in which master is performing a transaction of bulk of data with the same slave in same direction (i.e., either transmitting or receiving) then there is no need to send the address frame before each data byte. Only, before the first data byte address is signaled then series of data frames are signaled, this is the burst transaction mode. However, if there is need to switch to some other slave or to change the direction of flow of data then the address frame will be signaled before the next data frame. This is illustrated in Fig. 1.

If there are multiple masters or slaves on I2C bus, then slaves can not communicate with each other. Similarly, masters cannot communicate among themselves. Only master and slave communication is possible. If two masters want to communicate with each other then one has to become a slave, similarly for communication between slaves. Figure 2 shows the block diagram containing one I2C master and multiple slaves.

In case of multiple masters, only one master will be holding the bus at a given time. Only the selected master issues a start condition. This condition is followed by address frame and data frame. After the completion of data transfer master can generate stop condition.

Depending on the master or slave configuration various interrupts can be generated. For example, in master mode interrupt can be generated when transmit or receive operation is completed or when the data transfer is aborted due to an error. In slave mode interrupts can be generated when the data has been sent to or requested by the master.

2 Serial Peripheral Interface (SPI)

2.1 Introduction

SPI sometimes also known as four wire serial communication as it based on four lines, Clock, Data Input, Data Output, and Slave Select. The module enables to select between the three frame formats, Motorola SPI, National Semiconductor Microwire, or the Texas Instruments Synchronous Serial Interface. This communication is preferred whenever high-speed serial communication is required. Bit rate that can be achieved on the TM4C123GH6PM (microcontroller on Tiva LaunchPad) is from 2 Mbps to 25 Mbps. On PadmaBoard there is microSD card and TV output connected to the SPI pins. For TV output SPI protocol is not required, but it requires fast toggling of the GPIO which can be achieved easily on the SPI bus. Microcontroller on Tiva LaunchPad, TM4C123GH6PM has four such modules with various features mentioned below:

- It can be configured for both master and slave operation.
- It has programmable bit rate.
- Separate transmit and receive FIFOs, each 16-bit wide and has 8 locations deep.
- Programmable data frame size varying from 4- to 16-bits.
- Various FIFO based interrupts and end of transmission interrupts.

2.2 Functional Description

This module performs parallel to serial data conversion before transmitting and serial to parallel data conversion on the received data. In SPI, master configures the clock which is supported by the slave device. The master generates the slave select signal, then with each clock cycle master sends the bit on data out line and receives a bit on data input line. In this way, full duplex communication is established between slave and master. Master can configure the clock polarity, which determines the idle state of clock and clock phase. Clock phase determines clock edge when the data bit

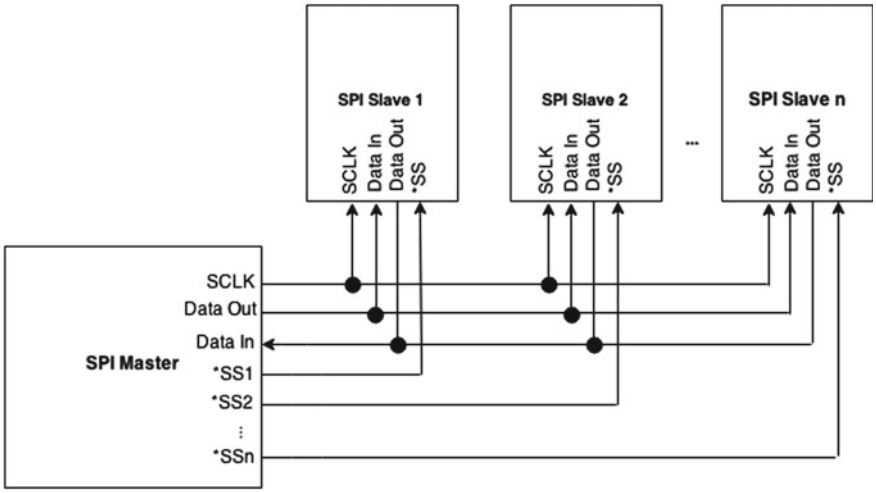

Fig. 3 Block diagram for SPI connection using independent slave configuration

is captured and transmitted by the module. There are four different modes, Mode0, Mode1, Mode2, and Mode3 which determines the clock polarity and clock phase.[3]

Though in microcontroller on Tiva LaunchPad only one slave select pin is available per module. But the GPIOs can also be used as slave select pins which enable to use single SPI module configured in master mode for various slave devices. Figure 3 shows the block diagram containing one SPI master controlling multiple slaves. This type of configuration in which each slave has its own select line is independent slave configuration. Multiple slaves can be connected using daisy chain configuration as shown in Fig. 4. In this configuration, data out of master is connected to data input of slave and data output of first slave is connected to the data input of second slave, and so on. And, data output of last slave is connected to data input of the master. This whole chain acts as shift register and it requires only one select line.

3 Experiment 48—Sine Wave Generator

3.1 Objective

Generate the sine wave using I2C based serial DAC.

[3]By setting one of these mode, user is setting the clock polarity and clock phase. To determine the value of clock polarity and phase in a particular mode refer to microcontroller datasheet.

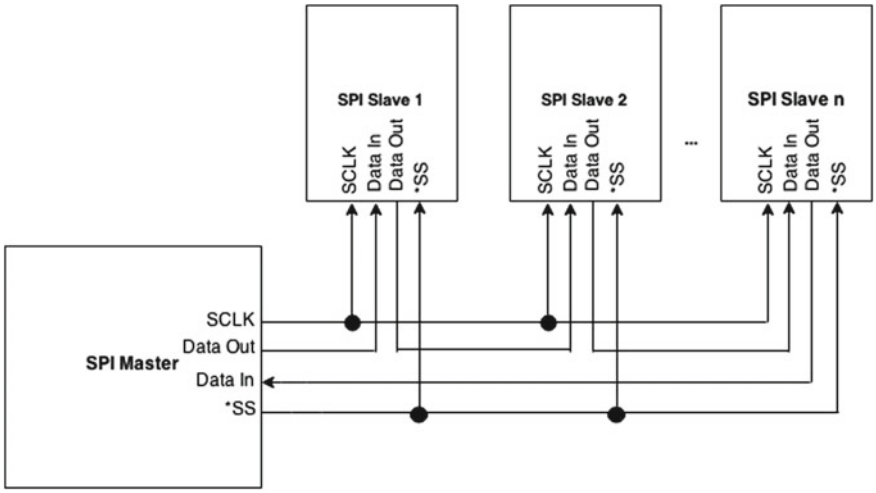

Fig. 4 Block diagram for SPI connection using daisy chain configuration

3.2 Hardware Description

To perform this experiment serial I2C based 12-bit DAC is used to generate sine wave. Oscilloscope or data acquisition card is required to observe the sine waveform. PadmaBoard has two serial 12-bit DAC 7571, the address of both DACs are made unique by using external A0 (LSB of Address). Therefore, the addresses of two DACs are 0×4C and 0×4D (both the addresses are in hexadecimal). In this experiment, DAC with address 0×4C is used whose output can be observed on connector X9. Figure 5 show the block diagram of the hardware setup for experiment. It uses I2C module 0, whose pin is multiplexed with GPIOs, PB2 and PB3 as shown in block diagram.

3.3 Program Flow

In this experiment to generate sine wave, a lookup table is used. Look up table has 512 values containing 12-bit sine values (ranges from 0 to 4095) of one period. Since, only data need to be transmitted to DAC (there is no need to read the data from DAC). Hence, slave address and direction of flow of data will not change. So, there is no need to transfer address frame again and again. Only data frame needs to be transmitted from master to slave. So burst mode of I2C module can be used. In this mode, after start condition address frame is transmitted then bulk of data frames followed by stop condition to stop the transfer of data. Since, in burst mode address frame is not transmitted after each data frame, therefore DAC will be updated

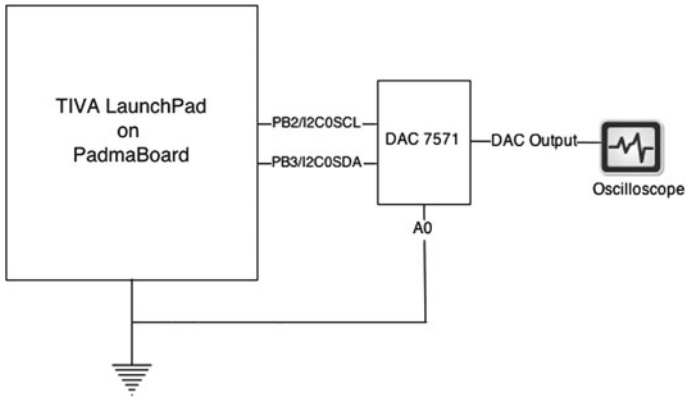

Fig. 5 Block diagram for sine wave generator

more frequently with new data. So, by using burst mode the frequency of sine wave achieved will be higher than frequency achieved during single send mode.[4] Since, in this experiment sine wave is generated continuously so there no need to stop transmitting the data to the serial DAC. Hence, in the program code burst send finish command is included after never ending loop so that the transmission of sine data values to DAC is continuous. That statement is included to show how to terminate burst transactions, although that statement is redundant for this experiment. The algorithm to perform the above experiment is mentioned below:

1. Enable the system clock, GPIO Port B, and I2C Module 0.
2. Configure the I2C module with appropriate configuration parameters and slave address.
3. Start the burst transaction. Read the 12-bit sine value from look up table and break it into 2 bytes to be transferred one after other.
4. Continue transmitting in the burst transaction mode.

The program flow for the experiment us shown in Fig. 6.

3.4 Useful API Function Calls

Useful API calls to perform the experiment involving I2C peripheral are mentioned below.

[4]In single send mode stop condition is generated after each byte transfer. So whenever next byte transfer has to take place it need to send the address over the data line again. Generally this mode is used whenever there need to perform alternate read and write. And burst mode is used whenever there is continuous read or write of data bytes.

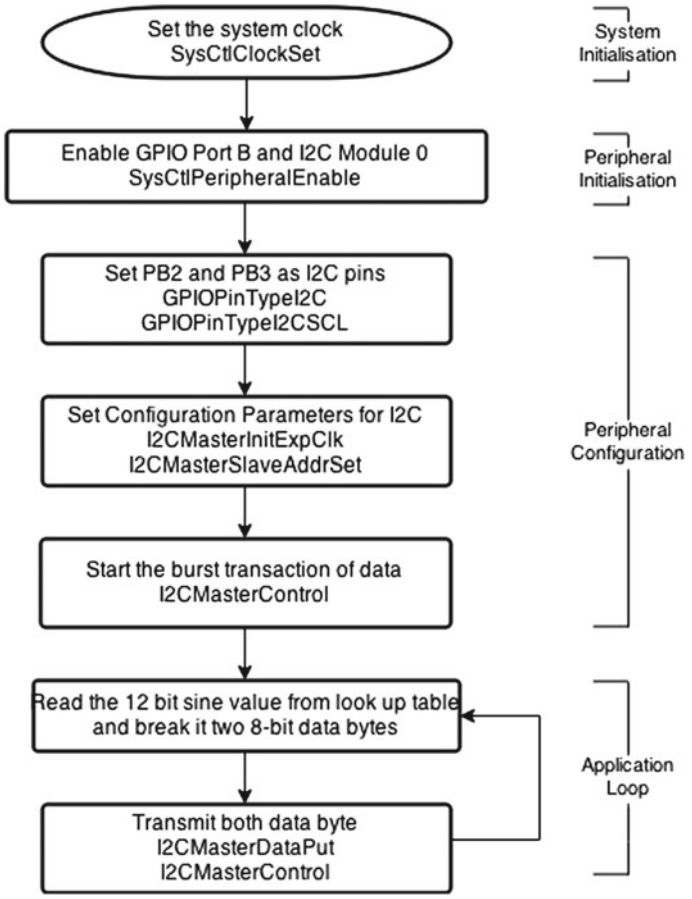

Fig. 6 Program flow for sine wave generator

1. **GPIOPinTypeI2C** - It configures the pin(s) for I2C peripheral.
   Prototype: void GPIOPinTypeI2C(uint32_t ui32Port, uint8_t ui8Pins)
   Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s).
   Description: This function configures the pin(s) to be used as I2C pins only if they are muxed with the I2C peripheral.
   Returns: None.
2. **GPIOPinTypeI2CSCL** - It configures the pin(s) as SCL for I2C peripheral
   Prototype: void GPIOPinTypeI2CSCL(uint32_t ui32Port, uint8_t ui8Pins)
   Parameters: ***ui32Port*** is the base address of the GPIO port. ***ui8Pins*** is the bit-packed representation of the pin(s).
   Description: This function configures the pin(s) to be used as SCL pin of I2C

peripheral, only if they are muxed with the SCL pin of I2C peripheral.
Returns: None.

3. **I2CMasterDisable** - It disables the I2C master block.
   Prototype: void I2CMasterDisable(uint32_t ui32Base)
   Parameters: ***ui32Base*** is the base address of the I2C Master module.
   Description: This function disables operation of the I2C master block.
   Returns: None.
4. **I2CMasterEnable** - It enables the I2C master block.
   Prototype: void I2CMasterEnable(uint32_t ui32Base)
   Parameters: ***ui32Base*** is the base address of the I2C Master module.
   Description: This function enables operation of the I2C master block.
   Returns: None.
5. **I2CMasterInitExpClk** - It initializes the I2C Master block.
   Prototype: void I2CMasterInitExpClk(uint32_t ui32Base, uint32_t ui32I2CClk, bool bFast)
   Parameters: ***ui32Base*** is the base address of the I2C Master module. ***ui32I2CClk*** is the clock supplied to the I2C module. ***bFast*** set up for fast data transfer.
   Description: This function initializes the I2C Master block by configuring the bus speed and enabling the master block. If the parameter ***bFast*** is true it set the I2C data transfer speed as 400 Kbps and if it false it set the speed as 100 Kbps. To enable Fast mode plus (1 Mbps) or High-Speed Mode (3.33 Mbps) user need to write into the registers manually.
   Returns: None.
6. **I2CMasterSlaveAddrSet** - Sets the address that the I2C Master places on the bus.
   Prototype: void I2CMasterSlaveAddrSet(uint32_t ui32Base, uint8_t ui8Slave Addr, bool bReceive)
   Parameters: ***ui32Base*** is the base address of the I2C Master module.***ui8Slave Addr*** is the 7-bit slave address. ***bReceive*** is the flag indicates whether to receive or transmit data to slave.
   Description: This function configures the slave address that I2C master places on the bus and communicates with the same. the flag bit ***bReceive***, if it is **true** then it indicates master wants to read data from slave and if it is **false** master wants to write data to slave.
   Returns: None.
7. **I2CMasterDataPut** - It transmit the byte from I2C Master.
   Prototype: void I2CMasterDataPut(uint32_t ui32Base, uint8_t ui8Data)
   Parameters: ***ui32Base*** is the base address of the I2C Master module.***ui8Data*** data to be transmitted from Master
   Description: This function transmits the data from the I2C master block.
   Returns: None.
8. **I2CMasterControl** - It controls the I2C Master block.
   Prototype: void I2CMasterControl(uint32_t ui32Base, uint32_t ui32Cmd).
   Parameters: ***ui32Base*** is the base address of the I2C Master module. ***ui32Cmd*** command issued to the I2C Master module.

Description: This function controls the state of the operations of the I2C Master module. The parameter **ui32Cmd** can take the following commands.

I2C_MASTER_CMD_SINGLE_SEND
I2C_MASTER_CMD_SINGLE_RECEIVE
I2C_MASTER_CMD_BURST_SEND_START
I2C_MASTER_CMD_BURST_SEND_CONT
I2C_MASTER_CMD_BURST_SEND_FINISH
I2C_MASTER_CMD_BURST_SEND_ERROR_STOP
I2C_MASTER_CMD_BURST_RECEIVE_START
I2C_MASTER_CMD_BURST_RECEIVE_CONT
I2C_MASTER_CMD_BURST_RECEIVE_FINISH
I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP
I2C_MASTER_CMD_QUICK_COMMAND
I2C_MASTER_CMD_HS_MASTER_CODE_SEND

Returns: None.

3.5 Program Code

The complete C program for the experiment is given below. The program is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for the I2C */
#include "inc/hw_i2c.h"
#include "driverlib/i2c.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

#define SLAVE_ADDRESS 0x4C

/* Array of 512 values containing 12-bit sine values for one time period */
unsigned int sine_9Bit[512] =
{
2048, 2073, 2098, 2123, 2148, 2174, 2199, 2224,
2249, 2274, 2299, 2324, 2349, 2373, 2398, 2423,
```

```
2448, 2472, 2497, 2521, 2546, 2570, 2594, 2618,
2643, 2667, 2690, 2714, 2738, 2762, 2785, 2808,
2832, 2855, 2878, 2901, 2924, 2946, 2969, 2991,
3013, 3036, 3057, 3079, 3101, 3122, 3144, 3165,
3186, 3207, 3227, 3248, 3268, 3288, 3308, 3328,
3347, 3367, 3386, 3405, 3423, 3442, 3460, 3478,
3496, 3514, 3531, 3548, 3565, 3582, 3599, 3615,
3631, 3647, 3663, 3678, 3693, 3708, 3722, 3737,
3751, 3765, 3778, 3792, 3805, 3817, 3830, 3842,
3854, 3866, 3877, 3888, 3899, 3910, 3920, 3930,
3940, 3950, 3959, 3968, 3976, 3985, 3993, 4000,
4008, 4015, 4022, 4028, 4035, 4041, 4046, 4052,
4057, 4061, 4066, 4070, 4074, 4077, 4081, 4084,
4086, 4088, 4090, 4092, 4094, 4095, 4095, 4095,
4095, 4095, 4095, 4095, 4094, 4092, 4090, 4088,
4086, 4084, 4081, 4077, 4074, 4070, 4066, 4061,
4057, 4052, 4046, 4041, 4035, 4028, 4022, 4015,
4008, 4000, 3993, 3985, 3976, 3968, 3959, 3950,
3940, 3930, 3920, 3910, 3899, 3888, 3877, 3866,
3854, 3842, 3830, 3817, 3805, 3792, 3778, 3765,
3751, 3737, 3722, 3708, 3693, 3678, 3663, 3647,
3631, 3615, 3599, 3582, 3565, 3548, 3531, 3514,
3496, 3478, 3460, 3442, 3423, 3405, 3386, 3367,
3347, 3328, 3308, 3288, 3268, 3248, 3227, 3207,
3186, 3165, 3144, 3122, 3101, 3079, 3057, 3036,
3013, 2991, 2969, 2946, 2924, 2901, 2878, 2855,
2832, 2808, 2785, 2762, 2738, 2714, 2690, 2667,
2643, 2618, 2594, 2570, 2546, 2521, 2497, 2472,
2448, 2423, 2398, 2373, 2349, 2324, 2299, 2274,
2249, 2224, 2199, 2174, 2148, 2123, 2098, 2073,
2048, 2023, 1998, 1973, 1948, 1922, 1897, 1872,
1847, 1822, 1797, 1772, 1747, 1723, 1698, 1673,
1648, 1624, 1599, 1575, 1550, 1526, 1502, 1478,
1453, 1429, 1406, 1382, 1358, 1334, 1311, 1288,
1264, 1241, 1218, 1195, 1172, 1150, 1127, 1105,
1083, 1060, 1039, 1017,  995,  974,  952,  931,
910,  889,  869,  848,  828,  808,  788,  768,
749,  729,  710,  691,  673,  654,  636,  618,
600,  582,  565,  548,  531,  514,  497,  481,
465,  449,  433,  418,  403,  388,  374,  359,
345,  331,  318,  304,  291,  279,  266,  254,
242,  230,  219,  208,  197,  186,  176,  166,
156,  146,  137,  128,  120,  111,  103,   96,
88,   81,   74,   68,   61,   55,   50,   44,
39,   35,   30,   26,   22,   19,   15,   12,
10,    8,    6,    4,    2,    1,    1,    0,
0,    0,    1,    1,    2,    4,    6,    8,
10,   12,   15,   19,   22,   26,   30,   35,
39,   44,   50,   55,   61,   68,   74,   81,
88,   96,  103,  111,  120,  128,  137,  146,
156,  166,  176,  186,  197,  208,  219,  230,
242,  254,  266,  279,  291,  304,  318,  331,
345,  359,  374,  388,  403,  418,  433,  449,
465,  481,  497,  514,  531,  548,  565,  582,
600,  618,  636,  654,  673,  691,  710,  729,
749,  768,  788,  808,  828,  848,  869,  889,
910,  931,  952,  974,  995, 1017, 1039, 1060,
1083, 1105, 1127, 1150, 1172, 1195, 1218, 1241,
1264, 1288, 1311, 1334, 1358, 1382, 1406, 1429,
1453, 1478, 1502, 1526, 1550, 1575, 1599, 1624,
```

```
1648, 1673, 1698, 1723, 1747, 1772, 1797, 1822,
1847, 1872, 1897, 1922, 1948, 1973, 1998, 2023
};

unsigned char value1, value2;



/* Transmitting 12-bit sine value in packets of 8 bits through I2C */
void dac_data(unsigned int value)
{
value1 = value/256;
value2 = value%256;
I2CMasterDataPut(I2C0_BASE, value1);
I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_CONT);
while(I2CMasterBusy(I2C0_BASE))
  {
  }
I2CMasterDataPut(I2C0_BASE, value2);
I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_CONT);
while(I2CMasterBusy(I2C0_BASE))
  {
  }
}

int main(void)
{
unsigned int i;
/* Set the clock to 80MHz using PLL */
SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Enable the GPIO Port B */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

/* Enable the peripheral I2C Module 0 */
SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);

/* Make the I2C pins be peripheral controlled */
GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_2 | GPIO_PIN_3);
GPIOPinTypeI2CSCL(GPIO_PORTB_BASE,GPIO_PIN_2);

/* Disable the I2C Module 0 */
I2CMasterDisable(I2C0_BASE);

/* Set the configuration of a I2C Module 0 */
/* Parameter 'true' for 400Kbps and 'false' for 100Kbps mode */
I2CMasterInitExpClk(I2C0_BASE, SysCtlClockGet(), true);

/* Set the slave address to which I2C module (master) which to communicate*/
I2CMasterSlaveAddrSet(I2C0_BASE, SLAVE_ADDRESS, false);

/* Start the burst transaction of Data */
I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);

while(1)
  {
  for(i=0;i<512;i++)
    {
    dac_data(sine_9Bit[i]);
    }
```

```
    }

/* Finish the burst transaction of Data*/
/* Since above loop is never ending loop so this statement will never execute*/
I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);
}
```

4 Experiment 49—Real Time Clock

4.1 Objective

Configure the real time clock to current time and date setting and keep printing the updated year, month, day, hour, minute, and seconds on UART terminal.

4.2 Hardware Description

This experiment requires RTC. On PadmaBoard, RTC used is PCF8563 of NXP Semiconductors. The block diagram for the experiment is shown in Fig. 7. This RTC uses I2C module 0 on PadmaBoard whose pins are multiplexed with GPIOs PB2 and PB3. On PadmaBoard, RTC is powered with 3V coin cell in addition to 3.3 V coming from board, so that when the power is turned off, RTC will remain powered and updating the present date and time.[5]

4.3 Program Flow

Real time clock, PCF8563 has the slave address 0x51. The algorithm to perform the experiment is as follows:

1. Enable the system clock, GPIO Port A, GPIO Port B, UART Module 0 and I2C Module 0.
2. Configure the UART and I2C modules with appropriate configuration parameters.
3. Setup the RTC with current date and time which include second, minute, hour, day, weekday, month, and year. Since while setting up the I2C many data need to be written into the RTC registers, so for this burst transaction mode can be used.
4. Send the address of registers needed to read for updated parameters of date and time (like seconds, minutes, hours, day, weekday, month and year) and read back the corresponding register values.

[5]When the power is turned off and then turned on, note that the application need not to setup the RTC again with present date and time. Application should just read date and time from RTC.

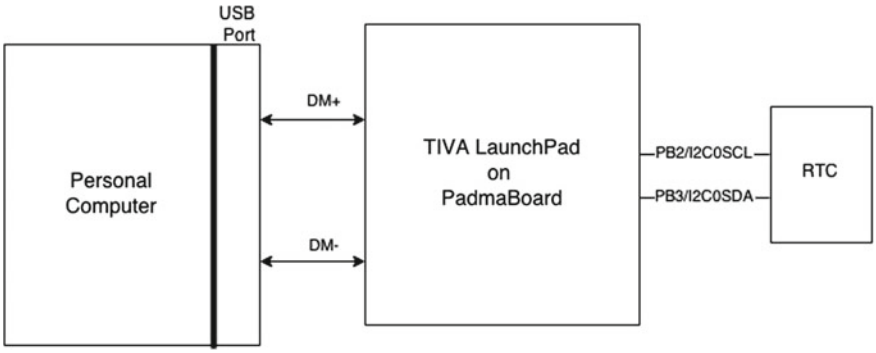

Fig. 7 Block diagram for real time clock

5. Print the updated time and date value on the UART Terminal on host PC.

Figure 8 refers the program flow of experiment.

5 Experiment 50—Alarm Clock

5.1 Objective

Set alarm in RTC to trigger at a particular time and day. When the alarm is triggered use buzzer to indicate it.

5.2 Hardware Description

This experiment requires RTC and buzzer to perform it. The block diagram of experiment is shown in Fig. 9. In this experiment, configure the RTC to trigger the alarm for a particular time and day. At that particular time and day RTC will generate the interrupt. Interrupt line is multiplexed with keypad scan input line connected to GPIO PE0. So if some application requires the use of both RTC and Keypad, application user will have intelligently handle the situation while writing the application code.[6] Once the interrupt is generated buzzer should be triggered.

[6]This multiplexing is done PadmaBoard due to shortage of GPIOs. If user wants to develop its own standalone project such multiplexing should avoid as it decreases both hardware and software complexity.

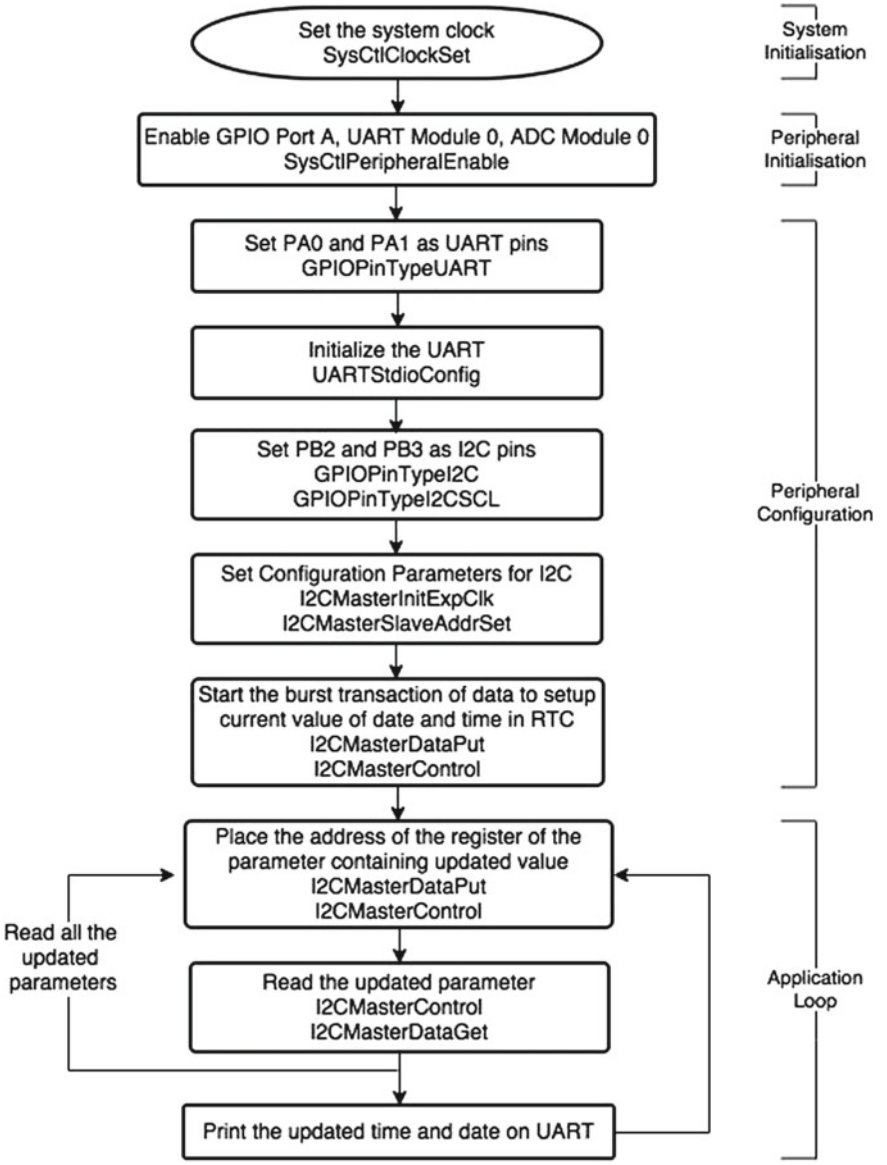

Fig. 8 Program flow for real time clock

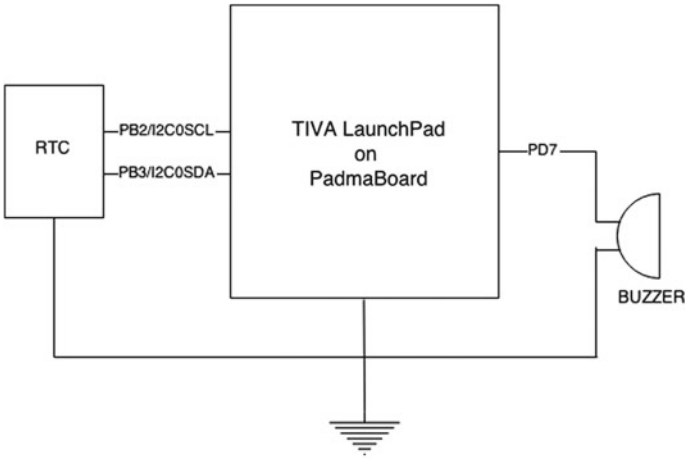

Fig. 9 Block diagram for alarm clock

5.3 Experiment Tips

In this experiment, alarm time and day is hard coded in the application. The same can be taken as input from user through UART terminal on host PC, so that user need not to change the code again and again while setting the alarm.

6 Experiment 51—Twilight Calculator

6.1 Objective

Use RTC to determine the twilight times of the present entire week and display it on UART terminal on host PC.

6.2 Hardware Description

To perform this experiment, RTC is required to keep track of present date and time. And based on this determine twilight times for the entire week. The block diagram for hardware description is same as discussed in experiment RTC.

7 Experiment 52—Sun Tracker

7.1 Objective

Use RTC to track position of the Sun and display it on UART Terminal on host PC.

7.2 Hardware Description

To perform this experiment the hardware requirement is same as discussed in experiment Real Time Clock. As to perform this experiment RTC is required to determine the current date and time as position of Sun depends on this factor. Also, position of Sun depends on the latitude and longitude of location.

8 Experiment 53—High Frequency Sine Wave Generator

8.1 Objective

Use I2C in Fast Mode Plus (1 Mbps) or High-Speed Mode (3.3 Mbps) to generate sine wave with more frequency than in previous experiment.

8.2 Hardware Description

The hardware requirement for this experiment is same as experiment sine wave generator as discussed earlier.

9 Experiment 54—Lissajous Figures

9.1 Objective

Generate the basic Lissajous figures: circle, ellipse, and line on the oscilloscope.

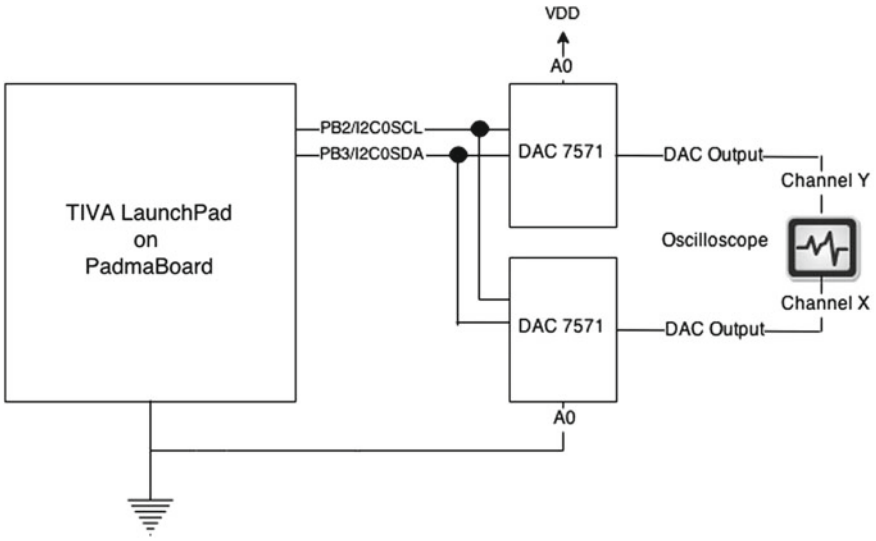

Fig. 10 Block diagram for Lissajous figures

9.2 Hardware Description

This experiment requires two DACs and the oscilloscope which has more than two channels and support X-Y mode. The block diagram for experiment is shown in Fig. 10. The two DACs used on PadmaBoard are I2C based DAC7571 with slave addresses $0\times4C$ and $0\times4D$. The output of both DACs can be observed on connectors X9 and X10, respectively. Connect these outputs to two channels of the oscilloscope and use oscilloscope in X-Y mode.

9.3 Experiment Tips

For better plot on oscilloscope use I2C in Fast Mode Plus or High-Speed Mode. As this experiment uses the concept of persistence of vision, as when signal is given to X and Y channel, then at particular instant only one point will be plotted on the oscilloscope. So, to generate the figure, move the point fast enough corresponding to that figure, so that due to persistence of vision, desired figure is observed on the oscilloscope. To plot the circle generate sine wave on one DAC output and generate cosine (or sine with 90 degree phase shift) of same amplitude on the second DAC. To plot the eclipse generate sine and cosine wave of different amplitudes. And for the line, establish a linear relation between analog outputs of two DACs.

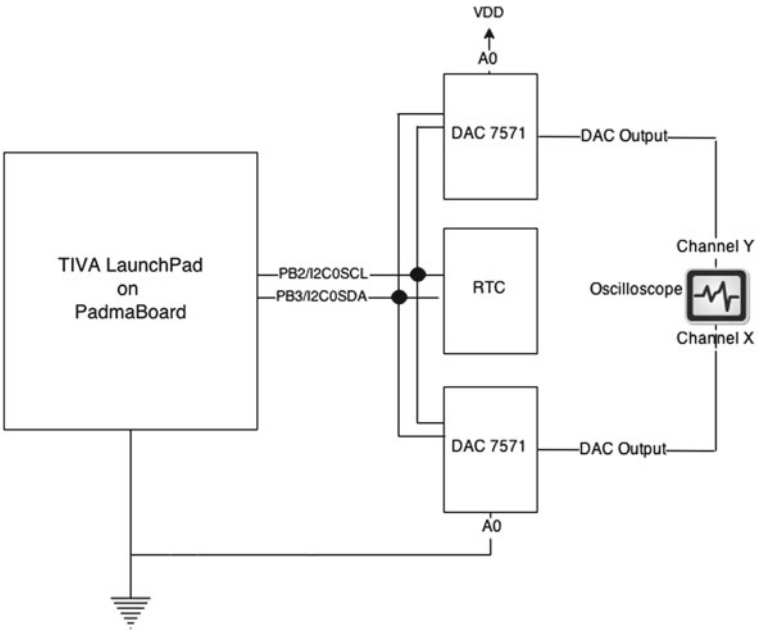

Fig. 11 Block diagram for oscilloscope clock

10 Experiment 55—Oscilloscope Clock

10.1 Objective

Read current time from RTC and display it on the oscilloscope as an analog clock.

10.2 Hardware Description

This experiment requires RTC to determine real time, and two DACs to generate the analog clock on the oscilloscope. Figure 11 shows the block diagram for hardware description of experiment. Analog clock on the oscilloscope can be simulated by using the same concept of experiment Lissajous Figures.

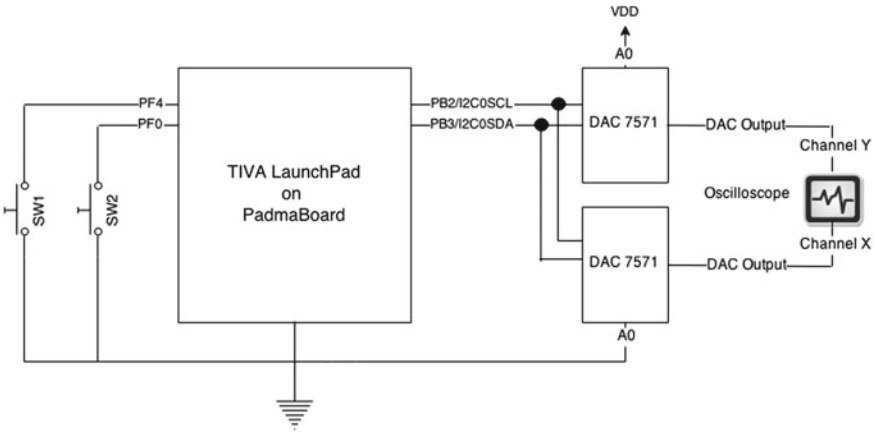

Fig. 12 Block diagram for classic Brick game

11 Experiment 56—Classic Brick Game

11.1 Objective

Develop the classic brick game on the oscilloscope. Use switch SW1 and SW2 on Tiva LaunchPad as control buttons.

11.2 Hardware Description

To perform this experiment two push buttons and two DACs are required. Two push buttons are on Tiva LaunchPad and 2 DACs are on PadmaBoard. The block diagram for the hardware description is shown in Fig. 12.

11.3 Experiment Tips

This experiment is similar to experiment implementation of Lissajous Figure but in this the image will be moving. Like to move a circle on the oscilloscope keep shifting the origin of circle after certain interval. In this way, figure with dynamic motion will be generated on the oscilloscope.

12 Experiment 57—Chaos

12.1 Objective

Plot a chaotic map on the oscilloscope. There are various chaotic maps having quite complex mathematical relations. Presence of floating point unit can be put to test by performing this experiment.

12.2 Hardware Description

The hardware requirement for this experiment is same as for the experiment Lissajous figures.

13 Experiment 58—Tiva on TV

13.1 Objective

Print custom message on the television using composite channel.

13.2 Hardware Description

The hardware of this experiment is pretty simple. It requires two resistors of 450 and 900 ohm and a female RCA connector. Generic resistor values available in market, close to mentioned values are 470 and 1000 ohm. These values of resistor will also work. Plug the cable between TV video input and RCA connector on PadmaBoard as video signal is generated in this experiment and want to observe it on TV. The block diagram of experiment is shown in Fig. 13. Television unit has internal 75 ohm of resistance, so combination of mentioned three resistance form 2-bit DAC. Also PD1 is multiplexed with potentiometer using the jumper, JP1. So place the jumper connection appropriately to select TV output.

13.3 Experiment Tips

The image consists of scan lines, each scan line is of 64 us. In beginning of the scan first horizontal sync pulse is sent for approximately 4 us. Then, some delay is

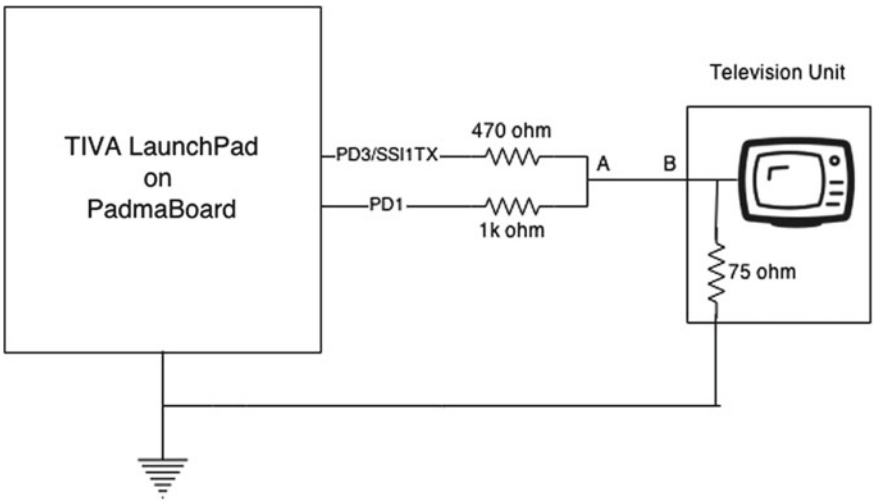

Fig. 13 Block diagram for Tiva on TV

generated for approximately 8 us, and data is transferred after that for the next 52 us. The time of horizontal pulse and delay varies from one TV to another as older ones will be having more delay. Use the systick timer interrupt which will be generating interrupts at every 64 us. Use the VGA font header file to have the character input. During sync keep both lines at logic low. For black, keep the PD1 at logic high and PD3 at logic low, and for white color keep both at logic high. Output on TV will be black and white, so for this PD1 will remain at logic high whereas PD3 will be toggling. Hence, to toggle PD3, use it as SPI data out to achieve high-toggling rate upto 25 Mbps. It is quite time-consuming experiment. Refer to detailed working and generation of TV video signals before performing this experiment. Also, optimize the code using option Optimize Size (-Os) in GCC compiler.

14 Experiment 59—Weather Channel

14.1 Objective

Use temperature sensor to sense the temperature and display temperature on television screen.

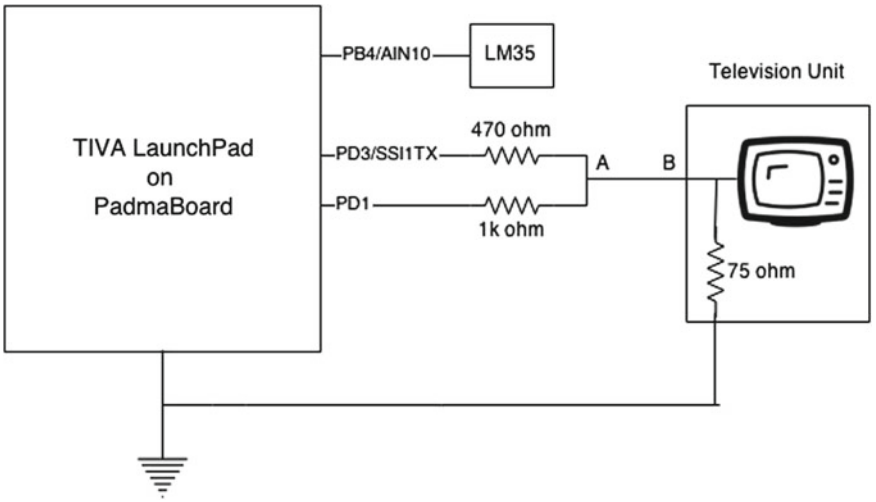

Fig. 14 Block diagram for weather channel

14.2 Hardware Description

This experiment requires a temperature sensor in addition to the hardware described in previous experiment. On PadmaBoard use LM35 as temperature sensor. The block diagram for hardware description of experiment is shown in Fig. 14. Also, place the jumper connections appropriately at JP1 and JP6.

15 Experiment 60—Hello SD Card!

15.1 Objective

Interface microSD card with Tiva LaunchPad and on pressing switch SW1, generate text file in microSD card. The text file should contain the "Hello World" message in it.

15.2 Hardware Description

To perform this experiment switch, microSD card holder, and a microSD card are required. MicroSD works on the SPI protocol. Already, a lot of experiments have

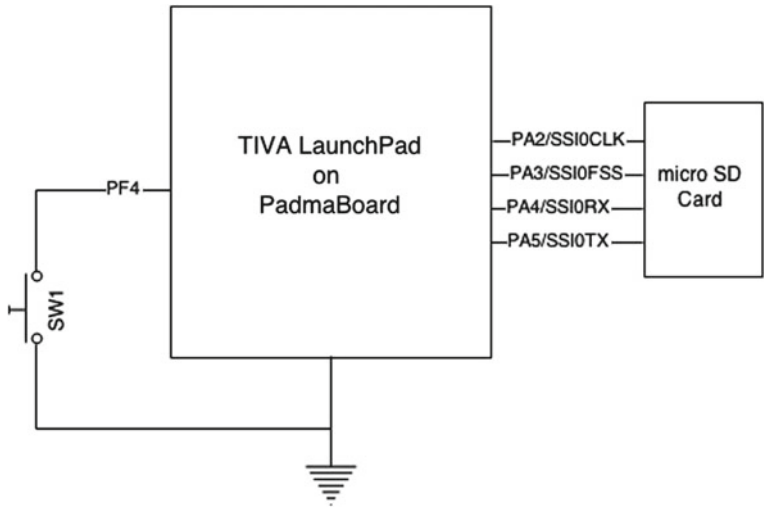

Fig. 15 Block diagram for hello SD card!

been discussed earlier on usage the switch, SW1 present on the Tiva LaunchPad. The hardware block diagram of the experiment is shown in Fig. 15.

15.3 Experiment Tips

Download the libraries of the microSD card, may require to tweak them a little bit to include the SPI function calls used in programming Tiva C Series microcontrollers. Also, change the optimization setting from none to "Optimize Size (-Os)". Also, may need to unchecked the "Do not use default libraries" and "Do not use the standard start files" option before building this project.

16 Experiment 61—Temperature Recorder

16.1 Objective

In this experiment whenever switch (SW1) is pressed, current ambient temperature will be stored in microSD card.

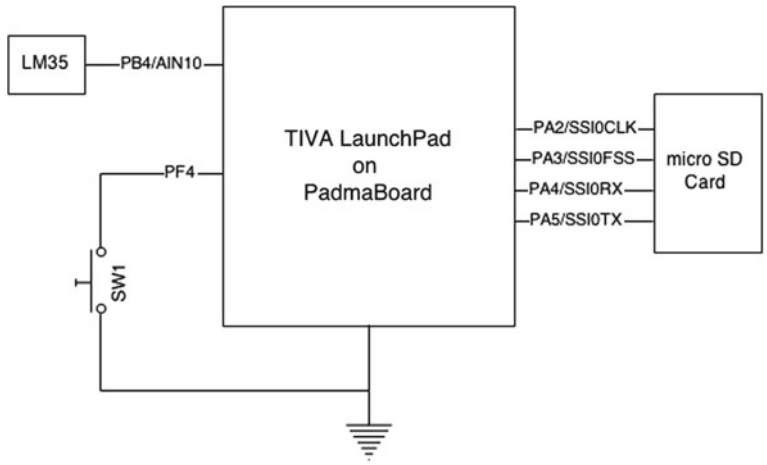

Fig. 16 Block diagram for temperature recorder

16.2 Hardware Description

To perform this experiment switch (SW1), LM35 to sense the temperature, and microSD card interface to store the data is required. The block diagram to perform this experiment is shown in Fig. 16. Place jumper connections appropriately on jumper, JP6 to select temperature sensor, LM35.

16.3 Experiment Tips

In this experiment, wait for SW1, to press, once it is pressed sample the analog channel corresponding to temperature sensor. Convert the sensor data into temperature in degree celsius value. Open temperature recorder text file and store the temperature value in that text file. After storing it in text file close the file and wait for the switch to be pressed again.

17 Experiment 62—Temperature Logger

17.1 Objective

Use RTC to keep logging temperature values in the text file in microSD card after certain interval. Use switch, SW1 to finish the experiment and close the file.

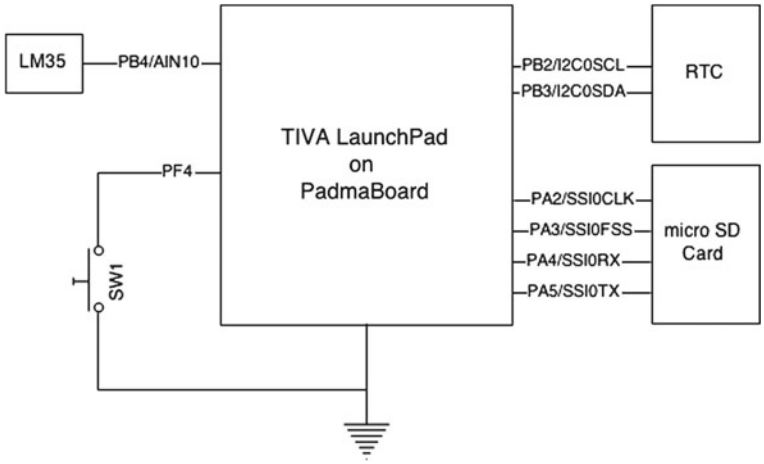

Fig. 17 Block diagram for temperature logger

17.2 Hardware Description

To perform this experiment use RTC for time stamping purpose, LM35 to sense temperature, microSD card interface to log temperature data in text file and switch SW1 to close the text file and finish the experiment. The block diagram to illustrate hardware associated with experiment is shown in Fig. 17.

18 Experiment 63—Voice Recorder

18.1 Objective

Start recording the audio input from microphone once the switch SW1 is pressed in microSD card. Once the switch SW2 is pressed stop the recording and start audio playback of recorded audio on 3.5 mm audio jack through the DAC 7571.

18.2 Hardware Description

This experiment requires two switches, SW1 and SW2, one electret microphone for audio input, microSD card interface to store audio input and DAC along with 3.5 mm audio jack at its output to play back the recorded audio. Figure 18 shows the block diagram of hardware description for experiment.

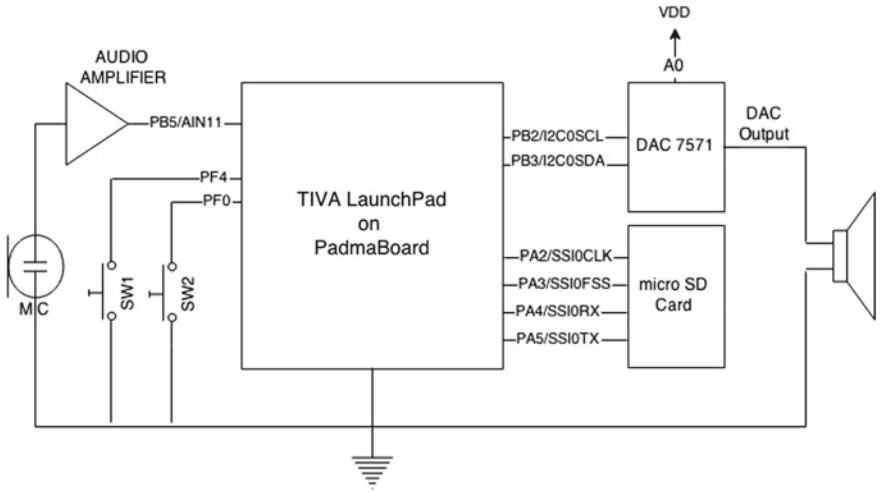

Fig. 18 Block diagram for voice recorder

18.3 Experiment Tips

Once the switch SW1 is pressed open a text file in microSD card and start recording the ADC values which are sampled from channel where microphone is connected. Recorded ADC values are stored in text file in microSD card. Try to sample the audio data from ADC as fast as possible. Once, switch SW2 is pressed start reading the analog values from the text file and pass those values to DAC7571, whose analog output is connected to 3.5 mm audio jack. Connect earphones or speaker on the audio jack for audio output.

19 Experiment 64—WAV Player

19.1 Objective

Store the audio files (.wav format) in microSD card and read these audio files and play it using 3.5 mm audio jack. Use two switches present on Tiva LaunchPad to move to next or previous song.

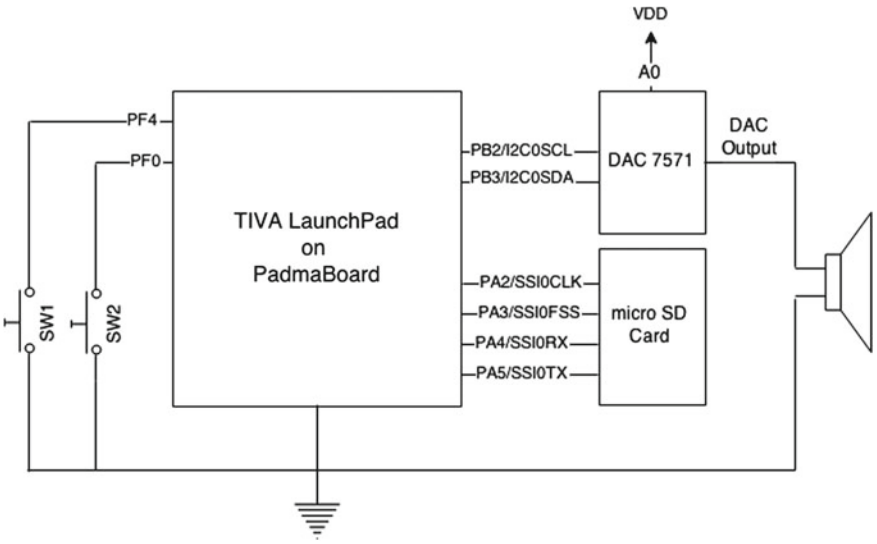

Fig. 19 Block diagram for voice recorder

19.2 Hardware Description

The hardware required to perform this experiment is 2 switches to select the previous or next audio file, microSD card to store the audio files (in .wav format) and DAC 7571 with its output connected to 3.5 mm audio jack to play audio files. Figure 19 shows the block diagram representing hardware description for the experiment.

19.3 Experiment Tips

Go through the format of .wav files thoroughly before performing this experiment. Read .wav file, in the beginning it contains useful information such as file size, length of the data, sample rate, bits per sample, etc and after that it contains the data. The initial information in the .wav file is important as it helps in reading the data. Also, DAC used is serial DAC so for better response use the audio files with sample rate of 22,050 Hz. Use the I2C in fast mode plus or high-speed mode.

Chapter 16
User Input and Output Devices

In this chapter more input and output user interfaces are discussed. In earlier chapters, for user interface UART terminal on host PC and switches were used. So, this chapter will discuss a bit more complex interfaces like 16×2 LCD, 16 key keypad and PS/2 based keyboard. LCD and 16 key Keypad are driven using serial shift registers. So, that both of these can be controlled by using just the 5 GPIO's. Having, such interfaces helps in developing standalone projects, as the user may not need to use host terminal to display messages and for user input. The user can display the messages on LCD and give input using 16 key keypad or PS/2 keyboard.

1 Serial LCD and Keypad

1.1 Shift Registers

Shift register used 74HC595 which is a 8-bit serial in, serial out or parallel out shift register. The 8-bit serial input can be observed as 8-bit parallel output on 74HC595. Serial out means serial input data will traverse through all the shift register stages before reaching to serial output. Since, it is a 8-bit shift register so it will take 8 clock cycles for input serial data to reach the serial output. Serial output can be used to cascade the two 8-bit shift register to generate 16-bit parallel output. The inputs to shift register are: serial data, serial clock, and latch clock. With each serial clock, serial data bit is shifted in shift register and with each latch clock, data is latched to the parallel output port of shift register.

As, two 74HC595 are cascaded on Serial LCD and 16 key keypad board as shown in Fig. 13. So, after each 16-bit of serial data transfer a rising edge clock pulse is generated at latch clock to observe the 16-bit serial input on the two 8-bit parallel output ports of two shift registers. By using this method, only 3 GPIOs are required to control 16 output lines. As, it is saving a lot of GPIOs, but is reducing the performance of designed system. As, if 16 output lines are available on microcontroller then data

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_16

flow will be parallel but here data flow is serial in chunks of 16-bits. Out of 16 output lines from shift registers, six are used to control the LCD, 8 are used by the 16 Key Keypad and 2 are left unconnected.

1.2 LCD Interfacing

The 16×2 LCD has 8 data lines, 3 control lines, 1 contrast adjustment line, and 4 supply lines. Among the 4 supply lines, 2 are for LCD and 2 are for backlight. LCD can be used in 8-bit mode or 4-bit mode. In 4-bit mode data and instructions are send in nibbles(packet of 4-bits), first the higher nibble is send and then the lower nibble is send. In this mode only 4 data lines (D4, D5, D6, and D7) are required. 3 control lines are Enable, RS and RW. RS is register select line, this line is used to select whether the data on data bus is an instruction or data. RW is read or write select line, but since the LCD is used for display, therefore RW line can fixed to a certain logic to set it in write mode only. Contrast of display of LCD varies with the voltage at contrast adjustment pin. So, a potentiometer is used to vary the voltage between 0 and +5 V at contrast adjustment line.

1.3 16 Keys Keypad

As shown in Fig. 13 of Chap. 4, there are 16 switches, which are connected in 8×2 fashion. 8×2 means there are 2 scan lines and 8 output lines. Generate logic high on all 8 output lines through shift registers. Since, all switches are pulled down to ground, so when no switch is pressed scan lines will read logic low. Whenever a switch is pressed, the corresponding scan line will read as logic high. Then, enable each output line one by one and check the scan line each time to determine which switch is pressed. If two switches are pressed at same time and they belong to the same scan line then use the priority encoding scheme.[1]

2 PS/2 Keyboard

Enabling the use of Keyboard in the projects enhances the user input experience to a great extent. PS/2 ports can be of 6 pin or 5 pins from which only 4 are used and others are not connected. PS/2 port have Data pin, clock pin, and 2 supply pins. Data

[1] If the multiple switches belonging to the same scan line are pressed, then switch with high priority will be considered. Let scan line 0 detected logic high, it implies that switch (or switches) belonging to this line is pressed. Then, start checking for each switches from the MSB of the output bus, once the switch is determined then do not check the lower output lines.

and clock line are made bidirectional by using 4 GPIOs as shown in Fig. 14. In case of keyboard, Tiva LaunchPad will be the host reading data from the device (keyboard) whenever key is press. Tiva LaunchPad can also transmit the data to the keyboard example in case of turning on Caps Lock LED on keyboard.

2.1 Protocol

There is two type of communication that exits device to host and host to device communication. In case of interfacing with PS/2 keyboard, keyboard is device and Tiva LaunchPad is host. This communication is half duplex that is at a particular time either of host or device can transmit. As per pin mapping in PadmaBoard, configuration of 4 GPIOs as per various situations is tabulated below:

Each data byte is transferred in a frame of 11–12 bits. This data frame is similar to data frame used for UART communication. The bits in data frame are 1 start bit, 8 data bits, 1 parity bit, 1 stop bit, and 1 acknowledge bit (this bit will be send only in case of host to device communication). The clock is always generated by the device only. When host needs to send the data then it must pull the clock low to inhibit the existing communication. After that, host must pull data line low and then release the clock line to generate the "Request to send" state to request the device to generate clock pulses so that host can transmit the data. When the data is sent from the device to host, it is read on the falling edge of clock. And, when the data is sent from host to device it is read on the rising of clock (Table 1).

2.2 Scan Codes

Keyboard has its own processor to monitor all the keys, whenever a key is pressed scan code corresponding to pressed key is sent over the PS/2 channel. The scan codes are of two types: make code and break code. Make code is sent when key is pressed and break code is sent when key is released. For every key there is a unique make

Table 1 GPIO configuration for PS/2 port on PadmaBoard

Configuration of Tiva LaunchPad	Host to device communication	Device to host communication
Tiva LaunchPad as host	Configure PC4 as input for clock and PC7 as output for data	Configure PC4 and PC5 as input for clock and data, respectively
Tiva LaunchPad as device	Configure PC6 as output for clock and PC5 as input data	Configure PC6 and PC7 as output for clock and data, respectively

Table 2 Example scan codes

Key	Make code	Break code
"R"	'h2D' or 'd45'	'hF0', 'h2D' or 'd240', 'd45'
"G"	'h34' or 'd52'	'hF0', 'h34' or 'd240', 'd52'
"B"	'h32' or 'd50'	'hF0', 'h32' or 'd240', 'd50'
"R CTRL"	'hE0', 'h14' or 'd224', 'd20'	'hE0', 'hF0', 'h14' or 'd224', 'd240', 'd20'

code and break code. There is no relation between the ASCII value of character on the key and the make code (or break code). While performing the experiments based on this refer to the table for make codes and break codes for keys. Make codes are 1 byte long for most keys and for extended keys length of make codes is 2 bytes. In break codes the byte 'hF0'(240 in decimal) is appended to the make code. So, the length of break code is 2 bytes for most keys and 3 bytes for extended keys. The example of make codes and break codes is shown in Table 2.

When the length of scan code is 2 bytes or more than 2 bytes example break code of "right ctrl" is 'hE0', 'hF0', and 'h14', then these bytes will send using three different data frames, first frame will contain the byte 'hE0', next frame will contain the byte "hF0" and the last frame will have 'h14'.

3 Experiment 65—Hello LCD!

3.1 Objective

Print "Hello World!!" message on 16×2 LCD using shift registers.

3.2 Hardware Description

To perform this experiment shift registers and 16×2 LCD are required. Serial LCD and Keypad Board uses two 74HC595 shift registers to drive 16×2 LCD and 16 Key Keypad. Plug the serial LCD and Keypad board to the 5×2 box connector on PadmaBoard as shown in Fig. 1. PE2 is used to transfer the serial data to the shift register, PE3 is used to generate clock to synchronize shift of serial data bits, and PE4 to latch data on the output port of shift register.

Though the Serial LCD and Keypad is a combined board on which PE0 and PE1 are the scan lines of Keypad. So, these GPIOs can be left unused as in this experiment Keypad is not used.

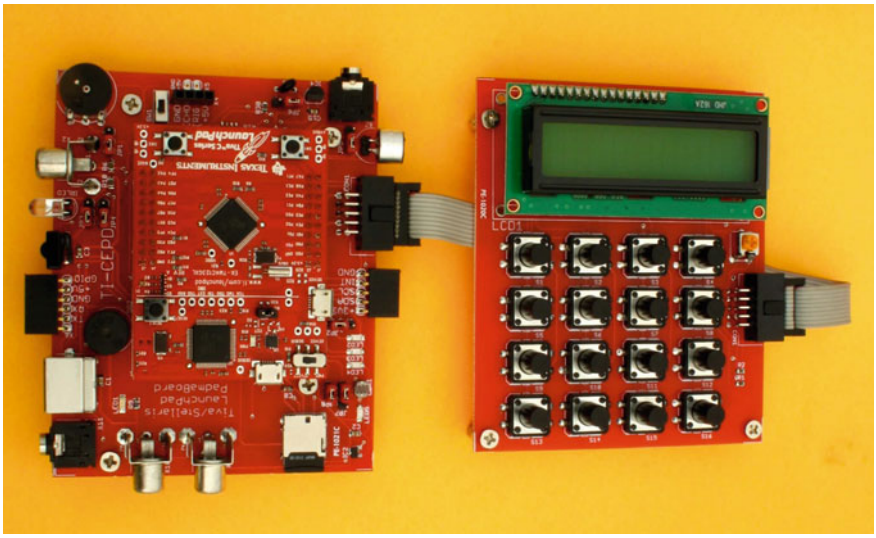

Fig. 1 Connection of serial LCD-keypad board and PadmaBoard

3.3 Program Flow

While using LCD, delay values between the instructions are most important. Values of delays may vary from one LCD driver to another. The shift registers are FIFO, so when the data is transferred with LSB first, LSB will appear at MSB of the parallel output bus of shift register. So, need to reverse the order commands before transmitting on the shift registers. Example if 0×02('b0000_0010') is transmitted serially to the shift register, then output observe on the parallel output bus of the shift register is 0×40('b0100_0000').[2] The program flow for experiment is listed below:

1. Enable the system clock and GPIO Port E.
2. Configure GPIO PE2, PE3 and PF4 as output.
3. Initialize the LCD in 4-bit mode and clear the LCD.
4. Transmit the data corresponding to text "Hello World!!" in data mode to the LCD.

Figure 2 shows the program flow for the experiment.

3.4 Program Code

The program code to perform the experiment is mentioned below. Program code is well commented for better understanding purpose.

[2]This can be avoided by transmitting the MSB first. In the program code data is shifted out with LSB first.

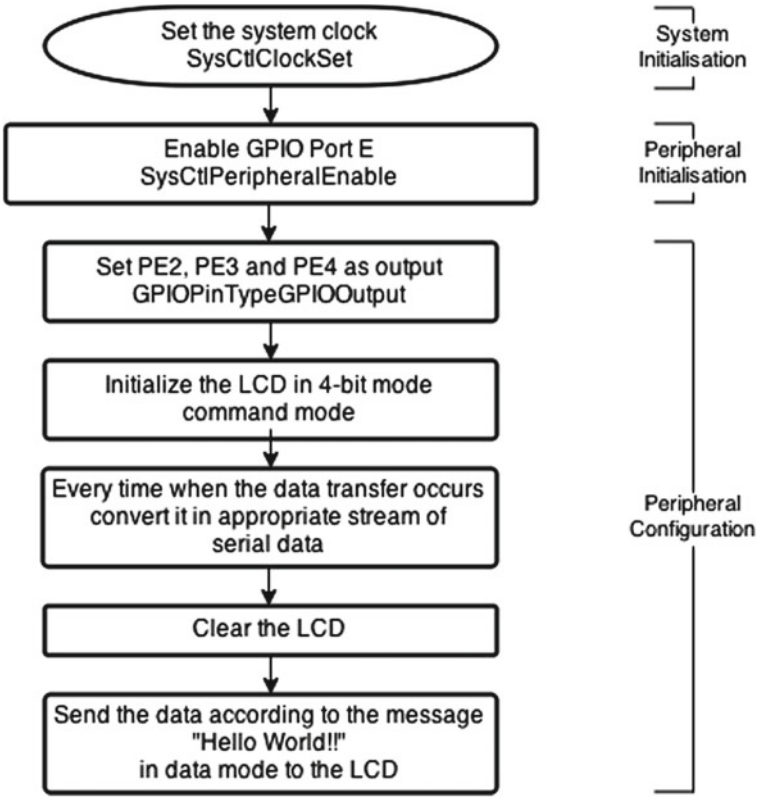

Fig. 2 Program flow for hello LCD!

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "inc/hw_gpio.h"
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "inc/hw_sysctl.h"
```

```
#include "driverlib/sysctl.h"

#define COMMAND 0x00FF
#define DATA    0x04FF
#define EN      0x0800

/* Output numBits bits of data out through a
 * 74HC595 shift register and latch to output */
void ShiftOut(unsigned long data, unsigned int numBits)
{
int i;
GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_4, 0);
for(i = 0; i < numBits; i++)
{
int bit = data % 2;
GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_3, 0);
GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, bit<<2);
GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_3, GPIO_PIN_3);
data = data / 2;
}
GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_4, GPIO_PIN_4);
}

unsigned long nibble1, nibble2;

void LCDSend(unsigned int data, unsigned long mode)
{
int bits[8], i;
for(i = 0; i < 8; i++)
{
bits[i] = data % 2;
data = data /2;
}
nibble1 = (bits[7] + (2*bits[6]) + (4*bits[5]) + (8*bits[4]))*4096;
nibble2 = (bits[3] + (2*bits[2]) + (4*bits[1]) + (8*bits[0]))*4096;
nibble1 = nibble1 + mode;
nibble2 = nibble2 + mode;

ShiftOut(nibble1+EN,16);
ShiftOut(nibble1,16);

ShiftOut(nibble2+EN, 16);
ShiftOut(nibble2,16);
SysCtlDelay((SysCtlClockGet()/10000)*2);
}

/* Initializes the LCD module in 4-bit mode
 * Bit Order - D4,D5,D6,D7,EN,RS,X,X */
void InitLCD(void)
```

```
{
/* Initial Reset Sequence */
ShiftOut(0xFFFF,16);
SysCtlDelay((SysCtlClockGet()/1000)*20);

ShiftOut(0xC8FF,16);
ShiftOut(0xC0FF,16);
SysCtlDelay((SysCtlClockGet()/1000)*10);

ShiftOut(0xC8FF,16);
ShiftOut(0xC0FF,16);
SysCtlDelay((SysCtlClockGet()/1000)*1);

ShiftOut(0xC8FF,16);
ShiftOut(0xC0FF,16);
SysCtlDelay((SysCtlClockGet()/1000)*1);

/* Set 4-bit mode */
ShiftOut(0x48FF,16);
ShiftOut(0x40FF,16);
SysCtlDelay((SysCtlClockGet()/1000)*1);

/* Send the Function set command with appropriate bits.
 * Set Data length to 4-bits, No. of lines to 2 */
LCDSend(0x28, COMMAND);
SysCtlDelay((SysCtlClockGet()/1000)*1);

/* Send Display On/Off Control with appropriate bits.
 * Set Display On, Set Cursor to display and turn on
 * cursor character blink */
LCDSend(0x0F, COMMAND);
SysCtlDelay((SysCtlClockGet()/1000)*1);
}

void LCDprint(char* text)
{
while(*text)
LCDSend(*text++, DATA);
}

void LCDclear(void)
{
LCDSend(0x01, COMMAND);
SysCtlDelay((SysCtlClockGet()/1000)*1);
}

int main(void)
{
/* Set the clock to 20Mhz from the crystal of 16MHz using PLL */
SysCtlClockSet(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Enable the GPIO Port E*/
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

/* Configure PE2, PE3 and PE4 as output */
GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_2 | GPIO_PIN_3 |
GPIO_PIN_4);

InitLCD();
LCDclear();
LCDprint("Hello World!!");
while(1)
  {
  }
}
```

4 Experiment 66—PS/2 Keyboard

4.1 Objective

Use PS/2 keyboard to control the RGB LED on Tiva LaunchPad using 'R', 'G', and 'B' key of the PS/2 keyboard.

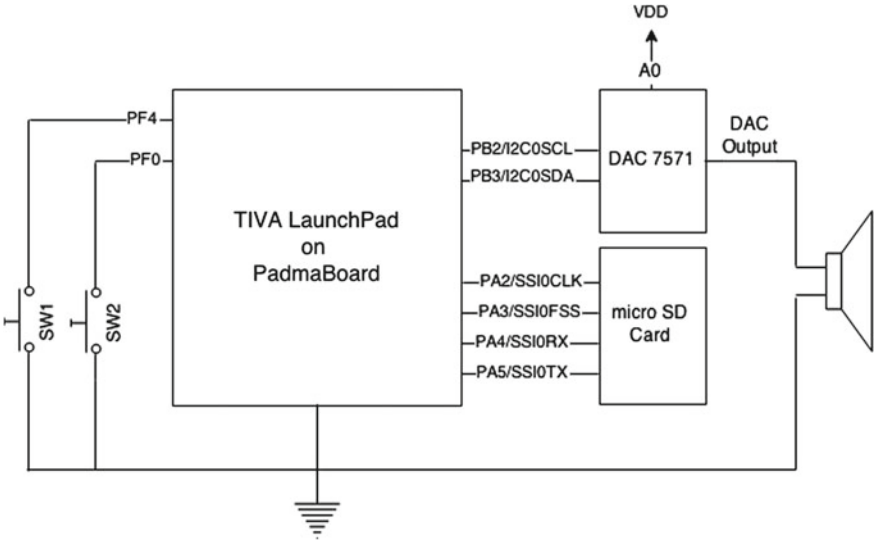

Fig. 3 Block diagram for PS/2 keyboard

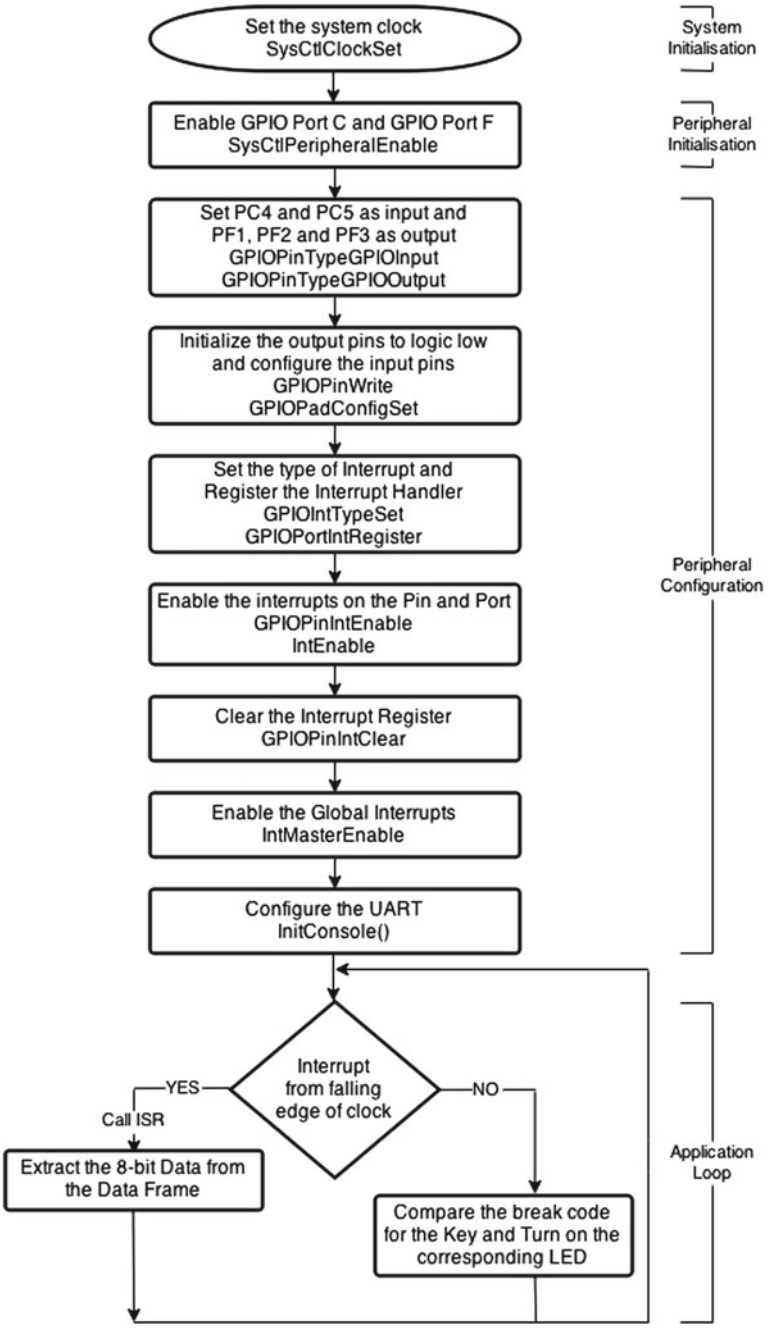

Fig. 4 Program flow for PS/2 keyboard

4.2 Hardware Description

To perform this experiment PS/2 keyboard and PS/2 interface with Tiva LaunchPad is required. Since Tiva LaunchPad is host and keyboard is device and device is transmitting data to host so from Table 1 PC4 and PC5 are configured as input pins. Also, RGB LED is connected to the GPIOs PF1, PF2, and PF3. The hardware description of experiment is shown in Fig. 3.

4.3 Program Flow

For the data sent from device to host can be read at falling edge. So, configure the interrupt at falling edge of clock, i.e., at GPIO PC4. In the ISR (which is called at each falling edge of the clock) read the data line. The algorithm to perform the experiment is enumerated below:

1. Enable the system clock, GPIO Port C and F.
2. Configure the pins PC4, PC5 as input and PF1, PF2, and PF3 as output and initialize output pins to logic low.
3. Configure and register the interrupt handler to PC4 and enable the interrupts.
4. In the interrupt ISR which is called on the falling edge of the clock, extract the data byte from the 11-bit data frame.
5. Check for the break code, if the key pressed is “R”, “G”, or “B” then turn on the corresponding LED. If it is ‘W’ then turn on all LEDs generating white color on the RGB LED and for any other key pressed turn off all the LEDs.

Figure 4 shows the program flow for the experiment.

4.4 Program Code

The program code to perform the experiment is mentioned below. Program code is well commented for better understanding purpose.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"
```

```
/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

/*  Macros that define the interrupt assignment */
#include "inc/hw_ints.h"

/* Prototypes for the NVIC Interrupt Controller Driver. */
#include "driverlib/interrupt.h"

/* Defines and Macros for UART API */
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

void InitConsole(void)
{
/* Initializing GPIO Port A and UART Module 0 */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

/* Make the UART pins be peripheral controlled */
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

/* Initialize the UART for console I/O */
UARTStdioConfig(0, 115200, SysCtlClockGet());
}

unsigned int g_ulcount = 0, array[8], mul=1,sum=0;
unsigned int break_code[8] = {0,0,0,0,1,1,1,1};
unsigned int i,cmp;
/* Interrupt subroutine for the falling edge interrupt at
 * PC4 or CLK input */
void Pin_Int(void)
{
GPIOIntClear(GPIO_PORTC_BASE, GPIO_PIN_4);
g_ulcount++;

if(g_ulcount==1)
  {
  sum=0;
  mul=1;
  }
```

```
else if((g_ulcount>=2)&&(g_ulcount<=9))
  {
  if(GPIOPinRead(GPIO_PORTC_BASE,GPIO_PIN_5)!=0)
    {
    array[g_ulcount-2]=1;
    sum=sum+mul;
    }
  else
    array[g_ulcount-2]=0;
  mul=mul*2;
  }
else if(g_ulcount==11)
  g_ulcount=0;
}

/* Function to compare two arrays */
int str_cmp(unsigned int arr1[8], unsigned int arr2[8])
{
cmp = 0;
for(i=0; i<8; i++)
  if(arr1[i] != arr2[i])
    cmp=1;
return(cmp);
}


int main(void)
{
/* Set the clock to 80Mhz from the crystal of 16MHz using PLL */
SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

/* Set the clock for the GPIO Port F and Port C */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

/* Set the type of the GPIO Pins */
GPIOPinTypeGPIOInput(GPIO_PORTC_BASE, GPIO_PIN_4|GPIO_PIN_5);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);

/* Initializing output pins to default value */
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3,0);

/* Configure GPIO pad with internal pull-up enabled */
GPIOPadConfigSet(GPIO_PORTC_BASE, GPIO_PIN_4|GPIO_PIN_5,
GPIO_STRENGTH_2MA,GPIO_PIN_TYPE_STD_WPU);

/* Set interrupt triggering sequence */
GPIOIntTypeSet(GPIO_PORTC_BASE, GPIO_PIN_4, GPIO_FALLING_EDGE);

/* Register Interrupt to call Interrupt Handler */
```

```
GPIOIntRegister(GPIO_PORTC_BASE, Pin_Int);

/* Enable interrupts on selected pin */
GPIOIntEnable(GPIO_PORTC_BASE, GPIO_PIN_4);

/* Enable interrupts on selected port */
IntEnable(INT_GPIOC);

/* Clear interrupt register */
GPIOIntClear(GPIO_PORTC_BASE, GPIO_PIN_4);

/* Enable global interrupts */
IntMasterEnable();

/* UART config */
InitConsole();

IntMasterEnable();

while(1)
  {
  /* Check for break code */
  if(str_cmp(array, break_code) == 0)
    {
    /* Delay is generated so that last byte of the break code can be
     * read during this time which is unique for each key */
    SysCtlDelay(SysCtlClockGet()/300);

    /* Printing decimal value of make code */
    UARTprintf("key dec value::%04d\n",sum);

    /* Check for scan code corresponding to key 'R' then turn on
     * the Red LED */
    if(sum==45)
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3,
      GPIO_PIN_1);

    /* Check for scan code corresponding to key 'B' then turn on
     * the Blue LED */
    else if(sum==50)
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3,
      GPIO_PIN_2);

    /* Check for scan code corresponding to key 'G' then turn on
     * the Green LED */
    else if(sum==52)
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3,
      GPIO_PIN_3);

    /* Check for scan code corresponding to key 'W' then turn on
     * all LED's */
    else if(sum==29)
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3,
      GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);

    /* Else, for any other key pressed turn off all LEDs */
    else
```

```
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3,0);
      }
    }
  }
```

5 Experiment 67—Scrolling Display

5.1 Objective

Print the scrolling message "Hello World!!" on 16×2 LCD, scrolling from right to left.

5.2 Hardware Description

The hardware description to perform this experiment is similar to experiment Hello LCD!!. To make this experiment a bit more interesting add switch to control the direction of scrolling of the message.

6 Experiment 68—Calculator

6.1 Objective

Use LCD and 16 key Keypad and imitate the functionality of calculator performing basic operation such as addition, subtraction, multiplication, and division.

6.2 Hardware Description

The hardware description to perform this experiment is similar to experiment Hello LCD!!. But unlike experiment Hello LCD, keypad is used in this experiment. So, configure the scan lines (GPIO PE0 and PE1) as digital inputs. To make this experiment a bit more user friendly or to take this experiment to next level use PS/2 keyboard for user input.

7 Experiment 69—VU Meter

7.1 Objective

Sample the sound from audio input either through 3.5 mm audio jack or the microphone and plot the intensity of sound on 16×2 LCD.

7.2 Hardware Description

To perform this experiment use 16×2 LCD for display purpose and electret microphone or 3.5 mm audio jack for audio input. Electret microphone and audio jack (for audio input) is connected to Analog Channel 11(AIN11) multiplexed with GPIO PB5 through a jumper, JP5. Refer Chap. 4, Sect. 5 for proper placement of jumper to select between the input either through microphone or through audio jack. LCD is accessible using shift registers as discussed in previous experiments. Figure 5 refers the hardware description of experiment.

7.3 Experiment Tips

Sample the sound levels from analog channel connected and plot it on LCD in form of horizontal bar. The length of bar will be proportional to the intensity of sound. May require to look into generation custom characters on the 16×2 LCD.

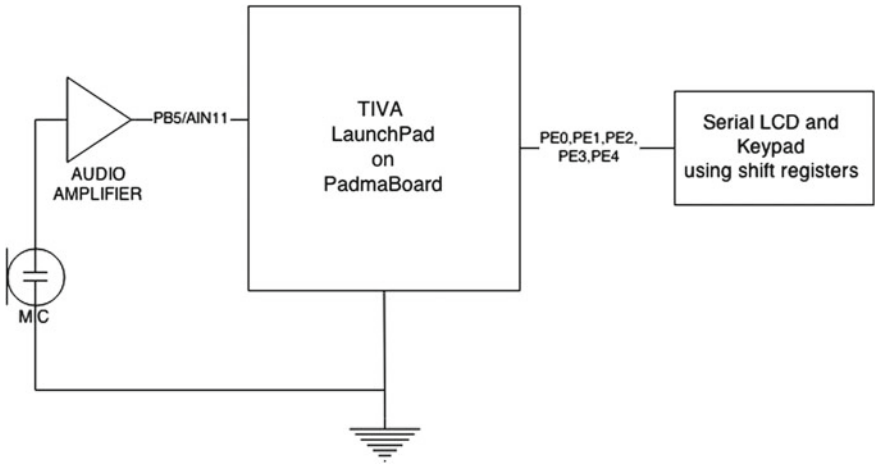

Fig. 5 Block diagram for PS/2 keyboard

8 Experiment 70—Digital Filters on LCD

8.1 Objective

Implement three digital filters: low pass, band pass and high pass filter for audio input. Depict the output of 3 filters by varying the height of vertical bars. Each bar will represent each filter.

8.2 Hardware Description

The hardware requirement to perform this experiment is similar to experiment VU Meter.

9 Experiment 71—Spectrometer

9.1 Objective

Display the sound spectrum of audio input on 16×2 LCD by implementing fast fourier transform (FFT) on the audio input.

9.2 Hardware Description

The hardware requirement to perform this experiment is same as experiment VU meter. Prior knowledge of the signal processing will be a great help in performing this experiment.

10 Experiment 72—Digital Clock

10.1 Objective

Display the current date and time on 16×2 LCD and be able to set alarm on RTC using PS/2 keyboard.

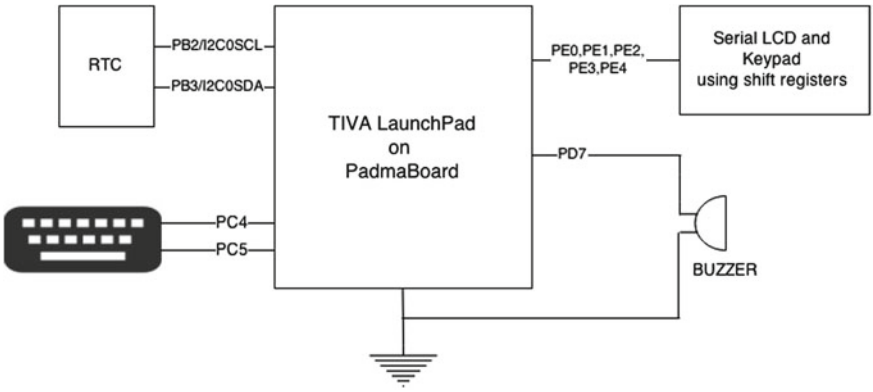

Fig. 6 Block diagram for digital clock

10.2 Hardware Description

This experiment requires RTC to keep the track of date and time, LCD and PS/2 keyboard for user interface, and buzzer to sound the alarm. the hardware description of the experiment is shown in Fig. 6. Use the PS/2 keyboard to configure alarm on RTC and use LCD for display purpose. To use RTC interrupt at alarm time go through the schematic carefully, as it has been multiplexed with the scan line of the keypad. For a falling edge on the RTC interrupt line, it will become rising edge on the scan line.

11 Experiment 73—Text Editor

11.1 Objective

Use PS/2 Keyboard to write text on the LCD, and save the text as text file in the microSD card.

11.2 Hardware Description

To perform this experiment Keyboard is required for user input, LCD is required for display purpose and microSD card interface to save the text file. The hardware description for the experiment is shown in Fig. 7.

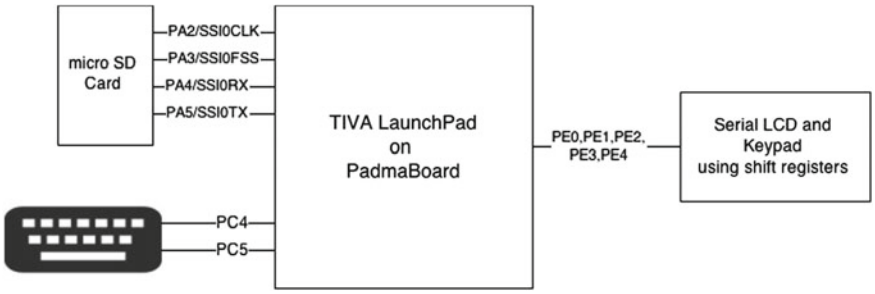

Fig. 7 Block diagram for text editor

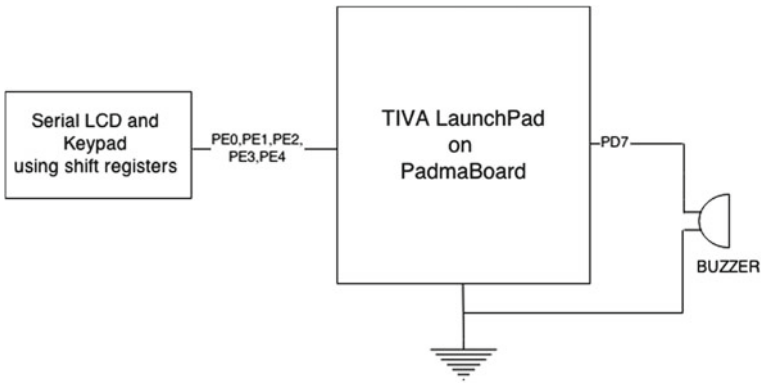

Fig. 8 Block diagram for piano

12 Experiment 74—Piano

12.1 Objective

Use keypad to generate various tones corresponding to the Piano Keys using buzzer.

12.2 Hardware Description

This experiment requires input from the keypad so configure the pins PE2, PE3, and PE4 as output and PE0 and PE1 as input lines to access serial keypad on PadmaBoard. PE0 and PE1 are scan lines of 8×2 keypad. Also, on PadmaBoard buzzer is connected to GPIO PD7. The hardware description for experiment is shown in Fig. 8.

12.3 Experiment Tips

There are 16 keys on keypad which are enough to mimic the basic keys of piano. For each key there is square wave (or PWM with 50% duty cycle) of particular frequency need to be generated at buzzer pin. Use the timers to generate the 50% PWM of frequencies dependent on the key pressed for the buzzer pin.

13 Experiment 75—Tone Recorder

13.1 Objective

Use the keypad to generate various tones and store simultaneously in the SRAM of Tiva LaunchPad, so that they can be played later on.

13.2 Hardware Description

This hardware requirement for this experiment is same as the previous experiment Piano.

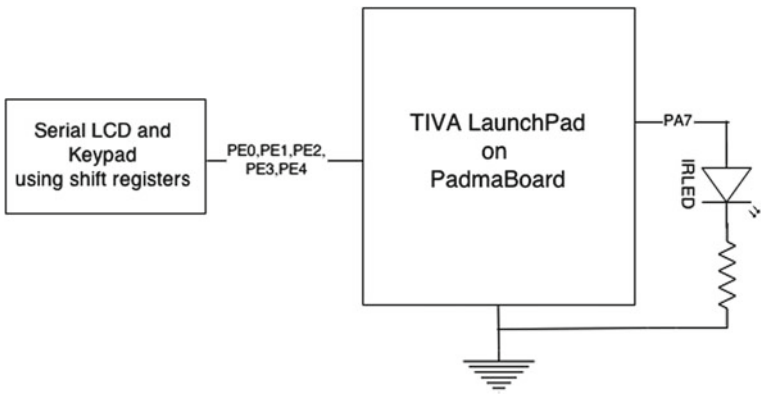

Fig. 9 Block diagram for universal remote

14 Experiment 76—Universal Remote

14.1 Objective

Use keypad to generate the mixture of various TV remote signals, AC remote signals and sound system remote signals and transmit them using IR LED and simultaneously display the signal name on LCD. So, in practical application this experiment can be used to control many household appliances (remote based) through one common remote.

14.2 Hardware Description

To perform this experiment use LCD for display purpose, keypad for data input and IR LED to generate signals. The block diagram of hardware description is shown in Fig. 9.

Chapter 17
Tiva C Series Based Standalone Projects

This chapter focuses on developing of standalone projects based on Tiva C Series of microcontrollers. As, discussed in Chap. 5 about the breakout board of Tiva C Series microcontroller, same breakout board is used in this chapter to perform experiments regarding the standalone projects. There are not much changes required in program code while using other Tiva C Series microcontrollers to perform the experiments, as the API function calls are common for all microcontrollers. Though, it is required to define the corresponding microcontroller in toolchain setup while performing the experiments.

1 Experiment 77—Controlled Blinky

1.1 Objective

Blink a LED on Tiva breakout board with a variable delay depending on which switch is pressed.

1.2 Hardware Description

To perform this experiment, Tiva LaunchPad or JTAG Programmer is required to program the Tiva breakout board. Make the connections between Tiva breakout board and Tiva LaunchPad as explained in Chap. 5. In addition to this, one LED and two switches are required which are present on the Tiva breakout board. The microcontroller used on the TIVA breakout board is TM4C1231H6PZ. Figure 1 shows the hardware description of experiment in block diagram format.

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2_17

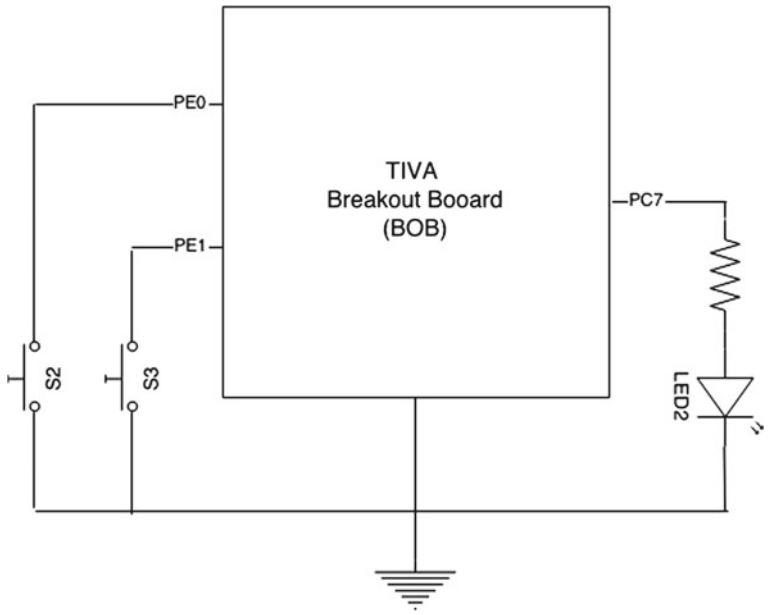

Fig. 1 Block diagram for controlled blinky

1.3 Program Flow

First define the corresponding microcontroller (TM4C1231H6PZ) in toolchain setup. So, in the toolchain setup, in the GCC C compiler under Preprocessor add PART_TM4C1231H6PZ instead of PART_TM4C123GH6PM in defined symbols. Blink the LED connected to GPIO PC7 with a delay of 500 ms. Use the switch SW2 connected to GPIO PE0 to decrease the blink delay to 100 ms, and switch SW3 connected to GPIO SW3 to increase the blink delay to 1 s. Since external oscillator used on breakout board is 20 MHz, corresponding parameter is set in the SysCtlClockSet function. The algorithm to perform the experiment is illustrated below:

1. Enable the system clock and GPIO Port C and Port E.
2. Configure the pin, PC7 as output and pin PE0 and PE1 as input.
3. Read the switch SW2 if it is pressed blink the LED with a delay of 100 ms, if switch SW3 is pressed blink the LED with delay of 1 s and none of the switch is pressed blink the LED with delay of 500 ms.

Program flow of the experiment is illustrated in the Fig. 2.

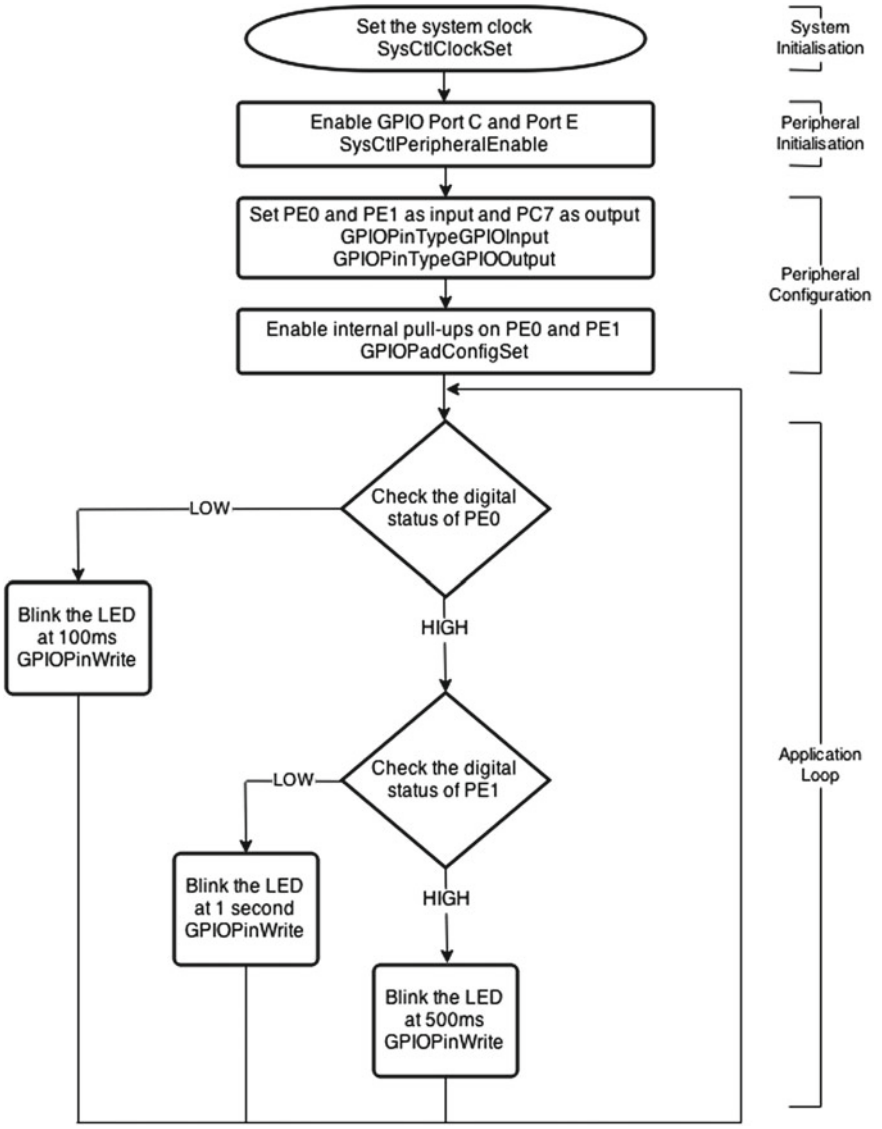

Fig. 2 Program flow for controlled blinky

1.4 Program Code

The complete C program for experiment is given below. The program is well commented for ease in understanding it.

```
/* Defines boolean and integer data types */
#include <stdbool.h>
#include <stdint.h>

/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

int main(void)
{
/* Set the clock to directly run from the crystal at 20MHz */
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_20MHZ);

/* Set the clock for the GPIO Port C and Port E */
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

/* Set the type of the GPIO PC7 as Output */
GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_7);
GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7,0);

/* Set the type of GPIO PE0 and PE1 as Input */
GPIOPinTypeGPIOInput(GPIO_PORTE_BASE, GPIO_PIN_0|GPIO_PIN_1);
GPIOPadConfigSet(GPIO_PORTE_BASE, GPIO_PIN_0|GPIO_PIN_1,
GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);

while(1)
 {
 /* Reading the state of switch SW2
  * If read state is logic low implies switch is pressed
  * Blink the LED with the delay of 100 milliseconds */
  if(GPIOPinRead(GPIO_PORTE_BASE,GPIO_PIN_0)==0)
    {
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7,GPIO_PIN_7);
    SysCtlDelay(SysCtlClockGet()/30);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7,0);
    SysCtlDelay(SysCtlClockGet()/30);
    }

  /* Reading the state of switch SW3
   * If read state is logic low implies switch is pressed
   * Blink the LED with the delay of 1 second */
  else if(GPIOPinRead(GPIO_PORTE_BASE,GPIO_PIN_1)==0)
    {
```

```
      GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7,GPIO_PIN_7);
      SysCtlDelay(SysCtlClockGet()/3);
      GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7,0);
      SysCtlDelay(SysCtlClockGet()/3);
      }

    /* No switch is pressed
     * Blink the LED with the delay of 500 milliseconds */
    else
      {
      GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7,GPIO_PIN_7);
      SysCtlDelay(SysCtlClockGet()/6);
      GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7,0);
      SysCtlDelay(SysCtlClockGet()/6);
      }
    }
}
```

2 Experiment 78—UART—Based LED Control

2.1 Objective

Use the input from host PC through UART to control the state of LED on Tiva breakout board.

2.2 Hardware Description

To perform this experiment, Tiva LaunchPad will be used to program and power (+3.3 V and GND) Tiva breakout board. Also, Tiva LaunchPad will used as

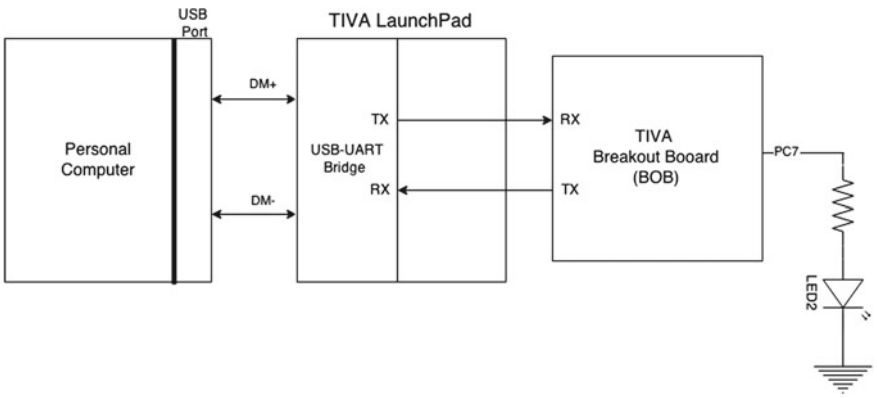

Fig. 3 Block diagram for UART-based LED control

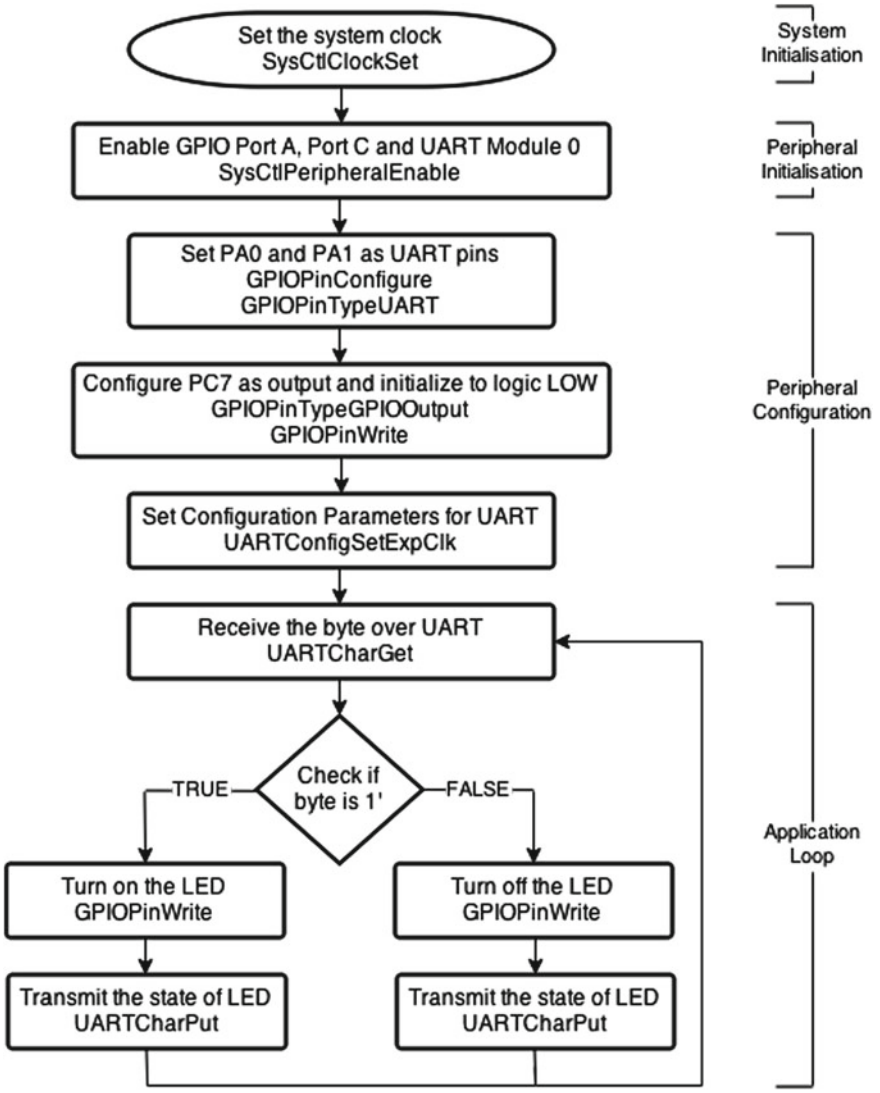

Fig. 4 Program flow for UART-based LED control

USB-UART bridge between Tiva breakout board and host PC. Make the connections between Tiva LaunchPad as explained in Chap. 5. The GPIOs which are available on the box connector are PA0 and PA1 and are multiplexed with the UART Module 0 pins. In addition to above experiment an LED which is present on Tiva breakout board connected to PC7 is required. Figure 3 describes the hardware description for experiment. In the figure only UART connections are shown because once breakout board is programmed correctly as per the needs of experiment, then JTAG connections can be removed.

2.3 Program Flow

Since UART Module 0 Rx and Tx pins are available on the box connector use this module to perform the experiment. The algorithm for the experiment is illustrated as below:

1. Enable the System Clock, GPIO Port A, GPIO Port C and UART Module 0.
2. Configure the pins PA0 and PA1 as UART pins and GPIO PC7 as output and initializes to logic low.
3. Wait till data is available on the UART buffer.
4. Read the data byte from the UART buffer if it is '1' turn on the LED and echo back "ON" string on PC host.
5. For any other character turn off the LED and echo back "OFF". And, again wait for the next character in UART buffer.

Figure 4 shows the program flow for the experiment.

3 Experiment 79—Temperature Display

3.1 Objective

Use the Tiva breakout board to design the temperature display using seven-segment displays. Display the temperature in both Celsius and Fahrenheit.

3.2 Hardware Description

This experiment requires four seven-segment displays, LM35 to sense the temperature, a switch to control the display format of temperature (or to select between Celsius and Fahrenheit) and Tiva LaunchPad to program the Tiva breakout board. Develop the motherboard for the Tiva breakout board which can be plug below Tiva breakout board. Motherboard should be having its own 3.3 and 5 V regulated supply. The schematic for motherboard is shown in Fig. 5. In schematic 5 V supply is taken from USB and same is supplied to LM1117 to generate regulated +3.3 V. Four segment displays are used, three for number display and last one to denote whether the temperature is in Celsius (display 'C' on the last seven-segment display) or Fahrenheit (display 'F' on the last seven-segment display). In schematic, seven-segment displays are used in multiplexed fashion in order to save the number of GPIOs. As shown in Fig. 5 the data lines to all four seven-segment displays are common, and each seven segment display is selected using common anode pin through a PNP transistor.

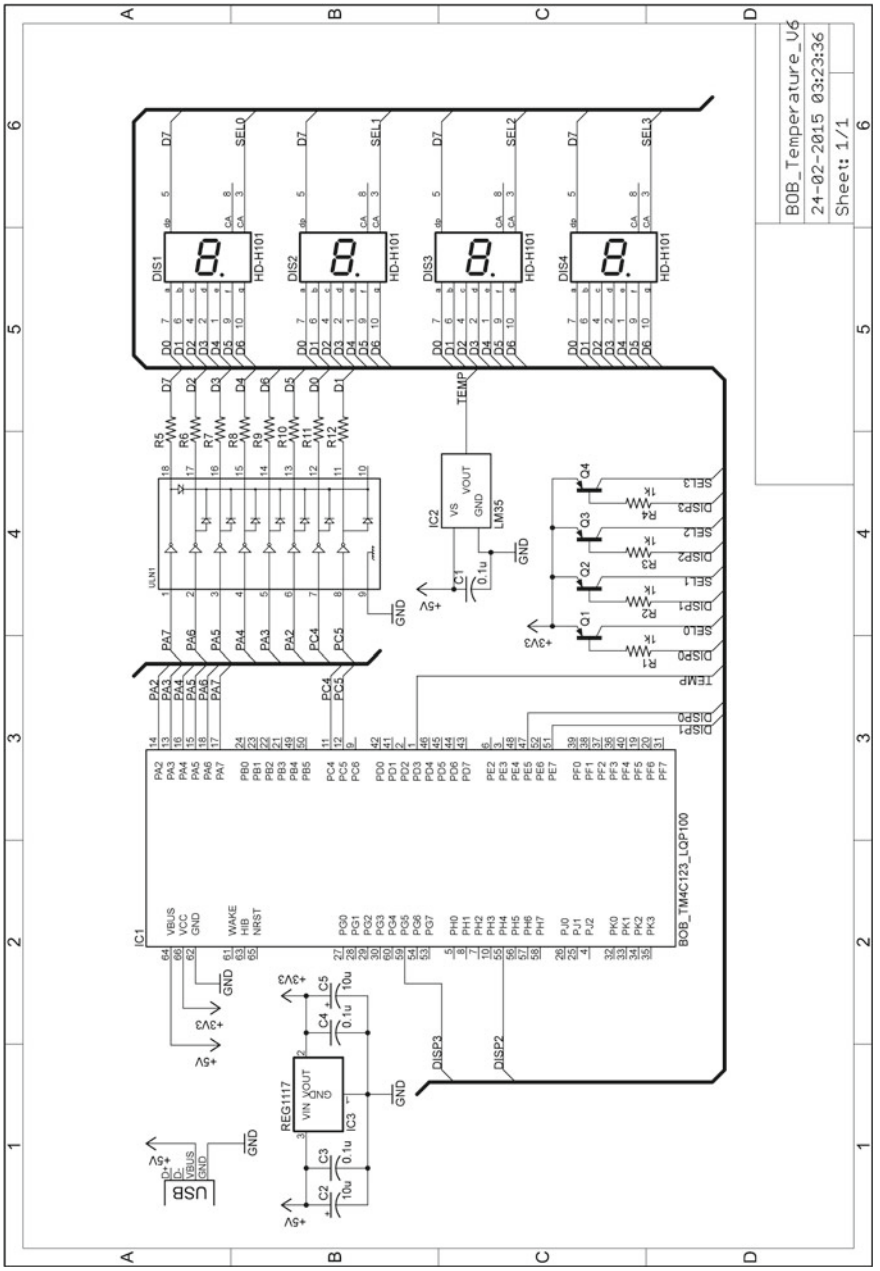

Fig. 5 Schematic for the temperature display

The PNP transistor is required because microcontroller is not capable to produce the sufficient current to light up the display. For similar reasons, microcontroller is not capable to sink such large current flowing out of display, hence transistor array IC ULN2803 is used with the data lines. The temperature sensor LM35 is connected to analog channel AIN12 which is multiplexed with the GPIO PD3. PA2, PA3, PA4, PA5, PA6, PA7, PC4, and PC5 are used as the data lines of the seven segment displays and PE5, PE7, PG5, and PH4 are used as select lines of the seven-segment displays. These GPIOs are selected so that the routing or borad layout of motherboard becomes easy, that is why these GPIOs seem to be a bit random. And, the switch is present on Tiva breakout board itself on PE0, so no need to add a switch on motherboard.

3.3 Experiment Tips

Read analog value from temperature sensor LM35 and convert it into Celsius and Fahrenheit value. If switch is pressed, display the Fahrenheit value, else display the Celsius value. While displaying the temperature value on multiplexed seven-segment display, extract the data required for each seven-segment display like each digit of temperature value. Then turn on the single seven-segment display and display the digit, then turn it off and now turn on the next seven-segment display and display the corresponding digit and so on. Run this display code in loop so that by persistence of vision temperature will be observed on seven segment displays connected in multiplexed format.

4 Experiment 80—Talking Range Finder

4.1 Objective

Talking devices are gaining a lot popularity these days. Develop a standalone system to measure the distance using ultrasonic module and relay the data through audio output.

4.2 Hardware Description

To perform this experiment ultrasonic module interface is required to measure the distance, DAC (preferably 12-bit DAC as it provides nice resolution for audio), microSD card interface to store the audio corresponding to each numeric value (0–9) and a switch to trigger the measurement and generate the corresponding audio output. The block diagram for hardware description of experiment is shown in Fig. 6.

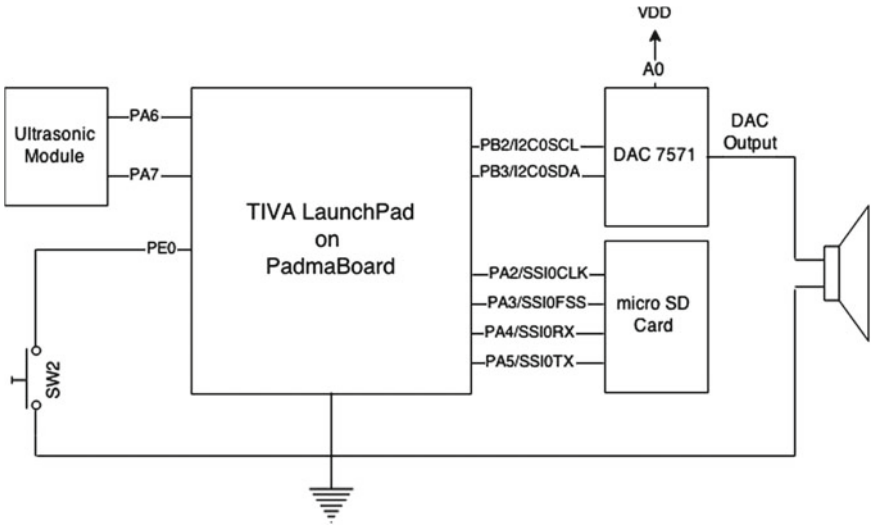

Fig. 6 Block diagram for the talking range finder

The DAC shown in block diagram is 7571, I2C based, which was discussed earlier. However there are SPI-based serial DAC also available which are faster than the I2C-based DAC. The pins and modules shown in block diagram are just for the examples.

Bibliography

For more detailed information regarding ARM Processor Family and specifically about the ARM Cortex-M4 Processor refer to below-mentioned material. These books and articles are also been used while developing this manual.

- ARM (2010, December 16), Cortex-M4 Devices Generic User Guide. http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf
- ARM (2010, March 02), Cortex-M4 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439b/DDI0439B_cortex_m4_r0p0_trm.pdf
- Dhananjay V. Gadre, Rohit Dureja, Shanjit S. Jajmann (2013), Getting started with the Stellaris Guru development kit, 1st Edition. Universities Press, Hyderabad, India
- Joseph Yiu (2013, November 1), Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd Edition. Newnes
- Trevor Martin (2013, May 13), Designer's Guide to the Cortex-M Processor Family, 1st Edition. Newnes
- Texas Instruments Pvt. Ltd (2014, April 15), Tiva C SeriesTM4C123G LaunchPad Evaluation Kit User's Manual. http://www.ti.com/lit/pdf/spmu296
- William Hohl (2009, March 19), ARM Assembly Language, 1st Edition. CRC Press

D.V. Gadre and S. Gupta, *Getting Started with Tiva ARM Cortex M4 Microcontrollers*, https://doi.org/10.1007/978-81-322-3766-2